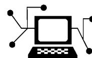
Computer Vision System Toolbox™

User's Guide

R2012a

MATLAB®

How to Contact MathWorks



`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Computer Vision System Toolbox™ User's Guide

© COPYRIGHT 2000–2012 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` 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` for more information.

Revision History

July 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Second printing	Revised for Version 1.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.2 (Release 14SP3)
November 2005	Online only	Revised for Version 2.0 (Release 14SP3+)
March 2006	Online only	Revised for Version 2.1 (Release 2006a)
September 2006	Online only	Revised for Version 2.2 (Release 2006b)
March 2007	Online only	Revised for Version 2.3 (Release 2007a)
September 2007	Online only	Revised for Version 2.4 (Release 2007b)
March 2008	Online only	Revised for Version 2.5 (Release 2008a)
October 2008	Online only	Revised for Version 2.6 (Release 2008b)
March 2009	Online only	Revised for Version 2.7 (Release 2009a)
September 2009	Online only	Revised for Version 2.8 (Release 2009b)
March 2010	Online only	Revised for Version 3.0 (Release 2010a)
September 2010	Online only	Revised for Version 3.1 (Release 2010b)
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)

Contents

1 Input, Output, and Conversions

File Opening, Loading and Saving **1-2**
Import from Video Files **1-2**
Export to Video Files **1-5**
Batch Process Image Files **1-6**
Display a Sequence of Images **1-9**
Partition Video Frames to Multiple Image Files **1-11**
Combine Video and Audio Streams into a Single Video File **1-15**
Import MATLAB Workspace Variables **1-17**
Send or Receive Audio and Video Content Over a Network **1-18**
Import a Live Video Stream **1-20**

Colorspace Formatting and Conversions **1-21**
Resample Image Chroma **1-21**
Convert Intensity to Binary Images **1-25**
Convert R'G'B' to Intensity Images **1-37**
Process Multidimensional Color Video Signals **1-42**

Data Formats **1-49**
Video Formats **1-49**
Video Data Stored in Column-Major Format **1-50**
Image Formats **1-50**

2 Display and Graphics

Display **2-2**
View Streaming Video in MATLAB using Video Player and Deployable Video Player System Objects **2-2**
Preview Video in MATLAB using MPlay Function **2-2**

View Video in Simulink using the Video Viewer and To Video Display Blocks ... **2-4**
View Video in Simulink using MPlay Function as a Floating Scope ... **2-4**
MPlay ... **2-7**

Graphics ... **2-24**
Abandoned Object Detection ... **2-24**
Annotate Video Files with Frame Numbers ... **2-30**

3 Registration and Stereo Vision

Feature Detection, Extraction, and Matching ... **3-2**
Detect Edges in Images ... **3-2**
Detect Lines in Images ... **3-8**
Detect Corner Features in an Image ... **3-11**
Find Possible Point Matches Between Two Images ... **3-12**
Measure an Angle Between Lines ... **3-14**

Image Registration ... **3-26**
Automatically Determine Geometric Transform for Image Registration ... **3-26**
Transform Images and Display Registration Results ... **3-27**
Remove the Effect of Camera Motion from a Video Stream. ... **3-28**

Stereo Vision ... **3-29**
Compute Disparity Depth Map ... **3-29**
Find Fundamental Matrix Describing Epipolar Geometry ... **3-30**
Rectify Stereo Images ... **3-32**

4 Motion Estimation and Tracking

Detect and Track Moving Objects Using Gaussian Mixture Models 4-2

Video Mosaicking 4-3

Track an Object Using Correlation 4-4

Create a Panoramic Scene 4-12

5 Geometric Transformations

Rotate an Image 5-2

Resize an Image 5-10

Crop an Image 5-16

Interpolation Methods 5-22
Nearest Neighbor Interpolation 5-22
Bilinear Interpolation 5-23
Bicubic Interpolation 5-24

Automatically Determine Geometric Transform for Image Registration 5-26

6 Filters, Transforms, and Enhancements

Adjust the Contrast of Intensity Images 6-2

Adjust the Contrast of Color Images **6-8**

Remove Periodic Noise from a Video **6-14**

Remove Salt and Pepper Noise from Images **6-23**

Sharpen an Image **6-30**

7 Statistics and Morphological Operations

Find the Histogram of an Image **7-2**

Correct Nonuniform Illumination **7-9**

Count Objects in an Image **7-17**

8 Fixed-Point Design

Fixed-Point Signal Processing **8-2**
Fixed-Point Features **8-2**
Benefits of Fixed-Point Hardware **8-2**
Benefits of Fixed-Point Design with System Toolboxes
Software **8-3**

Fixed-Point Concepts and Terminology **8-4**
Fixed-Point Data Types **8-4**
Scaling **8-5**
Precision and Range **8-6**

Arithmetic Operations **8-10**
Modulo Arithmetic **8-10**

Two's Complement 8-11
Addition and Subtraction 8-12
Multiplication 8-13
Casts 8-16

Fixed-Point Support for MATLAB System Objects **8-21**
Getting Information About Fixed-Point System Objects 8-21
Displaying Fixed-Point Properties 8-22
Setting System Object Fixed-Point Properties 8-23

Specify Fixed-Point Attributes for Blocks **8-25**
Fixed-Point Block Parameters 8-25
Specify System-Level Settings 8-28
Inherit via Internal Rule 8-29
Select and Specify Data Types for Fixed-Point Blocks 8-40

9 Code Generation

Code Generation with System Objects **9-2**

Functions that Generate Code **9-7**

Shared Library Dependencies **9-8**

Accelerating Simulink Models **9-9**

10 Define New System Objects

Define Basic System Objects **10-2**

Change Number of Step Method Inputs or Outputs **10-4**

Validate Property and Input Values **10-7**

Initialize Properties and Setup One-Time Calculations **10-10**

Set Property Values at Construction from Name-Value Pairs **10-13**

Reset Algorithm State **10-16**

Define Property Attributes **10-18**

Hide Inactive Properties **10-21**

Limit Property Values to a Finite Set of Strings **10-23**

Process Tuned Properties **10-26**

Release System Object Resources **10-28**

Define Composite System Objects **10-30**

Define Finite Source Objects **10-34**

Methods Timing **10-36**
Setup Method Call Sequence **10-36**
Step Method Call Sequence **10-37**
Reset Method Call Sequence **10-37**
Release Method Call Sequence **10-38**

Index

Input, Output, and Conversions

Learn how to import and export videos, and perform color space and video image conversions.

- "File Opening, Loading and Saving" on page 1-2
- "Colorspace Formatting and Conversions" on page 1-21
- "Data Formats" on page 1-49

File Opening, Loading and Saving

In this section...

"Import from Video Files" on page 1-2
"Export to Video Files" on page 1-5
"Batch Process Image Files" on page 1-6
"Display a Sequence of Images" on page 1-9
"Partition Video Frames to Multiple Image Files" on page 1-11
"Combine Video and Audio Streams into a Single Video File" on page 1-15
"Import MATLAB Workspace Variables" on page 1-17
"Send or Receive Audio and Video Content Over a Network" on page 1-18
"Import a Live Video Stream" on page 1-20

Import from Video Files

In this example, you use the From Multimedia File source block to import a video stream into a Simulink® model and the To Video Display sink block to view it. This procedure assumes you are working on a Windows platform.

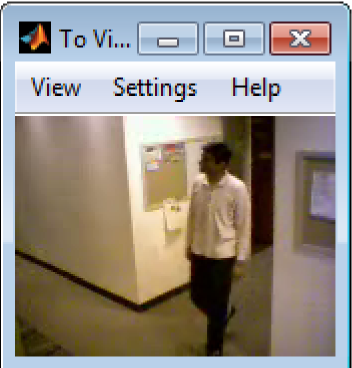
You can open the example model by typing

```
ex_import_mmf
```

at the MATLAB® command line.

1 Run your model.

2 View your video in the To Video Display window that automatically appears when you start your simulation.



Note The video that is displayed in the To Video Display window runs at the frame rate that corresponds to the input sample time. To run the video as fast as Simulink processes the video frames, use the Video Viewer block.

You have now imported and displayed a multimedia file in the Simulink model. In the "Export to Video Files" on page 1-5 example you can manipulate your video stream and export it to a multimedia file.

For more information on the blocks used in this example, see the From Multimedia File and To Video Display block reference pages. To listen to audio associated with an AVI file, use the To Audio Device block in DSP System Toolbox™ software.

Setting Block Parameters for this Example

The block parameters in this example were modified from default values as follows:

<table>
<tr><th>Block</th><th>Parameter</th></tr>
<tr><td>From Multimedia File</td><td>Use the From Multimedia File block to import the multimedia file into the model:

• If you do not have your own multimedia file, use the default <code>vipmen.avi</code> file, for the File name parameter.
• If the multimedia file is on your MATLAB path, enter the filename for the File name parameter.
• If the file is not on your MATLAB path, use the Browse button to locate the multimedia file.
• Set the Image signal parameter to <code>Separate color signals</code>.

By default, the Number of times to play file parameter is set to <code>inf</code>. The model continues to play the file until the simulation stops.</td></tr>
<tr><td>To Video Display</td><td>Use the To Video Display block to view the multimedia file.

• Image signal: <code>Separate color signals</code>

Set this parameter from the Settings menu of the display viewer.</td></tr>
</table>

Configuration Parameters

You can locate the **Configuration Parameters** by selecting **Configuration Parameters** from the **Simulation** menu. For this example, the parameters on the **Solver** pane, are set as follows:

- **Stop time** = `20`
- **Type** = `Fixed-step`
- **Solver** = `Discrete (no continuous states)`

Export to Video Files

The Computer Vision System Toolbox™ blocks enable you to export video data from your Simulink model. In this example, you use the To Multimedia File block to export a multimedia file from your model. This example also uses Gain blocks from the **Math Operations** Simulink library.

You can open the example model by typing

```
ex_export_to_mmf
```

at the MATLAB command line.

1 Run your model.

2 You can view your video in the To Video Display window.

By increasing the red, green, and blue color values, you increase the contrast of the video. The To Multimedia File block exports the video data from the Simulink model to a multimedia file that it creates in your current folder.

This example manipulated the video stream and exported it from a Simulink model to a multimedia file. For more information, see the To Multimedia File block reference page.

Setting Block Parameters for this Example

The block parameters in this example were modified from default values as follows:

Block	Parameter
Gain	The Gain blocks are used to increase the red, green, and blue values of the video stream. This increases the contrast of the video: • **Main** pane, **Gain** = `1.2` • **Signal Attributes** pane, **Output data type** = `Inherit: Same as input`
To Multimedia File	The To Multimedia File block exports the video to a multimedia file: • **Output file name** = `my_output.avi` • **Write** = `Video only` • **Image signal** = `Separate color signals`

Configuration Parameters

You can locate the **Configuration Parameters** by selecting **Configuration Parameters** from the **Simulation** menu. For this example, the parameters on the **Solver** pane, are set as follows:

- **Stop time** = `20`
- **Type** = `Fixed-step`
- **Solver** = `Discrete (no continuous states)`

Batch Process Image Files

A common image processing task is to apply an image processing algorithm to a series of files. In this example, you import a sequence of images from a folder into the MATLAB workspace.

Note In this example, the image files are a set of 10 microscope images of rat prostate cancer cells. These files are only the first 10 of 100 images acquired.

---

1 Specify the folder containing the images, and use this information to create a list of the file names, as follows:

```
fileFolder = fullfile(matlabroot,'toolbox','images','imdemos');
dirOutput = dir(fullfile(fileFolder,'AT3_1m4_*.tif'));
fileNames = {dirOutput.name}'
```

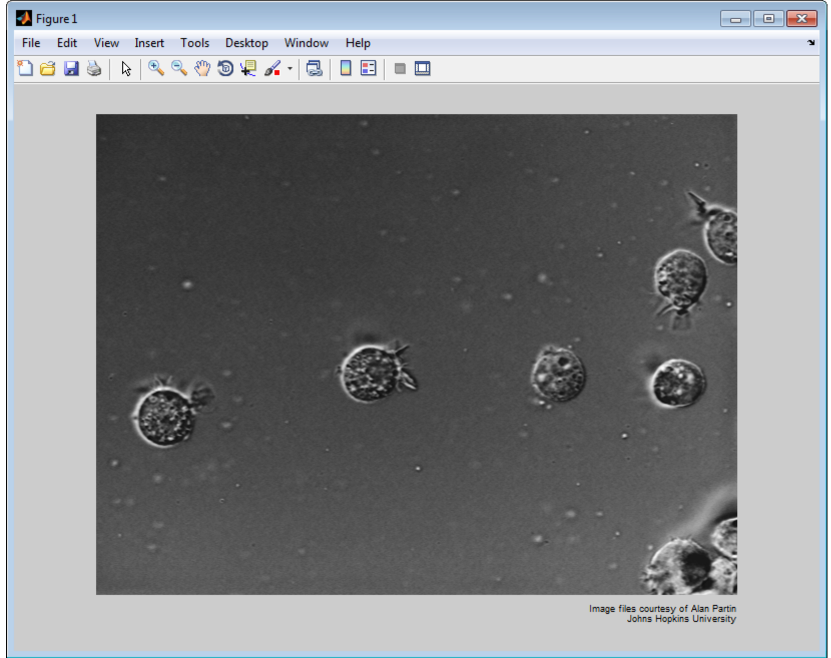
2 View one of the images, using the following command sequence:

```
I = imread(fileNames{1});
imshow(I);
text(size(I,2),size(I,1)+15, ...
    'Image files courtesy of Alan Partin', ...
    'FontSize',7,'HorizontalAlignment','right');
text(size(I,2),size(I,1)+25, ....
    'Johns Hopkins University', ...
    'FontSize',7,'HorizontalAlignment','right');
```

3 Use a for loop to create a variable that stores the entire image sequence. You can use this variable to import the sequence into Simulink.

```
for i = 1:length(fileNames)
    my_video(:,:,i) = imread(fileNames{i});
end
```

4 Run your model. You can view the image sequence in the Video Viewer window.



Because the Video From Workspace block's **Sample time** parameter is set to `1` and the "Configuration Parameters" on page 1-10 **Stop time** is set to `10`, the Video Viewer block displays 10 images before the simulation stops.

For more information on the blocks used in this example, see the Video From Workspace and Video Viewer block reference pages. For additional information about batch processing, see the Batch Processing Image Files Using Distributed Computing demo in Image Processing Toolbox. You can run this demo by typing `ipexbatch` at the MATLAB command prompt.

Configuration Parameters

You can locate the **Configuration Parameters** by selecting **Configuration Parameters** from the **Simulation** menu. For this example, the parameters on the **Solver** pane, are set as follows:

- **Stop time** = `10`
- **Type** = `Fixed-step`

- **Solver** = `Discrete (no continuous states)`

Display a Sequence of Images

This example displays a sequence of images, which were saved in a folder, and then stored in a variable in the MATLAB workspace. At load time, this model executes the code from the "Batch Process Image Files" on page 1-6 example, which stores images in a workspace variable.

You can open the example model by typing

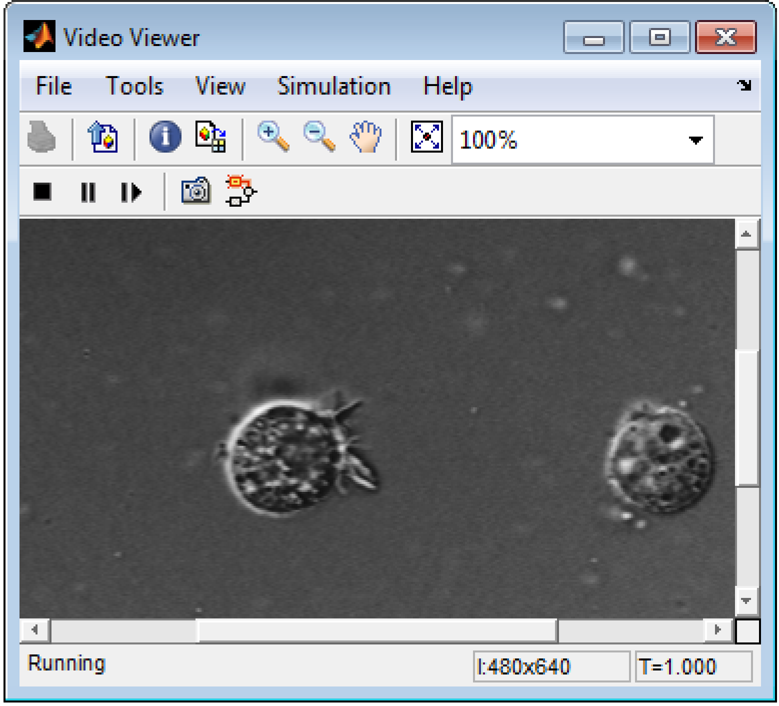
```
ex_display_sequence_of_images
```

at the MATLAB command line.

1. The Video From Workspace block reads the files from the MATLAB workspace. The **Signal** parameter is set to the name of the variable for the stored images. For this example, it is set to `my_video`.

2. The Video Viewer block displays the sequence of images.

3. Run your model. You can view the image sequence in the Video Viewer window.



4 Because the Video From Workspace block's **Sample time** parameter is set to 1 and the **Stop time** parameter in the configuration parameters, is set to 10, the Video Viewer block displays 10 images before the simulation stops.

Pre-loading Code

To find or modify the pre-loaded code, select the **Callbacks** tab in the **Model Properties** dialog located under **File** > **Model Properties**. For more details on how to set-up callbacks, see "Using Callback Functions".

Configuration Parameters

You can locate the **Configuration Parameters** by selecting **Configuration Parameters** from the **Simulation** menu. For this example, the parameters on the **Solver** pane, are set as follows:

- **Stop time** = 10

- **Type** = `Fixed-step`
- **Solver** = `Discrete (no continuous states)`

Partition Video Frames to Multiple Image Files

In this example, you use the To Multimedia File block, the Enabled Subsystem block, and a trigger signal, to save portions of one AVI file to three separate AVI files.

You can open the example model with the link below or by typing

```
ex_vision_partition_video_frames_to_multiple_files
```

at the MATLAB command line.

1. Run your model.
2. The model saves the three output AVI files in your current folder.
3. View the resulting files by typing the following commands at the MATLAB command prompt:

   ```
   mplay output1.avi
   mplay output2.avi
   mplay output3.avi
   ```

4. Press the **Play** button on the MPlay GUI.

For more information on the blocks used in this example, see the From Multimedia File, Insert Text, Enabled Subsystem, and To Multimedia File block reference pages.

Setting Block Parameters for this Example

The block parameters in this example were modified from default values as follows:

Block	Parameter
From Multimedia File	The From Multimedia File block imports an AVI file into the model. • Cleared **Inherit sample time from file** checkbox.
Insert Text	The example uses the Insert Text block to annotate the video stream with frame numbers. The block writes the frame number in green, in the upper-left corner of the output video stream. • **Text**: `'Frame %d'` • **Color**: `[0 1 0]` • **Location**: `[10 10]`
To Multimedia File	The To Multimedia File blocks send the video stream to three separate AVI files. These block parameters were modified as follows: • **Output file name**: `output1.avi`, `output2.avi`, and `output3.avi`, respectively • **Write**: `Video only`
Counter	The Counter block counts the number of video frames. The example uses this information to specify which frames are sent to which file. The block parameters are modified as follows: • **Number of bits**: `8` • **Sample time**: `1/30`

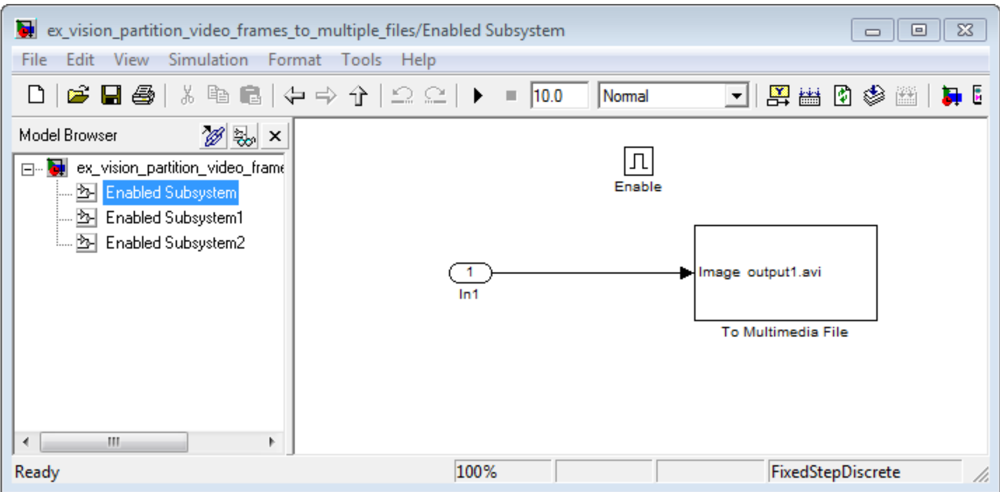
Block	Parameter
Bias	The bias block adds a bias to the input. The block parameters are modified as follows: • **Bias**: 1
Compare to Constant	The Compare to Constant block sends frames 1 to 9 to the first AVI file. The block parameters are modified as follows: • **Operator**: < • **Constant value**: 10
Compare to Constant1 **Compare to Constant2**	The Compare to Constant1 and Compare to Constant2 blocks send frames 10 to 19 to the second AVI file. The block parameters are modified as follows: • **Operator**: >= • **Constant value**: 10 The Compare to Constant2 block parameters are modified as follows: • **Operator**: < • **Constant value**: 20

Block	Parameter
Compare to Constant3	The Compare to Constant3 block send frames 20 to 30 to the third AVI file. The block parameters are modified as follows: • **Operator**: `>=` • **Constant value**: `20`
Compare to Constant4	The Compare to Constant4 block stopa the simulation when the video reaches frame 30. The block parameters are modified as follows: • **Operator**: `==` • **Constant value**: `30` • **Output data type mode**: `boolean`

Using the Enabled Subsystem Block

Each To Multimedia File block gets inserted into one Enabled Subsystem block, and connected to it's input. You can do this, by double-clicking the Enabled Subsystem blocks, then click-and-drag a To Multimedia File block into it.

Each enabled subsystem should look similar to the subsystem shown in the following figure.



Configuration Parameters

You can locate the **Configuration Parameters** by selecting **Configuration Parameters** from the **Simulation** menu. For this example, the parameters on the **Solver** pane, are set as follows:

- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

Combine Video and Audio Streams into a Single Video File

In this example, you use the From Multimedia File blocks to import video and audio streams into a Simulink model. You then write the audio and video to a single file using the To Multimedia File block.

You can open the example model by typing

```
ex_combine_video_and_audio_streams
```

on the MATLAB command line.

1 Run your model. The model creates a multimedia file called `output.avi` in your current folder.

2 Play the multimedia file using a media player. The original video file now has an audio component to it.

Setting Up the Video Input Block

The From Multimedia File block imports a video file into the model. During import, the **Inherit sample time from file** check box is deselected, which enables the **Desired sample time** parameter. The other default parameters are accepted.

The From Multimedia File block used for the input video file inherits its sample time from the `vipmen.avi` file. For video signals, the sample time equals the frame period. The frame period is defined as 1/(frame rate). Because the input video frame rate is 30 frames per second (fps), the block sets the frame period to `1/30` or `0.0333` seconds per frame.

Setting Up the Audio Input Block

The From Multimedia File1 block imports an audio file into the model.

The **Samples per audio frame** parameter is set to `735`. This output audio frame size is calculated by dividing the frequency of the audio signal (22050 samples per second) by the frame rate (approximately 30 frames per second).

You must adjust the audio signal frame period to match the frame period of the video signal. The video frame period is `0.0333` seconds per frame. Because the frame period is also defined as the frame size divided by frequency, you can calculate the frame period of the audio signal by dividing the frame size of the audio signal (735 samples per frame) by the frequency (22050 samples per second) to get `0.0333` seconds per frame.

frame period = (*frame size*)/(*frequency*)
frame period = (`735` *samples per frame*)/(22050 *samples per second*)
frame period = `0.0333` seconds per frame

Alternatively, you can verify that the frame period of the audio and video signals is the same using a Simulink Probe block.

Setting Up the Output Block

The To Multimedia File block is used to output the audio and video signals to a single multimedia file. The `Video and audio` option is selected for the

Write parameter and `One multidimensional signal` for the **Image signal** parameter. The other default parameters are accepted.

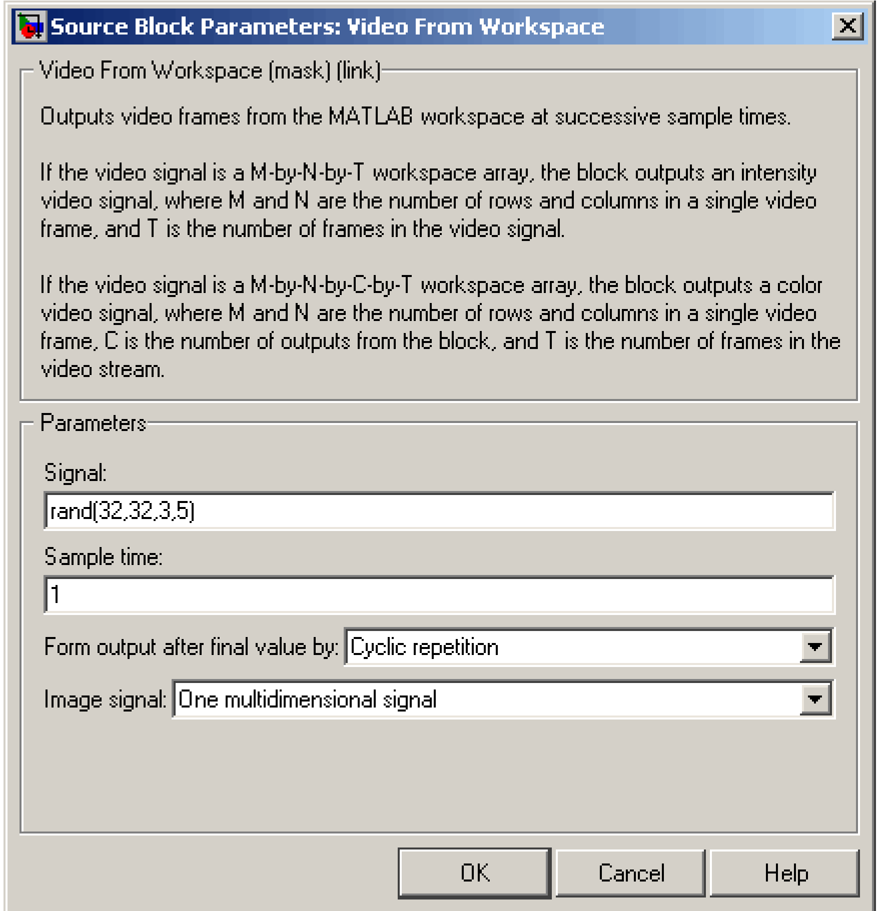
Configuration Parameters

You can locate the Configuration Parameters by selecting **Simulation > Configuration Parameters**. The parameters, on the **Solver** pane, are set as follows:

- **Stop time** = `10`
- **Type** = `Fixed-step`
- **Solver** = `Discrete (no continuous states)`

Import MATLAB Workspace Variables

You can import data from the MATLAB workspace using the Video From Workspace block, which is created specifically for this task.



Use the **Signal** parameter to specify the MATLAB workspace variable from which to read. For more information about how to use this block, see the Video From Workspace block reference page.

Send or Receive Audio and Video Content Over a Network

MATLAB and Simulink support network streaming via the Microsoft® MMS protocol (which is also known as the ASF, or advanced streaming format, protocol). This ability is supported on Windows® operating systems. If you

are using other operating systems, you can use UDP to transport your media streams. If you are using Simulink, use the To Multimedia File and From Multimedia File blocks. If you are using MATLAB, use the `VideoFileWriter` and the `VideoFileReader` System objects. It is possible to encode and view these streams with other applications.

In order to view an MMS stream generated by MATLAB, you should use Internet Explorer®, and provide the URL (e.g. "mms://127.0.0.1:81") to the stream which you wish to read. If you wish to create an MMS stream which can be viewed by MATLAB, download the Windows Media® Encoder or Microsoft Expression Encoder application, and configure it to produce a stream on a particular port (e.g. 81). Then, specify that URL in the **Filename** field of the From Multimedia File block or `VideoFileReader` System object.

If you run this example with **sendReceive** set to `'send'`, you can open up Internet Explorer and view the URL on which you have set up a server. By default, you should go to the following URL: `mms://127.0.0.1:80`. If you run this example with **sendReceive** set to `'receive'`, you will be able to view a MMS stream on the local computer on port `81`. This implies that you will need to set up a stream on this port using software such as the Windows Media Encoder (which may be downloaded free of charge from Microsoft).

Specify the **sendReceive** parameter to either ‘send’ to write the stream to the network or ‘receive’ to read the stream from the network.

```
    sendReceive = 'send';
    url = 'mms://127.0.0.1:81';
    filename = 'vipmen.avi';
```

Either send or receive the stream, as specified.

```
if strcmpi(sendReceive, 'send')
    % Create objects
    hSrc = vision.VideoFileReader(filename);
    hSnk = vision.VideoFileWriter;

    % Set parameters
    hSnk.FileFormat = 'WMV';
    hSnk.AudioInputPort = false;
    hSnk.Filename = url;
```

```
    % Run loop. Ctrl-C to exit
    while true
        data = step(hSrc);
        step(hSnk, data);
    end

else
    % Create objects
    hSrc = vision.VideoFileReader;
    hSnk = vision.DeployableVideoPlayer;

    % Set parameters
    hSrc.Filename = url;

    % Run loop. Ctrl-C to exit
    while true
        data = step(hSrc);
        step(hSnk, data);
    end
end
```

You cannot send and receive MMS streams from the same process. If you wish to send and receive, the sender or the receiver must be run in rapid accelerator mode or compiled as a separate application using Simulink Coder™.

Import a Live Video Stream

Image Acquisition Toolbox provides functions for acquiring images and video directly into MATLAB and Simulink from PC-compatible imaging hardware. You can detect hardware automatically, configure hardware properties, preview an acquisition, and acquire images and video.

See the live video processing demos to view demos that use the Image Acquisition Toolbox together with Computer Vision System Toolbox blocks. To see the full list of Computer Vision System Toolbox demos, type `visiondemos` at the MATLAB command prompt.

Colorspace Formatting and Conversions

In this section...

"Resample Image Chroma" on page 1-21

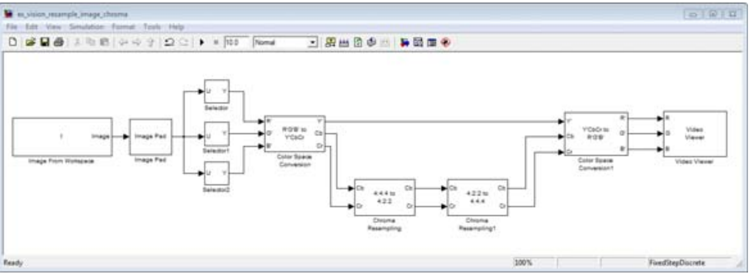
"Convert Intensity to Binary Images" on page 1-25

"Convert R'G'B' to Intensity Images" on page 1-37

"Process Multidimensional Color Video Signals" on page 1-42

Resample Image Chroma

In this example, you use the Chroma Resampling block to downsample the Cb and Cr components of an image. The Y'CbCr color space separates the luma (Y') component of an image from the chroma (Cb and Cr) components. Luma and chroma, which are calculated using gamma corrected R, G, and B (R', G', B') signals, are different quantities than the CIE chrominance and luminance. The human eye is more sensitive to changes in luma than to changes in chroma. Therefore, you can reduce the bandwidth required for transmission or storage of a signal by removing some of the color information. For this reason, this color space is often used for digital encoding and transmission applications.



You can open the example model by typing

```
ex_vision_resample_image_chroma
```

on the MATLAB command line.

1. Define an RGB image in the MATLAB workspace. To do so, at the MATLAB command prompt, type:

   ```
   I= imread('autumn.tif');
   ```

   This command reads in an RGB image from a TIF file. The image `I` is a 206-by-345-by-3 array of 8-bit unsigned integer values. Each plane of this array represents the red, green, or blue color values of the image.

2. To view the image this array represents, at the MATLAB command prompt, type:

   ```
   imshow(I)
   ```

3. Configure Simulink to display signal dimensions next to each signal line. Select **Format > Port/Signal Displays > Signal Dimensions**.

4. Run your model. The recovered image appears in the Video Viewer window. The Chroma Resampling block has downsampled the Cb and Cr components of an image.

5. Examine the signal dimensions in your model. The Chroma Resampling block downsamples the Cb and Cr components of the image from 206-by-346 matrices to 206-by-173 matrices. These matrices require less bandwidth for transmission while still communicating the information necessary to recover the image after it is transmitted.

Setting Block Parameters for This Example

The block parameters in this example are modified from default values as follows:

<table>
<tr><th>Block</th><th>Parameter</th></tr>
<tr><td>Image from Workspace</td><td>Import your image from the MATLAB workspace. Set the Value parameter to <code>I</code>.</td></tr>
<tr><td>Image Pad</td><td>Change dimensions of the input <code>I</code> array from 206-by-345-by-3 to 206-by-346-by-3. You are changing these dimensions because the Chroma Resampling block requires that the dimensions of the input be divisible by 2. Set the block parameters as follows:

• Method = <code>Symmetric</code>
• Add columns to = <code>Right</code>
• Number of added columns = <code>1</code>
• Add row to = <code>No padding</code>

The Image Pad block adds one column to the right of each plane of the array by repeating its border values. This padding minimizes the effect of the pixels outside the image on the processing of the image.

Note When you process video streams, be aware that it is computationally expensive to pad every video frame. You should change the dimensions of the video stream before you process it with Computer Vision System Toolbox blocks.</td></tr>
</table>

Block	Parameter
Selector, Selector1, Selector2	Separate the individual color planes from the main signal. Such separation simplifies the color space conversion section of the model. Set the Selector block parameters as follows:**Selector1** • **Number of input dimensions** = `3` • **Index 1** = `Select all` • **Index 2** = `Select all` • **Index 3** = `Index vector (dialog)` and **Index** = `1` **Selector2** • **Number of input dimensions** = `3` • **Index 1** = `Select all` • **Index 2** = `Select all` • **Index 3** = `Index vector (dialog)` and **Index** = `2` **Selector2** • **Number of input dimensions** = `3` • **Index 1** = `Select all` • **Index 2** = `Select all` • **Index 3** = `Index vector (dialog)` and **Index** = `3`
Color Space Conversion	Convert the input values from the R'G'B' color space to the Y'CbCr color space. The prime symbol indicates a gamma corrected signal. Set the **Image signal** parameter to `Separate color signals`.

Block	Parameter
Chroma Resampling	Downsample the chroma components of the image from the 4:4:4 format to the 4:2:2 format. Use the default parameters. The dimensions of the output of the Chroma Resampling block are smaller than the dimensions of the input. Therefore, the output signal requires less bandwidth for transmission.
Chroma Resampling1	Upsample the chroma components of the image from the 4:2:2 format to the 4:4:4 format. Set the **Resampling** parameter to `4:2:2 to 4:4:4`.
Color Space Conversion1	Convert the input values from the Y'CbCr color space to the R'G'B' color space. Set the block parameters as follows: • **Conversion** = `Y'CbCr to R'G'B'` • **Image signal** = `Separate color signals`
Video Viewer	Display the recovered image. Select **File>Image signal** to set **Image signal** to `Separate color signals`.

Configuration Parameters

Open the Configuration dialog box by selecting **Configuration Parameters...** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

Convert Intensity to Binary Images

- "Thresholding Intensity Images Using Relational Operators" on page 1-26
- "Thresholding Intensity Images Using the Autothreshold Block" on page 1-31

Binary images contain Boolean pixel values that are either 0 or 1. Pixels with the value 0 are displayed as black; pixels with the value 1 are displayed

as white. Intensity images contain pixel values that range between the minimum and maximum values supported by their data type. Binary images can contain only 0s and 1s, but they are not binary images unless their data type is Boolean.

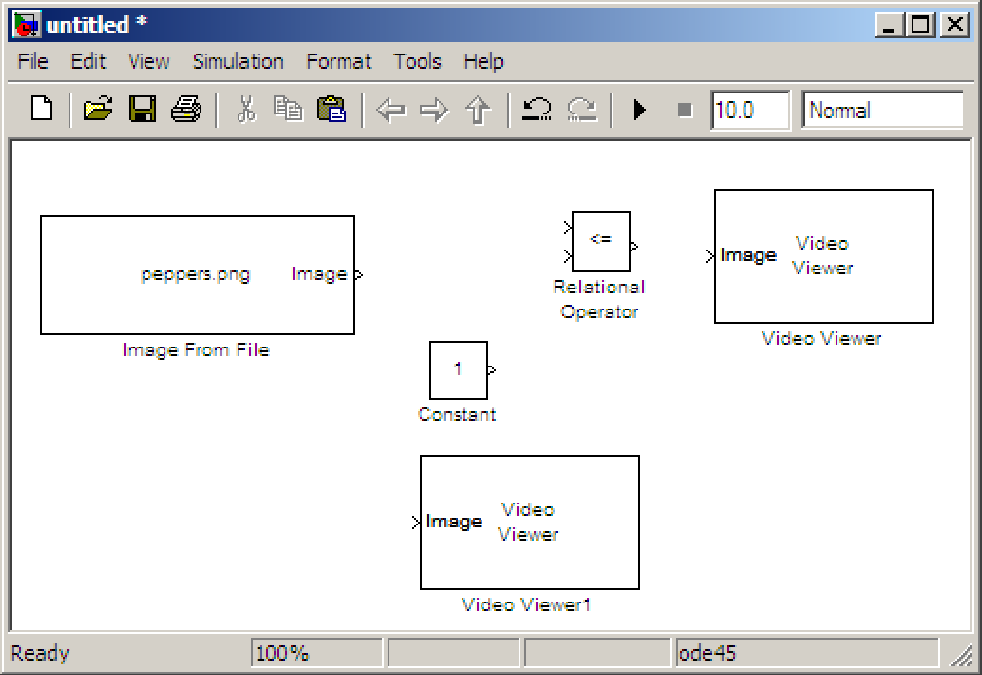
Thresholding Intensity Images Using Relational Operators

You can use the Relational Operator block to perform a thresholding operation that converts your intensity image to a binary image. This example shows you how to accomplish this task.

1 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Video Viewer	Computer Vision System Toolbox > Sinks	2
Relational Operator	Simulink > Logic and Bit Operations	1
Constant	Simulink > Sources	1

2 Position the blocks as shown in the following figure.

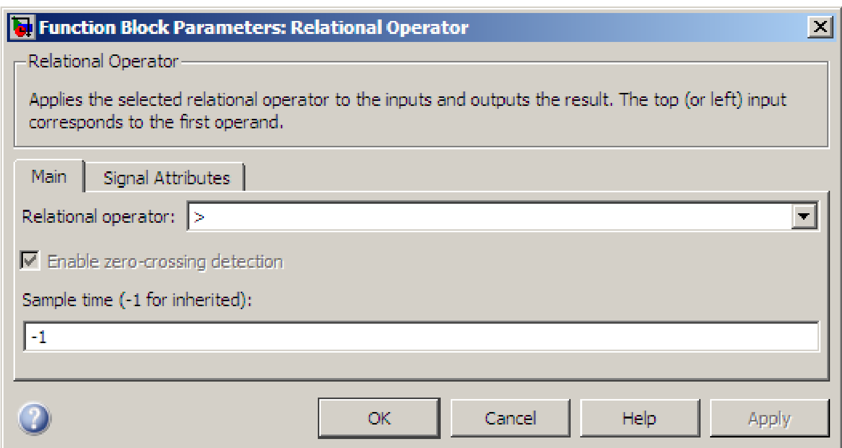


3 Use the Image from File block to import your image. In this example the image file is a 256-by-256 matrix of 8-bit unsigned integer values that range from 0 to 255. Set the **File name** parameter to `rice.png`

4 Use the Video Viewer1 block to view the original intensity image. Accept the default parameters.

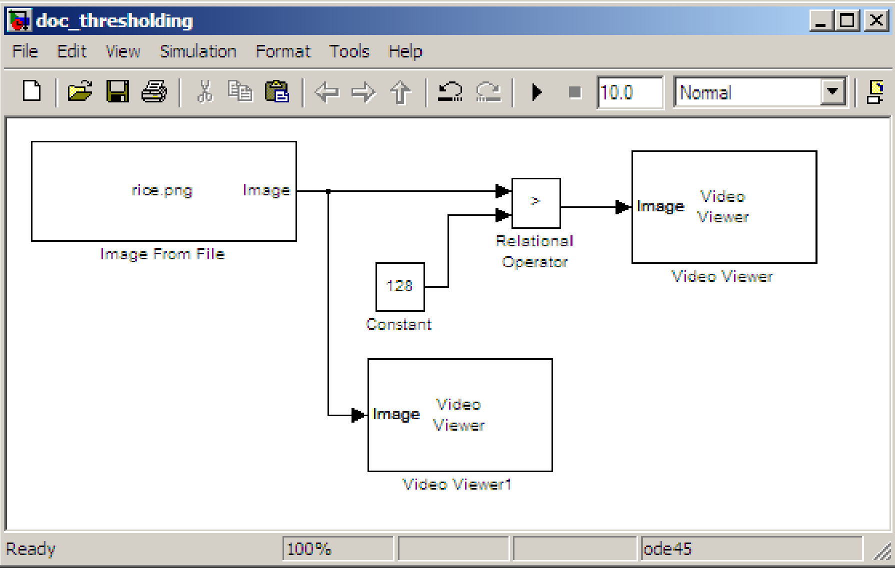
5 Use the Constant block to define a threshold value for the Relational Operator block. Since the pixel values range from 0 to 255, set the **Constant value** parameter to `128`. This value is image dependent.

6 Use the Relational Operator block to perform a thresholding operation that converts your intensity image to a binary image. Set the **Relational Operator** parameter to `>`. If the input to the Relational Operator block is greater than 128, its output is a Boolean 1; otherwise, its output is a Boolean 0.



7 Use the Video Viewer block to view the binary image. Accept the default parameters.

8 Connect the blocks as shown in the following figure.

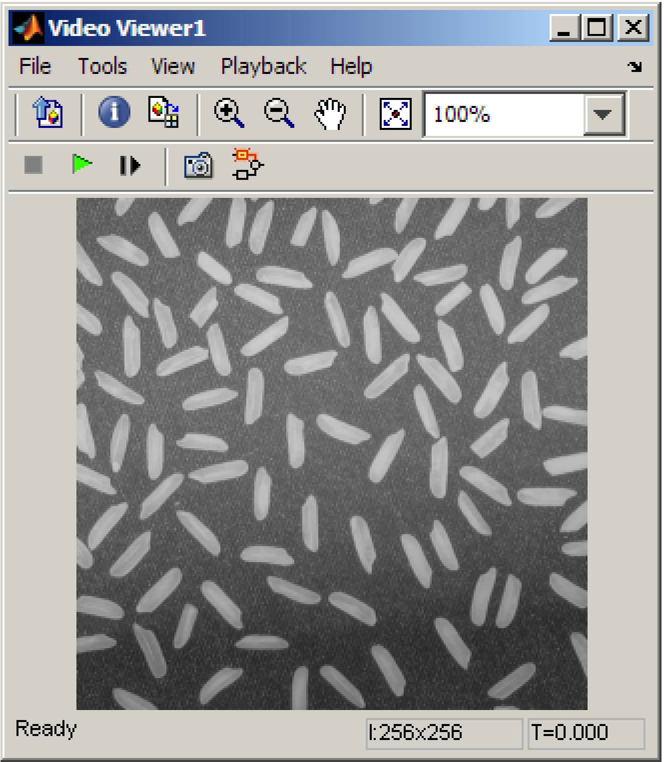


9 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

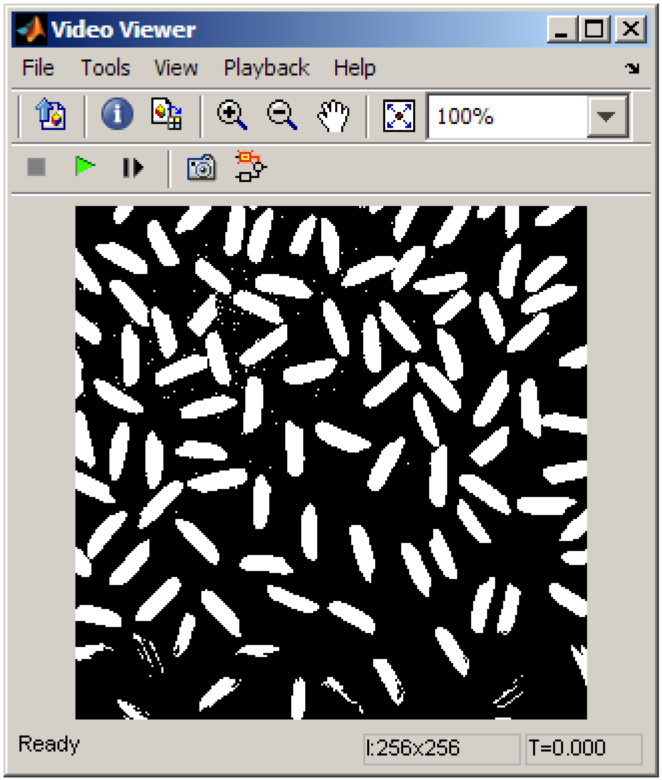
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

10 Run your model.

The original intensity image appears in the Video Viewer1 window.



The binary image appears in the Video Viewer window.



Note A single threshold value was unable to effectively threshold this image due to its uneven lighting. For information on how to address this problem, see "Correct Nonuniform Illumination" on page 7-9.

You have used the Relational Operator block to convert an intensity image to a binary image. For more information about this block, see the Relational Operator block reference page in the Simulink documentation. For additional information, see "Converting Between Image Types" in the Image Processing Toolbox documentation.

Thresholding Intensity Images Using the Autothreshold Block

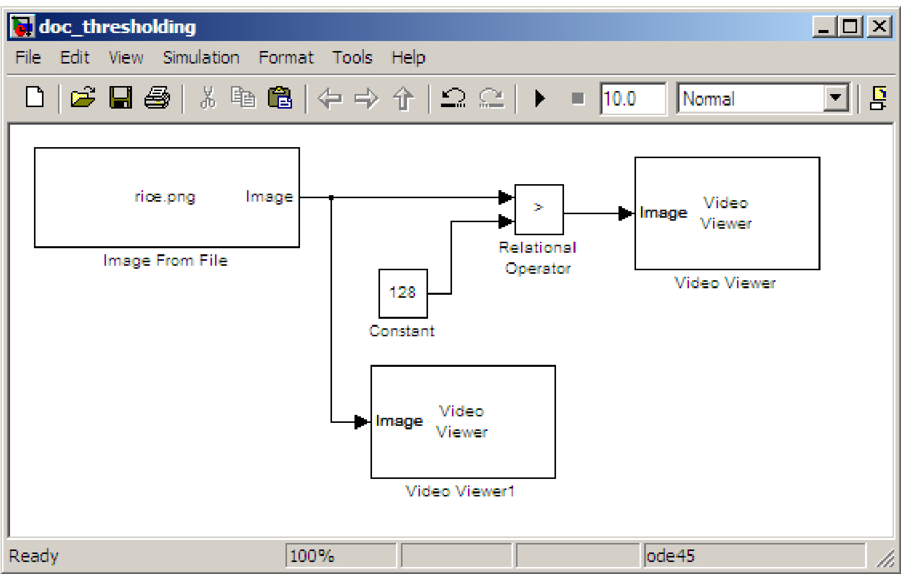
In the previous topic, you used the Relational Operator block to convert an intensity image into a binary image. In this topic, you use the Autothreshold block to accomplish the same task. Use the Autothreshold block when lighting conditions vary and the threshold needs to change for each video frame.

Running this example requires a DSP System Toolbox license.

1 If the model you created in “Thresholding Intensity Images Using Relational Operators” on page 1-26 is not open on your desktop, you can open an equivalent model by typing

```
ex_vision_thresholding
```

at the MATLAB command prompt.

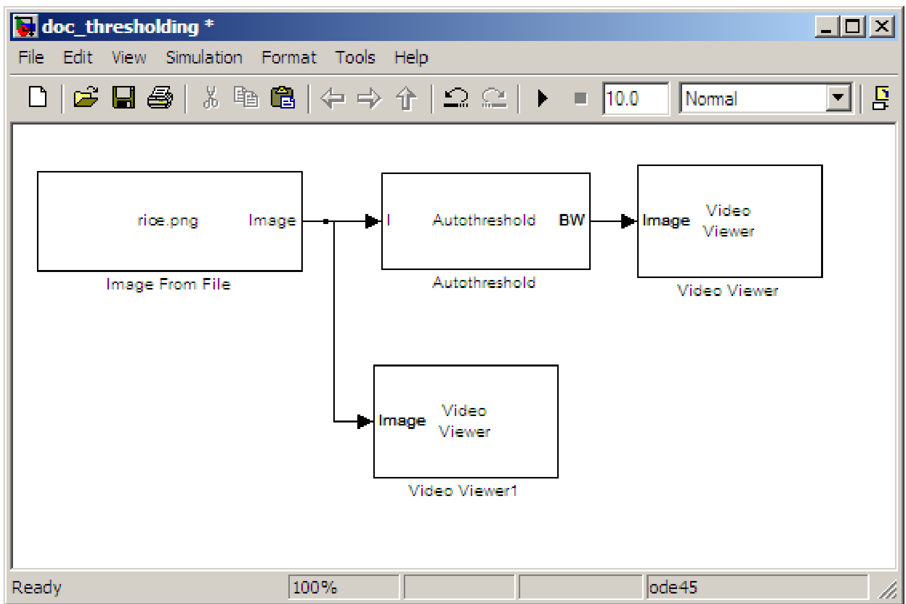


2 Use the Image from File block to import your image. In this example the image file is a 256-by-256 matrix of 8-bit unsigned integer values that range from 0 to 255. Set the **File name** parameter to `rice.png`

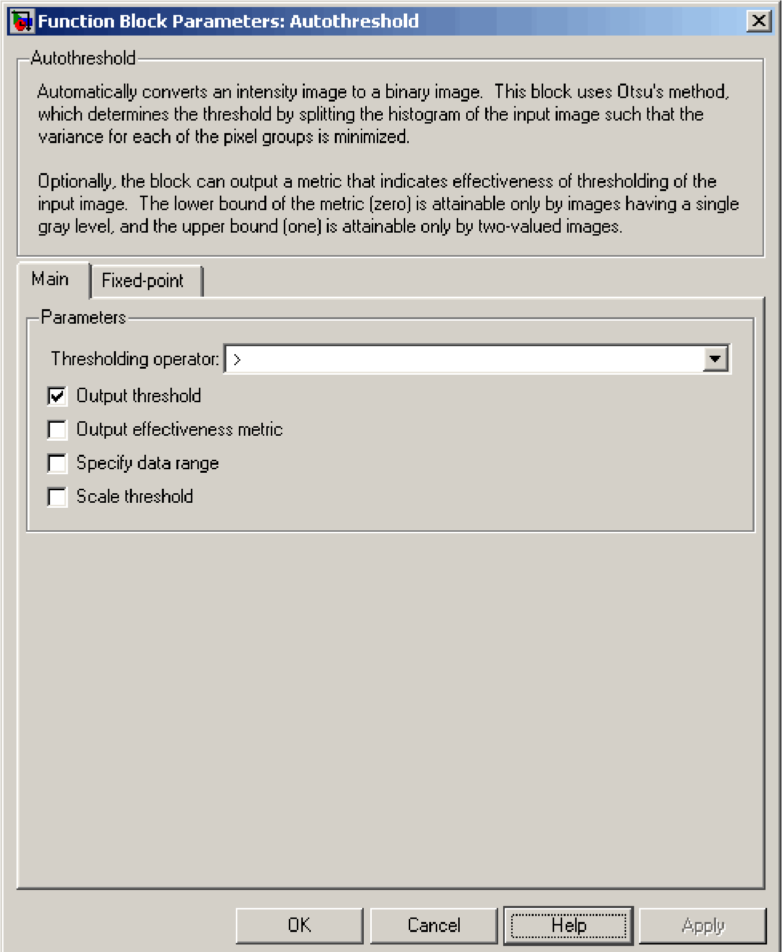
3 Delete the Constant and the Relational Operator blocks in this model.

4 Add an Autothreshold block from the Conversions library of the Computer Vision System Toolbox into your model.

5 Connect the blocks as shown in the following figure.



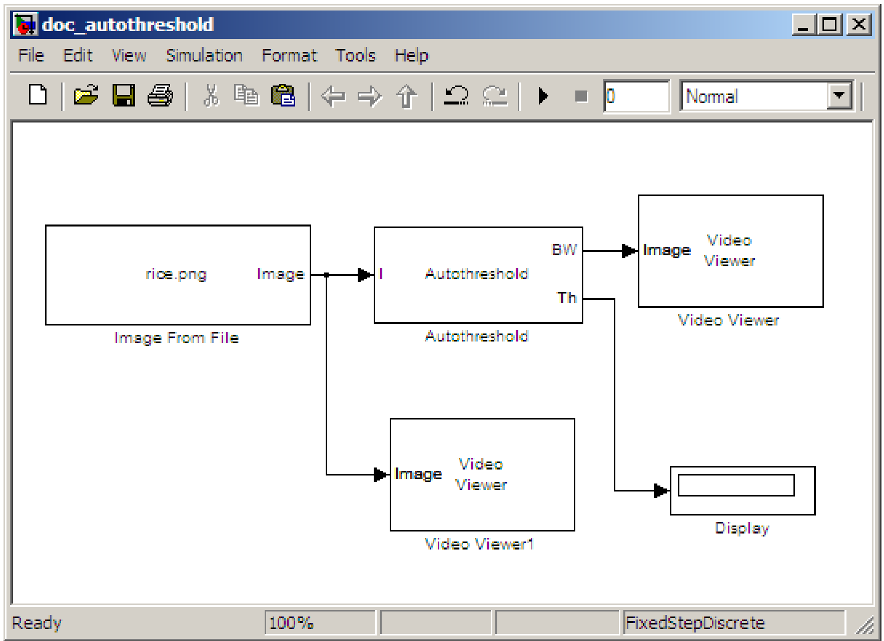
6 Use the Autothreshold block to perform a thresholding operation that converts your intensity image to a binary image. Select the **Output threshold** check box.



The block outputs the calculated threshold value at the **Th** port.

7 Add a Display block from the Sinks library of the DSP System Toolbox library. Connect the Display block to the **Th** output port of the Authothreshold block.

Your model should look similar to the following figure:



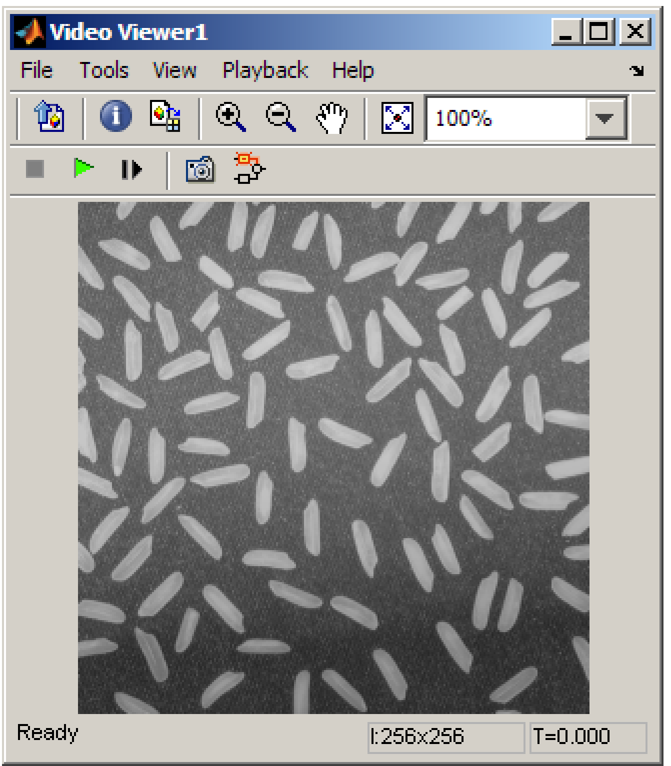
8 Double-click the Image From File block. On the **Data Types** pane, set the **Output data type** parameter to `double`.

9 If you have not already done so, set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

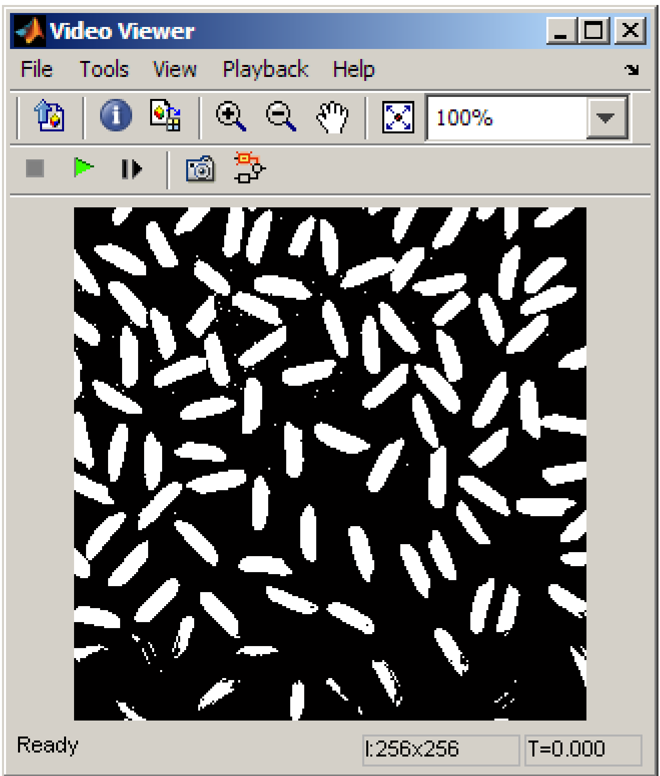
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

10 Run the model.

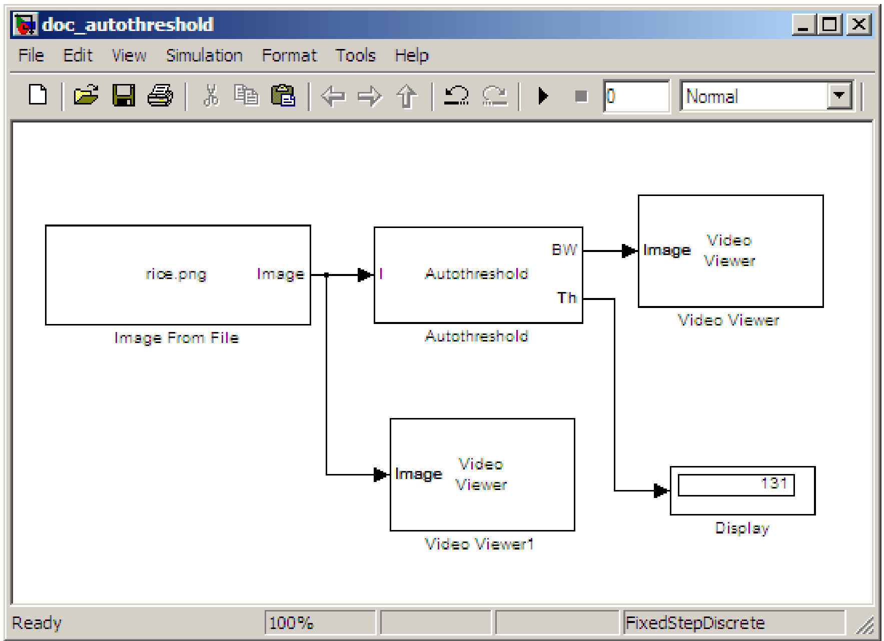
The original intensity image appears in the Video Viewer1 window.



The binary image appears in the Video Viewer window.



In the model window, the Display block shows the threshold value, calculated by the Autothreshold block, that separated the rice grains from the background.



You have used the Autothreshold block to convert an intensity image to a binary image. For more information about this block, see the Autothreshold block reference page in the *Computer Vision System Toolbox Reference*. To open a demo model that uses this block, type `vipstaples` at the MATLAB command prompt.

Convert R'G'B' to Intensity Images

The Color Space Conversion block enables you to convert color information from the R'G'B' color space to the Y'CbCr color space and from the Y'CbCr color space to the R'G'B' color space as specified by Recommendation ITU-R BT.601-5. This block can also be used to convert from the R'G'B' color space to intensity. The prime notation indicates that the signals are gamma corrected.

Some image processing algorithms are customized for intensity images. If you want to use one of these algorithms, you must first convert your image to intensity. In this topic, you learn how to use the Color Space Conversion block to accomplish this task. You can use this procedure to convert any R'G'B' image to an intensity image:

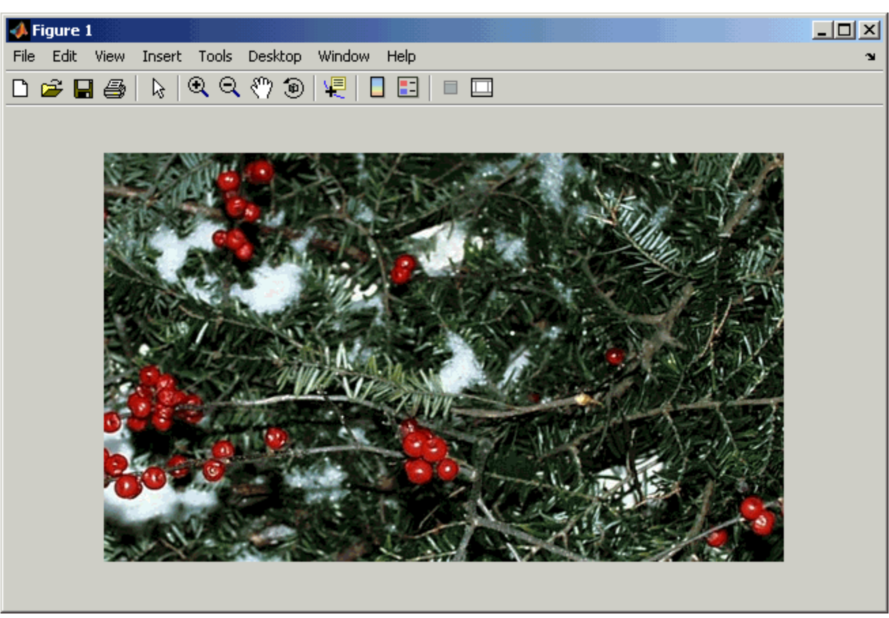
1 Define an R'G'B' image in the MATLAB workspace. To read in an R'G'B' image from a JPG file, at the MATLAB command prompt, type

```
I= imread('greens.jpg');
```

`I` is a 300-by-500-by-3 array of 8-bit unsigned integer values. Each plane of this array represents the red, green, or blue color values of the image.

2 To view the image this matrix represents, at the MATLAB command prompt, type

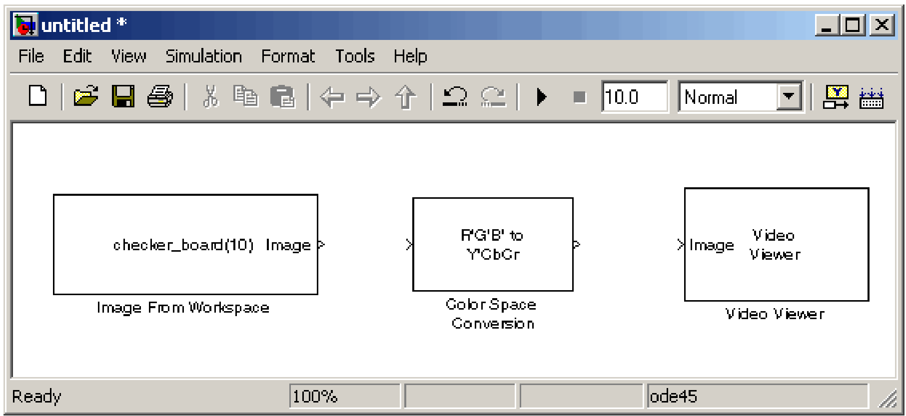
```
imshow(I)
```



3 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From Workspace	Computer Vision System Toolbox > Sources	1
Color Space Conversion	Computer Vision System Toolbox > Conversions	1
Video Viewer	Computer Vision System Toolbox > Sinks	1

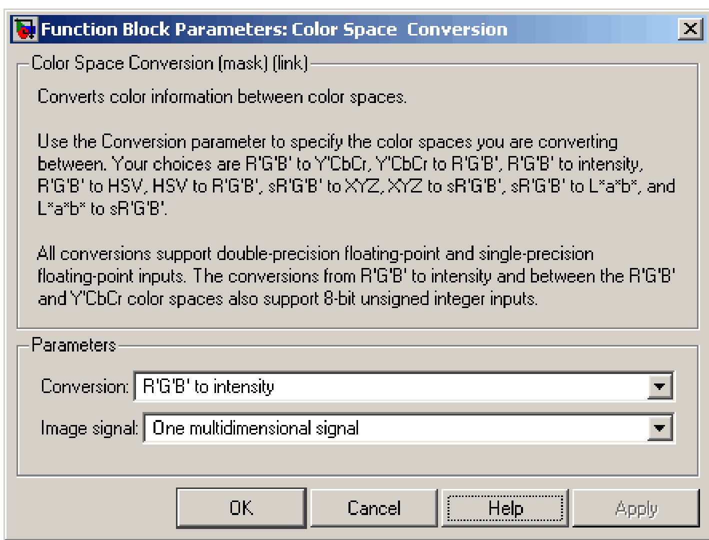
4 Position the blocks as shown in the following figure.



Once you have assembled the blocks needed to convert a R'G'B' image to an intensity image, you are ready to set your block parameters. To do this, double-click the blocks, modify the block parameter values, and click **OK**.

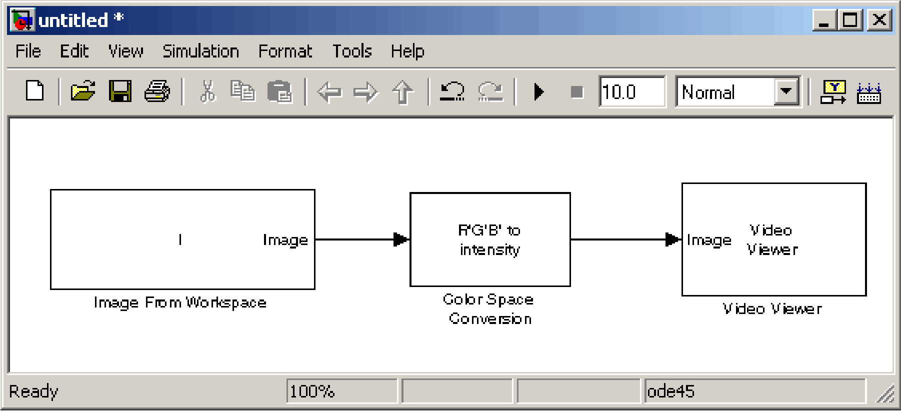
5 Use the Image from Workspace block to import your image from the MATLAB workspace. Set the**Value** parameter to `I`.

6 Use the Color Space Conversion block to convert the input values from the R'G'B' color space to intensity. Set the **Conversion** parameter to `R'G'B' to intensity`.



7 View the modified image using the Video Viewer block. Accept the default parameters.

8 Connect the blocks so that your model is similar to the following figure.

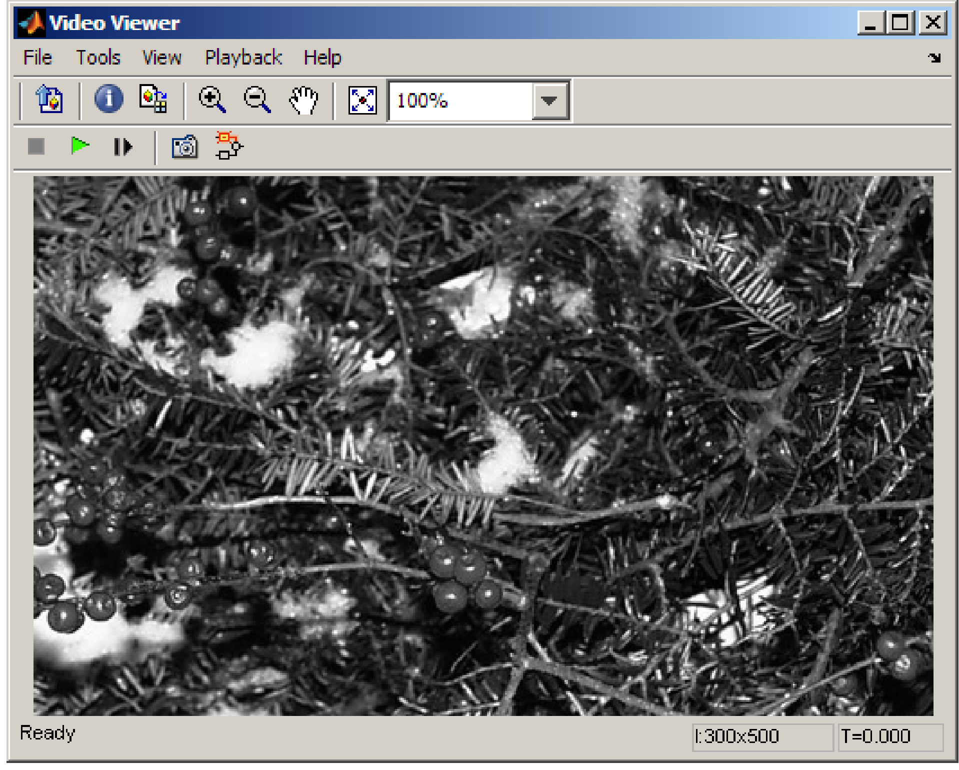


9 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

10 Run your model.

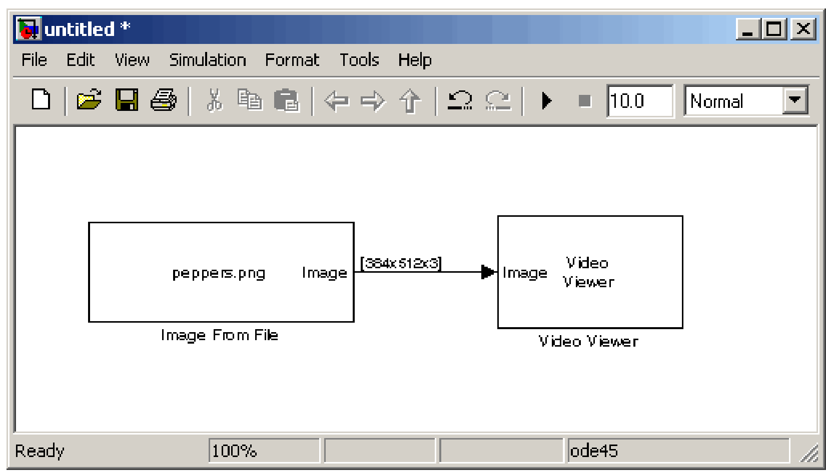
The image displayed in the Video Viewer window is the intensity version of the `greens.jpg` image.

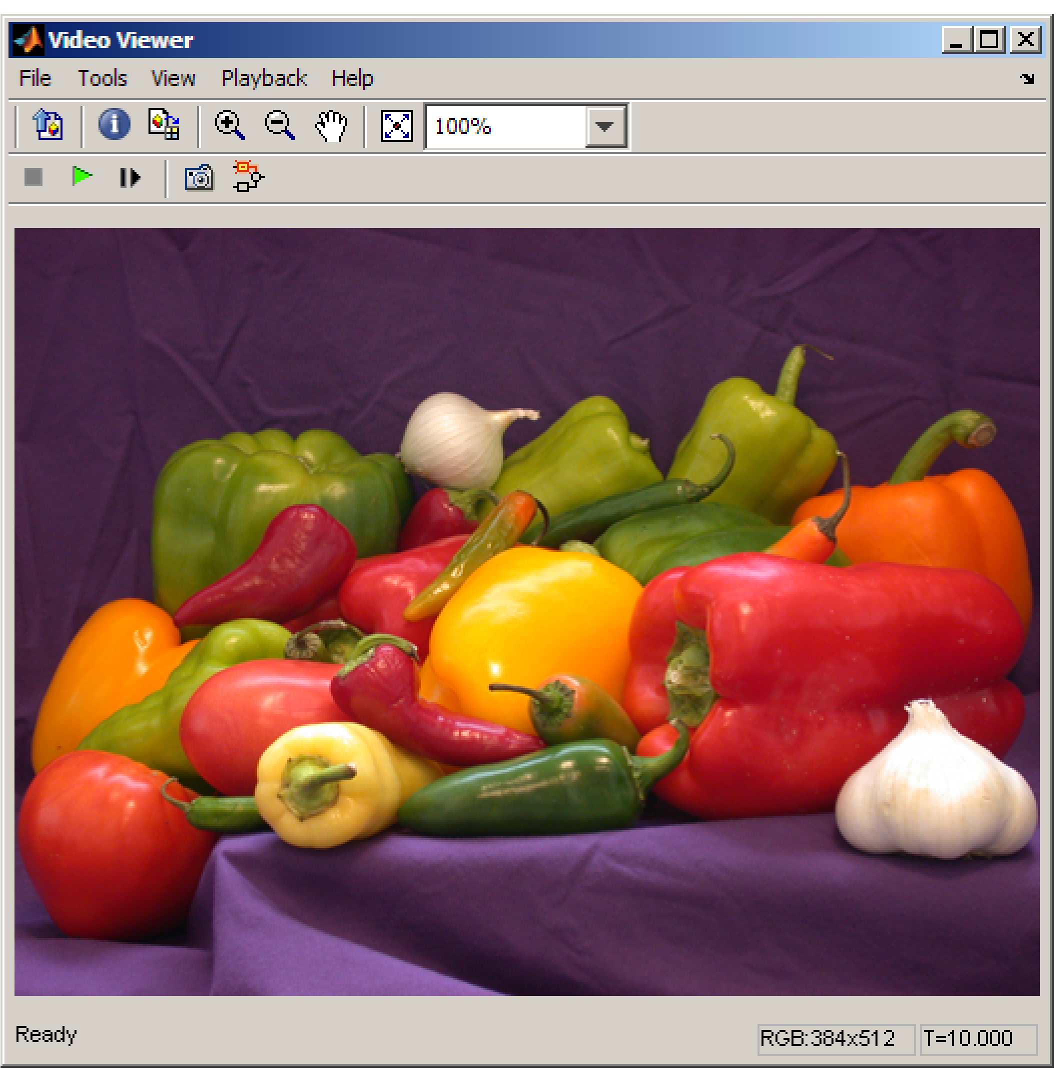


In this topic, you used the Color Space Conversion block to convert color information from the R'G'B' color space to intensity. For more information on this block, see the Color Space Conversion block reference page.

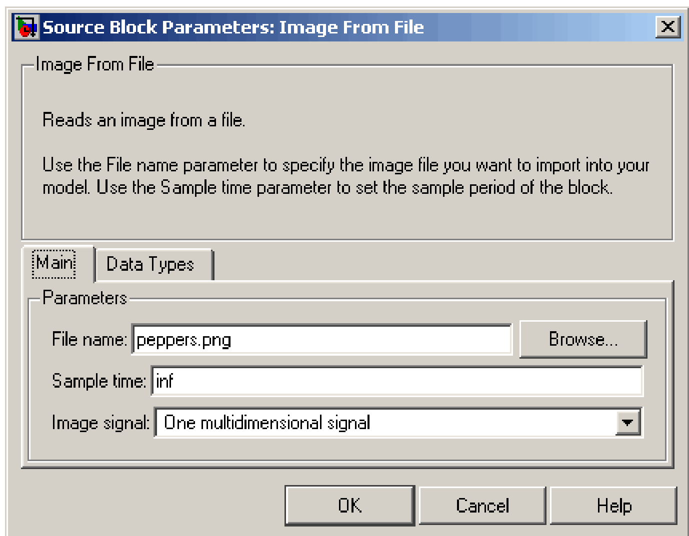
Process Multidimensional Color Video Signals

The Computer Vision System Toolbox software enables you to work with color images and video signals as multidimensional arrays. For example, the following model passes a color image from a source block to a sink block using a 384-by-512-by-3 array.





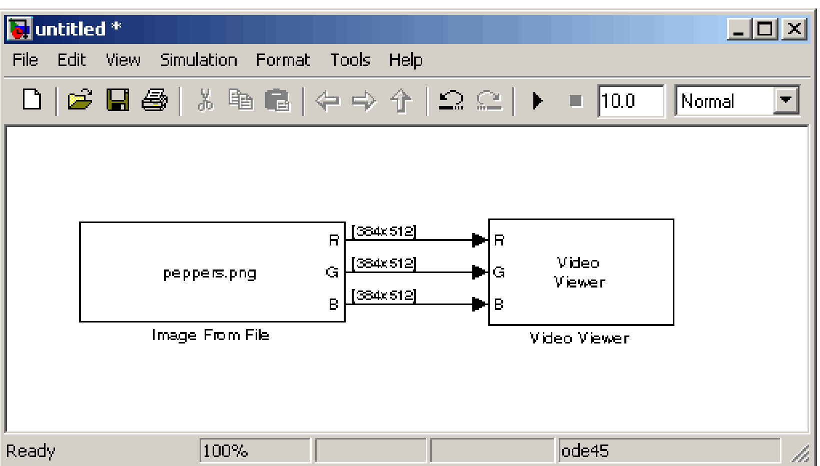
You can choose to process the image as a multidimensional array by setting the **Image signal** parameter to `One multidimensional signal` in the Image From File block dialog box.

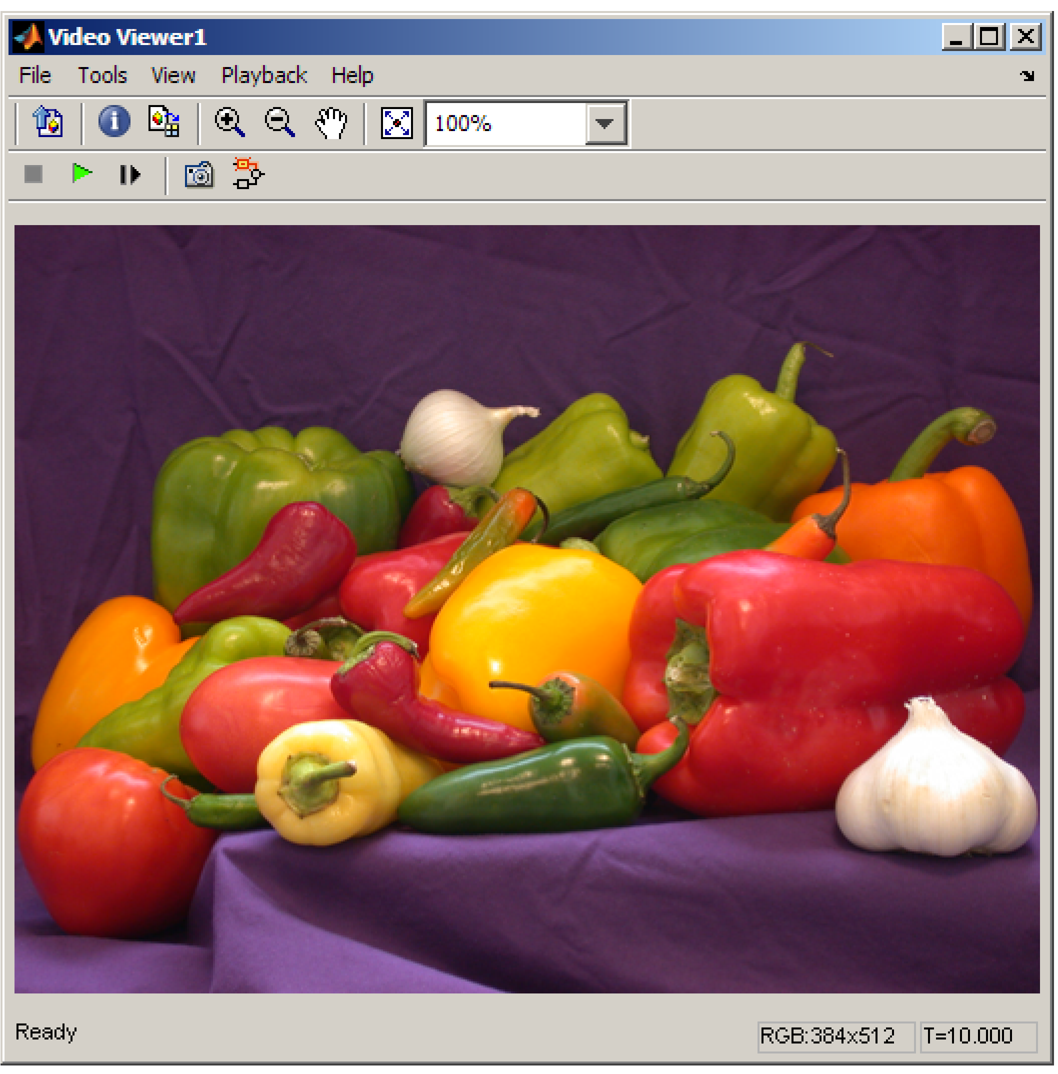


The blocks that support multidimensional arrays meet at least one of the following criteria:

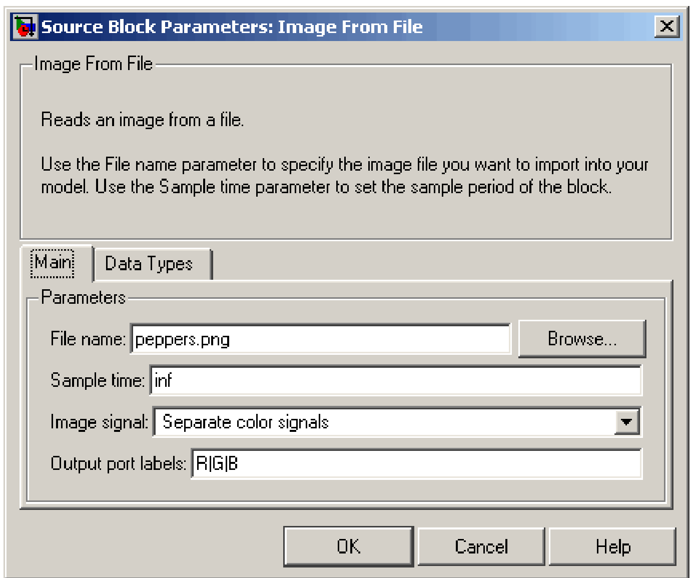
- They have the **Image signal** parameter on their block mask.
- They have a note in their block reference pages that says, “This block supports intensity and color images on its ports.”
- Their input and output ports are labeled “Image”.

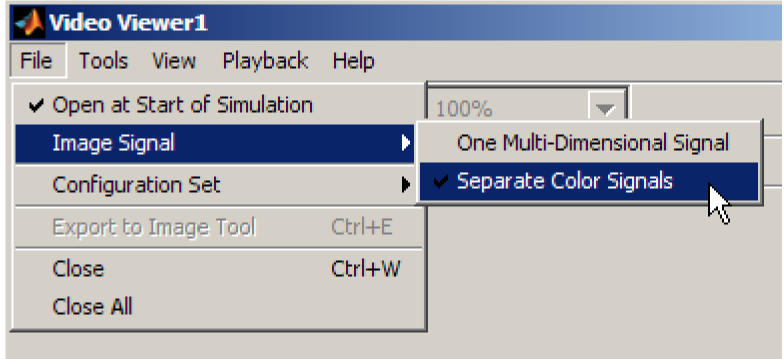
You can also choose to work with the individual color planes of images or video signals. For example, the following model passes a color image from a source block to a sink block using three separate color planes.


untitled *
File Edit View Simulation Format Tools Help
10.0
Normal
R
peppers.png
G
B
[384x512]
[384x512]
[384x512]
R
G
B
Video
Viewer
Image From File
Video Viewer
Ready
100%
ode45



To process the individual color planes of an image or video signal, set the **Image signal** parameter to `Separate color signals` in both the Image From File and Video Viewer block dialog boxes.





Note The ability to output separate color signals is a legacy option. It is recommend that you use multidimensional signals to represent color data.

If you are working with a block that only outputs multidimensional arrays, you can use the Selector block to separate the color planes. For an example of this process, see "Measure an Angle Between Lines" on page 3-14. If you are

working with a block that only accepts multidimensional arrays, you can use the Matrix Concatenation block to create a multidimensional array. For an example of this process, see "Find the Histogram of an Image" on page 7-2.

Data Formats

In this section...
"Video Formats" on page 1-49
"Video Data Stored in Column-Major Format" on page 1-50
"Image Formats" on page 1-50

Video Formats

Video data is a series of images over time. Video in binary or intensity format is a series of single images. Video in RGB format is a series of matrices grouped into sets of three, where each matrix represents an R, G, or B plane.

Defining Intensity and Color

Video data is a series of images over time. Video in binary or intensity format is a series of single images. Video in RGB format is a series of matrices grouped into sets of three, where each matrix represents an R, G, or B plane.

The values in a binary, intensity, or RGB image can be different data types. The data type of the image values determines which values correspond to black and white as well as the absence or saturation of color. The following table summarizes the interpretation of the upper and lower bound of each data type. To view the data types of the signals at each port, from the **Format** menu, point to **Port/Signal Displays**, and select **Port Data Types**.

Data Type	Black or Absence of Color	White or Saturation of Color
Fixed point	Minimum data type value	Maximum data type value
Floating point	0	1

Note The Computer Vision System Toolbox software considers any data type other than double-precision floating point and single-precision floating point to be fixed point.

For example, for an intensity image whose image values are 8-bit unsigned integers, 0 is black and 255 is white. For an intensity image whose image values are double-precision floating point, 0 is black and 1 is white. For an intensity image whose image values are 16-bit signed integers, -32768 is black and 32767 is white.

For an RGB image whose image values are 8-bit unsigned integers, 0 0 0 is black, 255 255 255 is white, 255 0 0 is red, 0 255 0 is green, and 0 0 255 is blue. For an RGB image whose image values are double-precision floating point, 0 0 0 is black, 1 1 1 is white, 1 0 0 is red, 0 1 0 is green, and 0 0 1 is blue. For an RGB image whose image values are 16-bit signed integers, -32768 -32768 -32768 is black, 32767 32767 32767 is white, 32767 -32768 -32768 is red, -32768 32767 -32768 is green, and -32768 -32768 32767 is blue.

Video Data Stored in Column-Major Format

The MATLAB technical computing software and Computer Vision System Toolbox blocks use column-major data organization. The blocks' data buffers store data elements from the first column first, then data elements from the second column second, and so on through the last column.

If you have imported an image or a video stream into the MATLAB workspace using a function from the MATLAB environment or the Image Processing Toolbox, the Computer Vision System Toolbox blocks will display this image or video stream correctly. If you have written your own function or code to import images into the MATLAB environment, you must take the column-major convention into account.

Image Formats

In the Computer Vision System Toolbox software, images are real-valued ordered sets of color or intensity data. The blocks interpret input matrices as images, where each element of the matrix corresponds to a single pixel in the displayed image. Images can be binary, intensity (grayscale), or RGB. This section explains how to represent these types of images.

Binary Images

Binary images are represented by a Boolean matrix of 0s and 1s, which correspond to black and white pixels, respectively.

For more information, see “Binary Images” in the Image Processing Toolbox™ documentation.

Intensity Images

Intensity images are represented by a matrix of intensity values. While intensity images are not stored with colormaps, you can use a gray colormap to display them.

For more information, see “Grayscale Images” in the Image Processing Toolbox documentation.

RGB Images

RGB images are also known as a true-color images. With Computer Vision System Toolbox blocks, these images are represented by an array, where the first plane represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities. In the Computer Vision System Toolbox software, you can pass RGB images between blocks as three separate color planes or as one multidimensional array.

For more information, see “Truecolor Images” in the Image Processing Toolbox documentation.

Display and Graphics

- "Display" on page 2-2
- "Graphics" on page 2-24

Display

In this section...

"View Streaming Video in MATLAB using Video Player and Deployable Video Player System Objects" on page 2-2

"Preview Video in MATLAB using MPlay Function" on page 2-2

"To Video Display Block" on page 2-4

"View Video in Simulink using MPlay Function as a Floating Scope" on page 2-4

"MPlay" on page 2-7

View Streaming Video in MATLAB using Video Player and Deployable Video Player System Objects

Video Player System Object

Use the video player System object when you require a simple video display in MATLAB.

For more information about the video player object, see the `vision.VideoPlayer` reference page.

Deployable Video Player System Object

Use the deployable video player object as a basic display viewer designed for optimal performance. This block supports code generation for the Windows platform.

For more information about the Deployable Video Player block, see the `vision.DeployableVideoPlayer` object reference page.

Preview Video in MATLAB using MPlay Function

The MPlay function enables you to view videos represented as variables in the MATLAB workspace.

You can open several instances of the MPlay function simultaneously to view multiple video data sources at once. You can also dock these MPlay GUIs in the MATLAB desktop. Use the figure arrangement buttons in the upper-right corner of the Sinks window to control the placement of the docked GUIs.

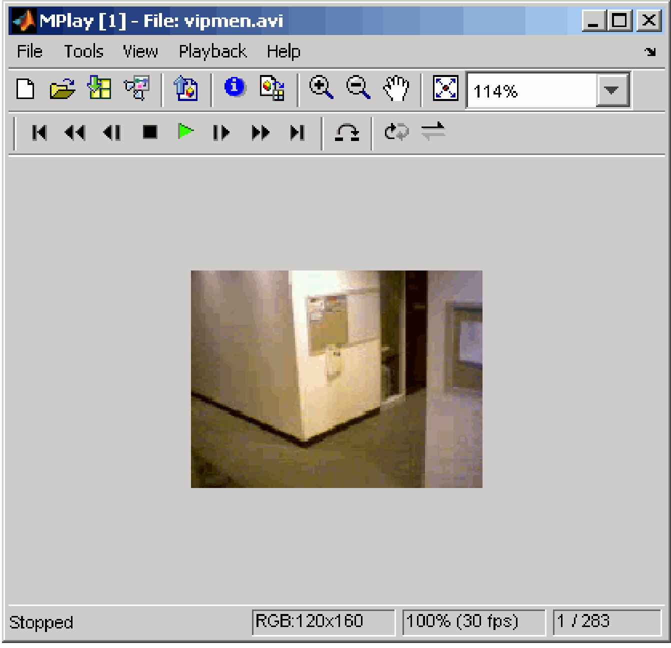
The MPlay GUI enables you to view videos directly from files without having to load all the video data into memory at once. The following procedure shows you how to use the MPlay GUI to load and view a video one frame at a time:

1 On the MPlay GUI, click *open file* element,

2 Use the Connect to File dialog box to navigate to the multimedia file you want to view in the MPlay window.

   For example, navigate to `$matlabroot\toolbox\vision\visiondemos\vipmen.avi`.

   Click **Open**. The first frame of the video appears in the MPlay window.



Note The MPlay GUI supports AVI files that the `VideoReader` supports.

3 Experiment with the MPlay GUI by using it to play and interact with the video stream.

View Video in Simulink using the Video Viewer and To Video Display Blocks

Video Viewer Block

Use the Video Viewer block when you require a wired-in video display with simulation controls in your Simulink model. The Video Viewer block provides simulation control buttons directly from the GUI. The block integrates play, pause, and step features while running the model and also provides video analysis tools such as pixel region viewer.

For more information about the Video Viewer block, see the Video Viewer block reference page.

To Video Display Block

Use the To Video Display block in your Simulink model as a simple display viewer designed for optimal performance. This block supports code generation for the Windows platform.

For more information about the To Video Display block, see the To Video Display block reference page.

View Video in Simulink using MPlay Function as a Floating Scope

The MPlay GUI enables you to view video signals in Simulink models without adding blocks to your model.

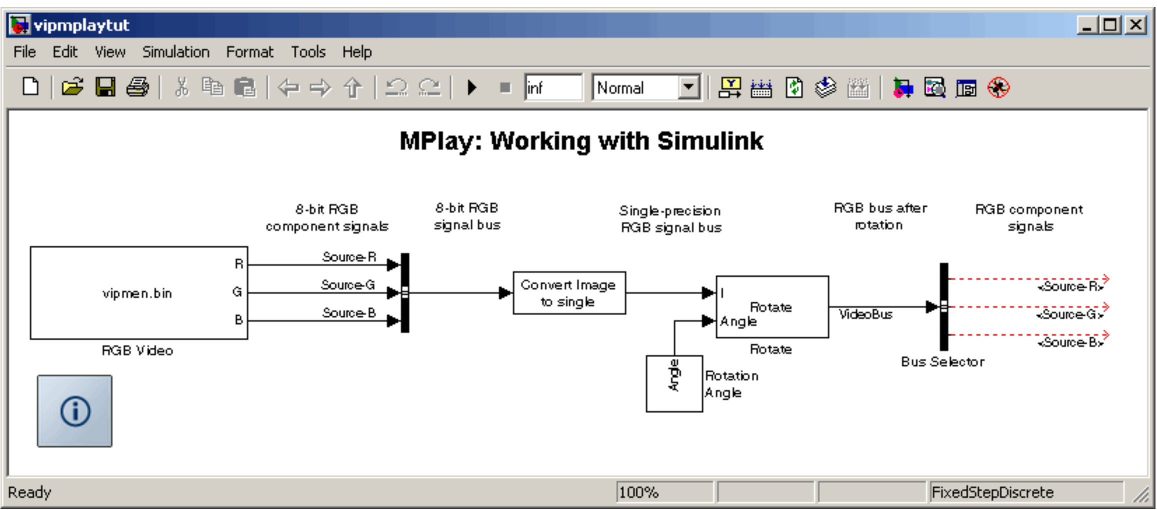
You can open several instances of the MPlay GUI simultaneously to view multiple video data sources at once. You can also dock these MPlay GUIs in the MATLAB desktop. Use the figure arrangement buttons in the upper-right corner of the Sinks window to control the placement of the docked GUIs.

Set Simulink simulation mode to `Normal` to use `mplay` . MPlay does not work when you use "Accelerating Simulink Models" on page 9-9.

The following procedure shows you how to use MPlay to view a Simulink signal:

1 Open a Simulink model. At the MATLAB command prompt, type

```
vipmplaytut
```



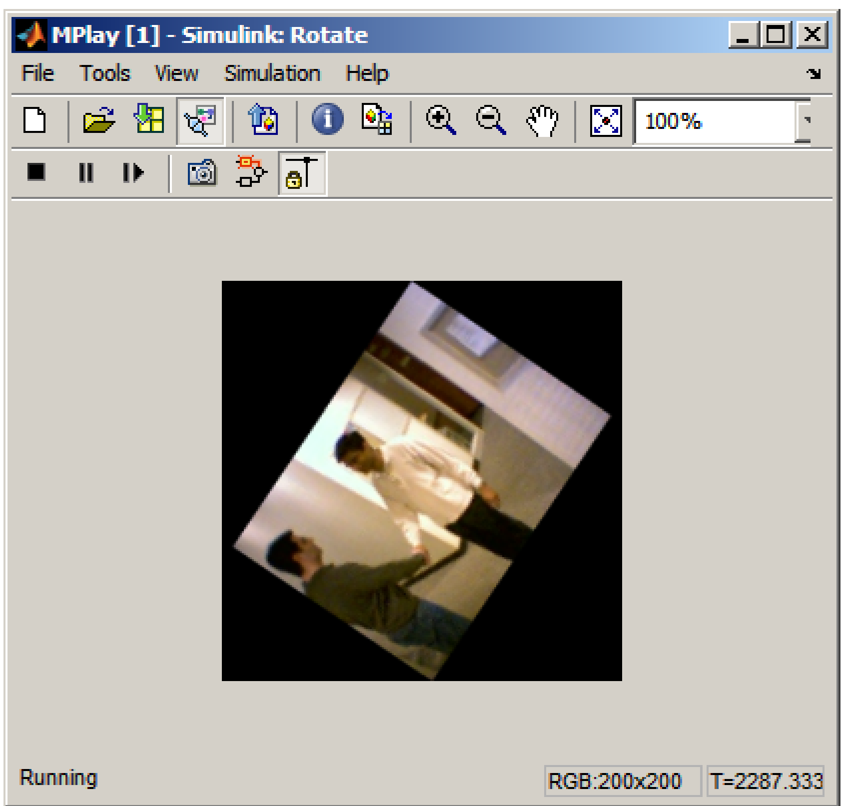
2 Open an MPlay GUI by typing `mplay` on the MATLAB command line.

3 Run the model.

4 Select the signal line you want to view. For example, select the bus signal coming out of the Rotate block.

5 On the MPlay GUI, click *Connect to Simulink Signal* GUI element,

The video appears in the MPlay window.



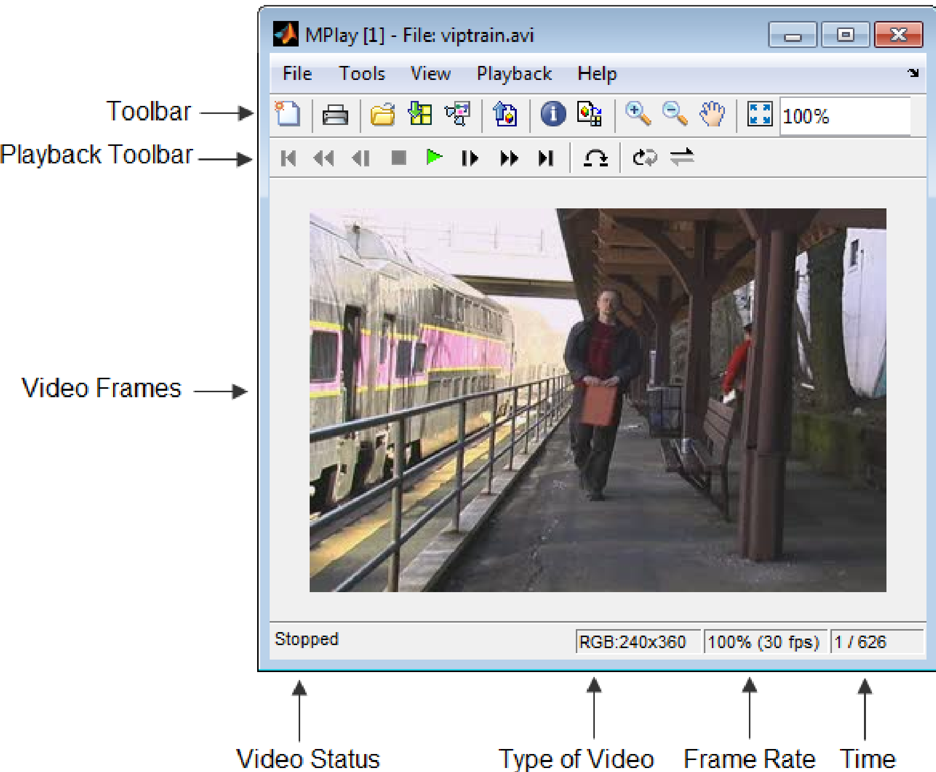
6 Change to floating-scope mode by clicking the *persistent connect* GUI element, button.

7 Experiment with selecting different signals and viewing them in the MPlay window. You can also use multiple MPlay GUIs to display different Simulink signals.

Note During code generation, the Simulink Coder does not generate code for the MPlay GUI.

MPlay

The following figure shows the MPlay GUI containing an image sequence.



The following sections provide descriptions of the MPlay GUI toolbar buttons and equivalent menu options.

Toolbar Buttons

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	File > New MPlay	**Ctrl+N**	Open a new MPlay GUI.
	File > Print	**Ctrl+P**	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 by pausing or stopping model simulation, or simulating the model in step-forward mode. To print the current window to a figure rather than sending it to your printer, select **File > Print to figure**.
	File > Print	**Ctrl+P**	Print the current scope window. Printing is only available when the scope display is not changing. You can enable printing by placing the scope in snapshot mode, or by pausing or stopping model simulation. To print the current scope window to a figure rather than sending it to your printer, select **File > Print to figure**.
	File > Open	**Ctrl+O**	Connect to a video file.

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	File > Import from Workspace	**Ctrl+I**	Connect to a variable from the base MATLAB workspace.
	File > Connect to Simulink Signal		Connect to a Simulink signal.
	File > Export to Image Tool	**Ctrl+E**	Send the current video frame to the Image Tool. For more information, see "Using the Image Tool to Explore Images" in the Image Processing Toolbox documentation. The Image Tool only knows the frame is an intensity image if the colormap of the frame is grayscale (`gray(256)`). Otherwise, the Image Tool assumes that the frame is an indexed image and disables the **Adjust Contrast** button.
	Tools > Video Information	**V**	View information about the video data source.
	Tools > Pixel Region	N/A	Open the Pixel Region tool. For more information about this tool, see the Image Processing Toolbox documentation.
	Tools > Zoom In	N/A	Zoom in on the video display.

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	Tools > Zoom Out	N/A	Zoom out of the video display.
	Tools > Pan	N/A	Move the image displayed in the GUI.
	Tools > Maintain Fit to Window	N/A	Scale video to fit GUI size automatically. Toggle the button on or off.
100%	N/A	N/A	Enlarge or shrink the video display. This option is available if you do not select the **Maintain Fit to Window** button.

Playback Toolbar — Workspace and File Sources

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	Playback > Go to First	**F**, **Home**	Go to the first frame of the video.
	Playback > Rewind	Up arrow	Jump back ten frames.
	Playback > Step Back	Left arrow, **Page Up**	Step back one frame.
	Playback > Stop	**S**	Stop the video.
	Playback > Play	**P**, Space bar	Play the video.
	Playback > Pause	**P**, Space bar	Pause the video. This button appears only when the video is playing.

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	Playback > Step Forward	Right arrow, **Page Down**	Step forward one frame.
	Playback > Fast Forward	Down arrow	Jump forward ten frames.
	Playback > Go to Last	**L**, **End**	Go to the last frame of the video.
	Playback > Jump to	**J**	Jump to a specific frame.
	Playback > Playback Modes > Repeat	**R**	Repeated video playback.
	Playback > Playback Modes > Forward play	**A**	Play the video forward.
	Playback > Playback Modes > Backwardplay	**A**	Play the video backward.
	Playback > Playback Modes > AutoReverse play	**A**	Play the video forward and backward.

Playback Toolbar — Simulink Sources

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	Simulation > Stop	S	Stop the video. This button also controls the Simulink model.
	Simulation > Start	**P**, Space bar	Play the video. This button also controls the Simulink model.
	Simulation > Pause	**P**, Space bar	Pause the video. This button also controls the Simulink model and appears only when the video is playing.
	Simulation > Step Forward	Right arrow, Page Down	Step forward one frame. This button also controls the Simulink model.
	Simulation > Simulink Snapshot	N/A	Click this button to freeze the display in the MPlay window.
	View > Highlight Simulink Signal	**Ctrl+L**	In the model window, highlight the Simulink signal the MPlay GUI is displaying.
	Simulation > Floating Signal Connection (not selected)	N/A	Indicates persistent Simulink connection. In this mode, the MPlay GUI always associates with the Simulink signal you selected before you clicked the **Connect to Simulink Signal** button.
	Simulation > Floating Signal	N/A	Indicates floating Simulink connection. In this mode, you can click different signals in

GUI	Menu Equivalent	Shortcut Keys and Accelerators	Description
	Connection (selected)		the model, and the MPlay GUI displays them. You can use only one MPlay GUI in floating-scope mode at a time.

Configuration

The MPlay Configuration dialog box enables you to change the behavior and appearance of the GUI as well as the behavior of the playback shortcut keys.

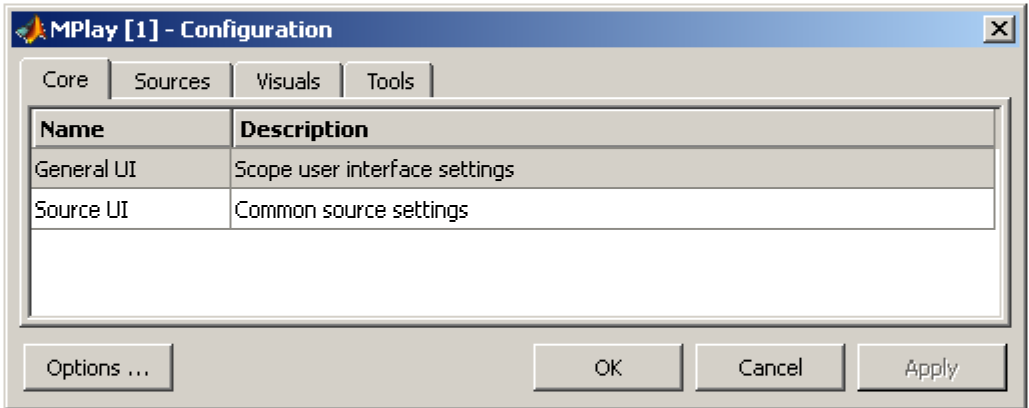
- To open the Configuration dialog box, select **File > Configuration Set > Edit**.
- To save the configuration settings for future use, select **File > Configuration Set > Save as**.

Note By default, the MPlay GUI uses the configuration settings from the file `mplay.cfg`. Create a backup copy of the file to store your configuration settings.

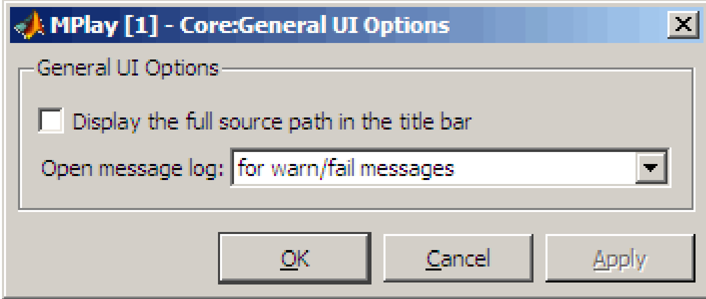
- To load a preexisting configuration set, select **File > Configuration Set > Load**.

Configuration Core Pane

The Core pane controls the graphic user interface (GUI) general and source settings.



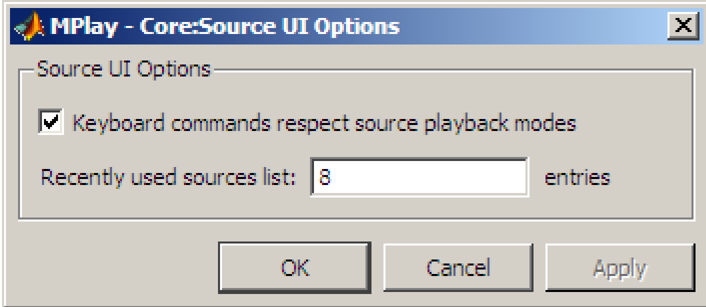
General UI
Click **General UI**, and then select the **Options** button to open the General UI Options dialog box.



If you select the **Display the full source path in the title bar** check box, the full Simulink path appears in the title bar. Otherwise, the title bar displays a shortened name.

Use the **Message log opens** 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`.

Source UI
Click Source UI, and then click the **Options** button to open the Source UI Options dialog box.



If you select the **Keyboard commands respect playback modes** check box, the keyboard shortcut keys behave in response to the playback mode you selected.

Using the Keyboard commands respect playback modes

Open and play a video using MPlay.

1 Select the **Keyboard commands respect playback modes** check box.

2 Select the **Backward playback** button.

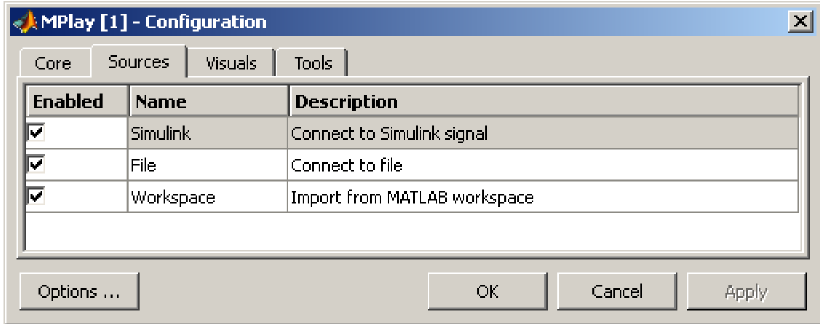
- Using the right keyboard arrow key moves the video backward, and using the left keyboard arrow key moves the video forward.
- With MPlay set to play backwards, the keyboard “forward” performs “forward with the direction the video is playing”.

To disconnect the keyboard behavior from the MPlay playback settings, clear the check box.

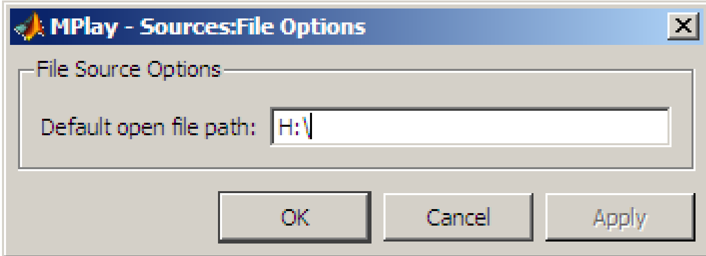
Use the **Recently used sources list** parameter to control the number of sources you see in the **File** menu.

Configuration Sources Pane

The Sources pane contains the GUI options that relate to connecting to different sources. Select the **Enabled** check box next to each source type to specify to which type of source you want to connect the GUI.

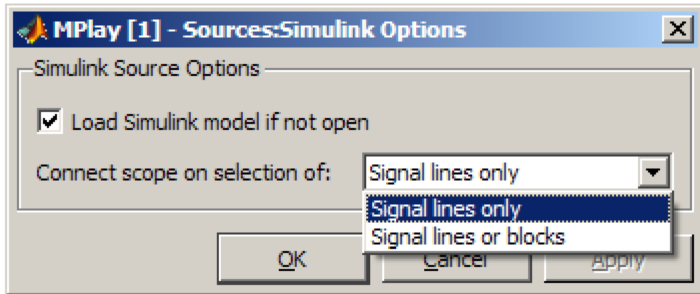


- Click **File**, and then click the **Options** button to open the **Sources:File Options** dialog box.



  Use the **Default open file path** parameter to control the folder that is displayed in the **Connect to File** dialog box. The **Connect to File** dialog box becomes available when you select **File > Open**.

- Click **Simulink**, and then click the **Options** button to open the **Sources:Simulink Options** dialog box.

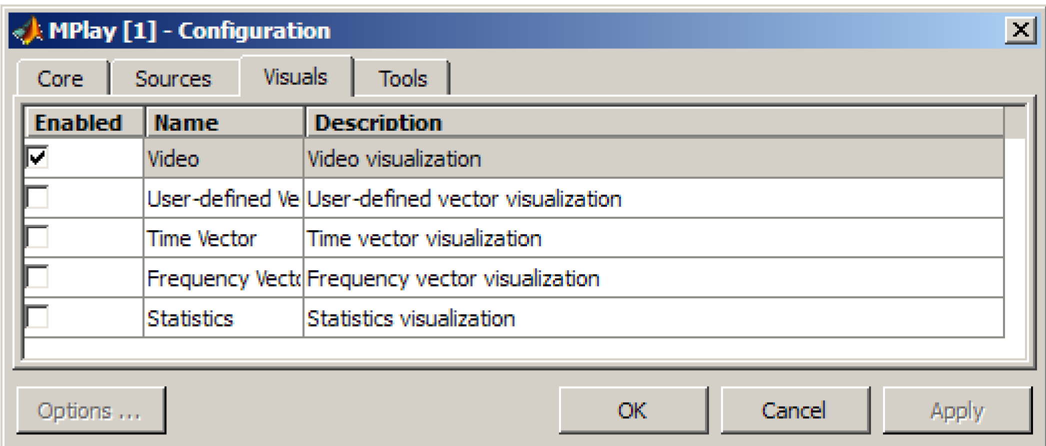


You can have the Simulink model associated with an MPlay GUI to open with MPlay. To do so, select the **Load Simulink model if not open** check box.

Select **Signal lines only** to sync the video display only when you select a signal line. If you select a block the video display will not be affected. Select **Signal lines or blocks** to sync the video display to the signal line or block you select. The default is **Signal lines only**.

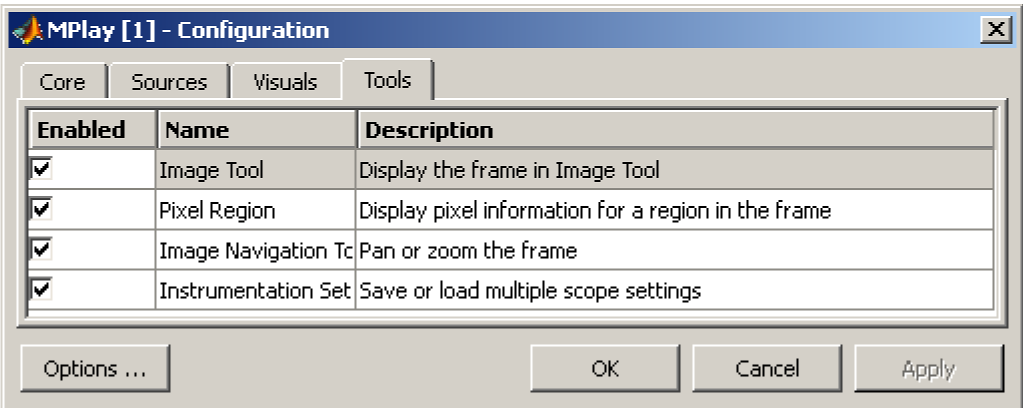
Configuration Visuals Pane

The Visuals pane contains the name of the visual type and its description.

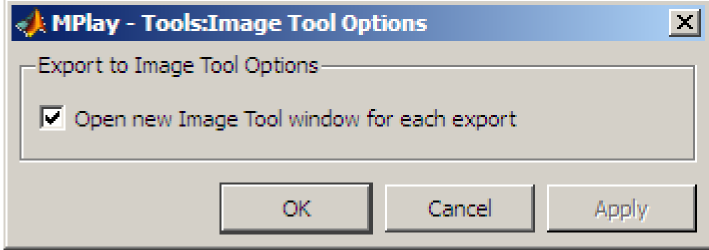


Configuration Tools Pane

The Tools pane contains the tools that are available on the MPlay GUI. Select the **Enabled** check box next to the tool name to specify which tools to include on the GUI.



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

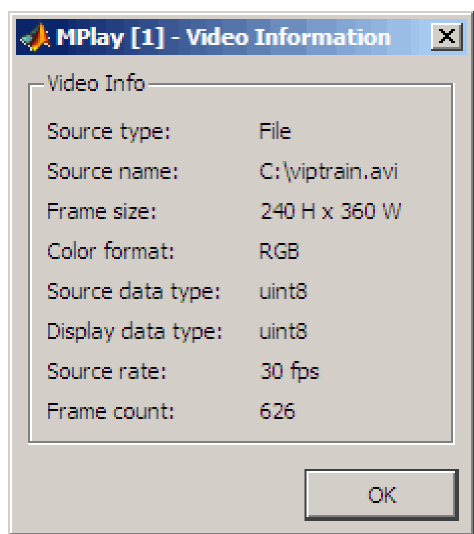
Select the **Pixel Region** check box to display and enable the pixel region GUI button. For more information on working with pixel regions, see Getting Information about the Pixels in an Image.

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.

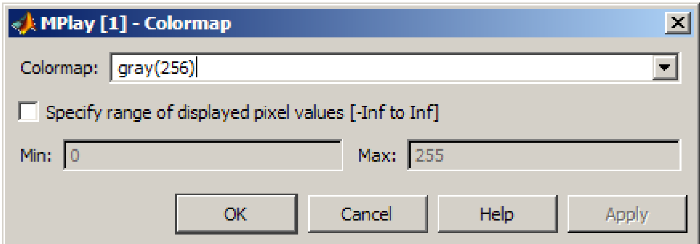
Video Information

The Video Information dialog box lets you view basic information about the video. To open this dialog box, select **Tools > Video Information** or click the information button .



Color Map 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.

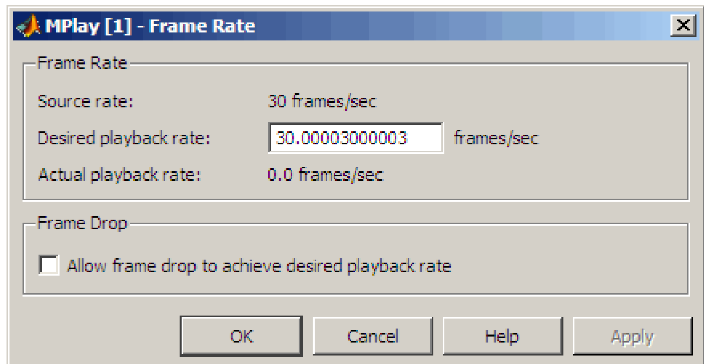
Sometimes, the pixel values do not use the entire data type range. In such cases, you can select the **Specify range of displayed pixel values** check box. You can then enter the range for your data. The dialog box automatically displays the range based on the data type of the pixel values.

Frame Rate

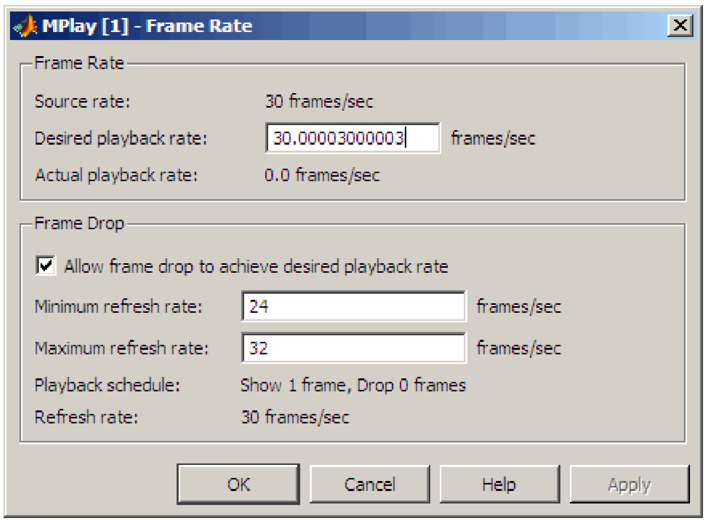
The Frame Rate dialog box displays the frame rate of the source. It also lets you change the rate at which the MPlay GUI plays the video and displays the actual playback rate.

Note This dialog box becomes available when you use the MPlay GUI to view a video signal.

The *playback rate* is the number of frames the GUI processes per second. You can use the **Desired playback rate** parameter to decrease or increase the playback rate. To open this dialog box, select **Playback > Frame Rate** or press **T**.



To increase the playback rate when system hardware cannot keep pace with the desired rate, select the **Allow frame drop to achieve desired playback rate** check box. This parameter enables the MPlay GUI to achieve the playback rate by dropping video frames. Dropped video frames sometimes cause lower quality playback.



You can refine further the quality of playback versus hardware burden, by controlling the number of frames to drop per frame or frames displayed. For example, suppose you set the **Desired playback rate** to 80 frames/sec. One way to achieve the desired playback rate is to set the **Playback schedule**

to `Show 1 frame, Drop 3 frames`. Change this playback schedule, by setting the refresh rates (which is how often the GUI updates the screen), to:

Maximum refresh rate: 21 frames/sec
Minimum refresh rate: 20 frames/sec

MPlay can achieve the desired playback rate (in this case, 80 frames/sec) by using these parameter settings.

In general, the relationship between the **Frame Drop** parameters is:

$$Desired_rate = refresh_rate * \frac{show_frames + drop_frames}{show_frames}$$

In this case, the *refresh_rate* includes a more accurate calculation based on both the minimum and maximum refresh rates.

Use the **Minimum refresh rate** and **Maximum refresh rate** parameters to adjust the playback schedule of video display. Use these parameters in the following way:

- Increase the **Minimum refresh rate** parameter to achieve smoother playback.
- Decrease the **Maximum refresh rate** parameter to reduce the demand on system hardware.

Saving the Settings of Multiple MPlay GUIs

The MPlay GUI enables you to save and load the settings of multiple GUI instances. You only have to configure the MPlay GUIs associated with your model once.

To save the GUI settings:

- Select **File > Instrumentation Sets > Save Set**

To open the preconfigured MPlay GUIs:

- Select **File > Instrumentation Sets > Load Set**

You can save instrument sets for instances of MPlay connected to a source. If you attempt to save an instrument set for an MPlay instance that is not connected to a source, the Message Log displays a warning.

Message Log

The Message Log dialog box 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` message indicates a new configuration file loaded.
- The `Extension` message indicates a registered component. For example, a `Simulink` message, indicating a registered component, available for configuration.

Status Bar

Along the bottom of the MPlay viewer is the status bar. It displays information, such as video status, Type of video playing (`I` or `RGB`), Frame size, Percentage of frame rate, Frame rate, and `Current frame:  Total frames`.

Note A minus sign `(-)` for `Current frame` indicates reverse video playback.

Graphics

In this section...

"Abandoned Object Detection" on page 2-24

"Annotate Video Files with Frame Numbers" on page 2-30

Abandoned Object Detection

This example tracks objects at a train station and determines which ones remain stationary. Abandoned objects in public areas concern authorities since they might pose a security risk. Algorithms, such as the one used in this example, can be used to assist security officers monitoring live surveillance video by directing their attention to a potential area of interest.

This example illustrates how to use the BlobAnalysis System object to identify objects and track them. The example implements this algorithm using the following steps:

- Extract a region of interest (ROI), thus eliminating video areas that are unlikely to contain abandoned objects.
- Perform video segmentation using background subtraction.
- Calculate object statistics using the blob analysis System object.
- Track objects based on their area and centroid statistics.
- Visualize the results.

Initialize Required Variables and System Objects

Use these next sections of code to initialize the required variables and System objects.

Rectangular ROI [x y width height], where [x y] is the uppef left corner of the ROI

```
roi = [100 80 360 240];
% Maximum number of objects to track
maxNumObj = 200;
```

```
% Number of frames that an object must remain stationary before an alarm is
% raised
alarmCount = 45;
% Maximum number of frames that an abandoned object can be hidden before it
% is no longer tracked
maxConsecutiveMiss = 4;
areaChangeFraction = 13;     % Maximum allowable change in object area in p
centroidChangeFraction = 18; % Maximum allowable change in object centroid
% Minimum ratio between the number of frames in which an object is detected
% and the total number of frames, for that object to be tracked.
minPersistenceRatio = 0.7;
% Offsets for drawing bounding boxes in original input video
PtsOffset = int32(repmat([roi(1), roi(2), 0, 0],[maxNumObj 1]));
```

Create a VideoFileReader System object to read video from a file.

```
hVideoSrc = vision.VideoFileReader;
hVideoSrc.Filename = 'viptrain.avi';
hVideoSrc.VideoOutputDataType = 'single';
```

Create a ColorSpaceConverter System object to convert the RGB image to Y'CbCr format.

```
hColorConv = vision.ColorSpaceConverter('Conversion', 'RGB to YCbCr');
```

Create an Autothresholder System object to convert an intensity image to a binary image.

```
hAutothreshold = vision.Autothresholder('ThresholdScaleFactor', 1.3);
```

Create a MorphologicalClose System object to fill in small gaps in the detected objects.

```
hClosing = vision.MorphologicalClose('Neighborhood', strel('square',5));
```

Create a BlobAnalysis System object to find the area, centroid, and bounding box of the objects in the video.

```
hBlob = vision.BlobAnalysis('MaximumCount', maxNumObj, 'ExcludeBorderBlobs'
hBlob.MinimumBlobArea = 100;
hBlob.MaximumBlobArea = 2500;
```

Create a ShapeInserter System object to draw rectangles around the abandoned objects.

```
hDrawRectangles1 = vision.ShapeInserter('Fill',true, 'FillColor', 'Custom',
    'CustomFillColor', [1 0 0], 'Opacity', 0.5);
```

Create a TextInserter System object to display the number of objects in the video.

```
hDisplayCount = vision.TextInserter('Text', '%4d', 'Color', [1 1 1]);
```

Create a ShapeInserter System object to draw rectangles around all the detected objects in the video.

```
hDrawRectangles2 = vision.ShapeInserter('BorderColor', 'Custom', ...
    'CustomBorderColor', [0 1 0]);
```

Create a ShapeInserter System object to draw a rectangle around the region of interest.

```
hDrawBBox = vision.ShapeInserter('BorderColor', 'Custom', ...
    'CustomBorderColor', [1 1 0]);
```

Create a ShapeInserter System object to draw rectangles around all the identified objects in the segmented video.

```
hDrawRectangles3 = vision.ShapeInserter('BorderColor', 'Custom', ...
    'CustomBorderColor', [0 1 0]);
```

Create System objects to display results.

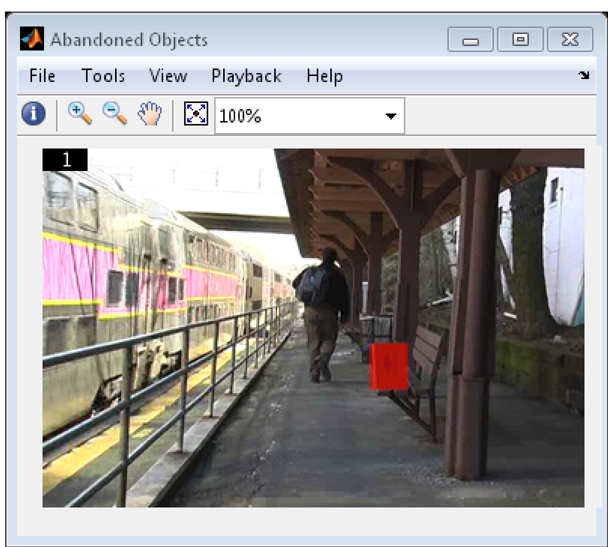
```
pos = [10 300 roi(3)+25 roi(4)+25];
hAbandonedObjects = vision.VideoPlayer('Name', 'Abandoned Objects', 'Positi
pos(1) = 46+roi(3); % move the next viewer to the right
hAllObjects = vision.VideoPlayer('Name', 'All Objects', 'Position', pos);
pos = [80+2*roi(3) 300 roi(3)-roi(1)+25 roi(4)-roi(2)+25];
hThresholdDisplay = vision.VideoPlayer('Name', 'Threshold', 'Position', pos
```

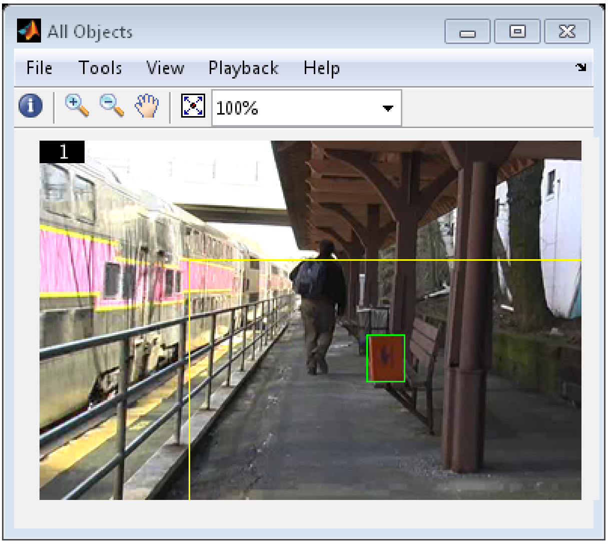
Video Processing Loop

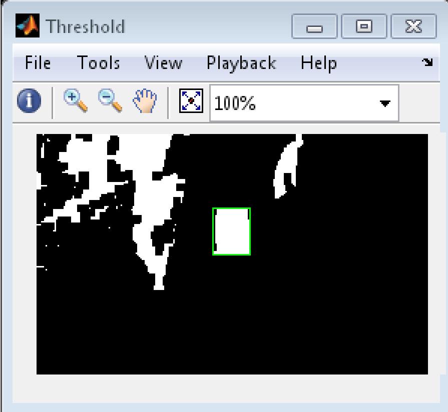
Create a processing loop to perform abandoned object detection on the input video. This loop uses the System objects you instantiated above.

```
firsttime = true;
while ~isDone(hVideoSrc)
    Im = step(hVideoSrc);

    % Select the region of interest from the original video
    OutIm = Im(roi(2):end, roi(1):end, :);

    YCbCr = step(hColorConv, OutIm);
    CbCr  = complex(YCbCr(:,:,2), YCbCr(:,:,3));

    % Store the first video frame as the background
    if firsttime
        firsttime = false;
        BkgY      = YCbCr(:,:,1);
        BkgCbCr   = CbCr;
    end
    SegY    = step(hAutothreshold, abs(YCbCr(:,:,1)-BkgY));
    SegCbCr = abs(CbCr-BkgCbCr) > 0.05;

    % Fill in small gaps in the detected objects
    Segmented = step(hClosing, SegY | SegCbCr);

    % Perform blob analysis
    [Area, Centroid, BBox] = step(hBlob, Segmented);

    % Call the helper function that tracks the identified objects and
    % returns the bounding boxes and the number of the abandoned objects.
    [OutCount, OutBBox] = videoobjtracker(Area, Centroid, BBox, maxNumObj, ...
       areaChangeFraction, centroidChangeFraction, maxConsecutiveMiss, ...
       minPersistenceRatio, alarmCount);

    % Display the abandoned object detection results
    Imr = step(hDrawRectangles1, Im, OutBBox+PtsOffset);
    Imr(1:15,1:30,:) = 0;
    Imr = step(hDisplayCount, Imr, OutCount);
    step(hAbandonedObjects, Imr);
```

```
        BlobCount = size(BBox,1);

        BBoxOffset = BBox + int32(repmat([roi(1) roi(2) 0  0],[BlobCount 1]));
        Imr = step(hDrawRectangles2, Im, BBoxOffset);

        % Display all the detected objects
        Imr(1:15,1:30,:) = 0;
        Imr = step(hDisplayCount, Imr, OutCount);
        Imr = step(hDrawBBox, Imr, roi);
        step(hAllObjects, Imr);

        % Display the segmented video
        SegBBox = PtsOffset;
        SegBBox(1:BlobCount,:) = BBox;
        SegIm = step(hDrawRectangles3, repmat(Segmented,[1 1 3]), SegBBox);
        step(hThresholdDisplay, SegIm);
end

release(hVideoSrc);
```







The `Abandoned Objects` window highlights the abandoned objects with a red box. The `All Objects` window marks the region of interest (ROI) with a yellow box and all detected objects with green boxes. The `Threshold` window shows the result of the background subtraction in the ROI.

Annotate Video Files with Frame Numbers

You can use the `vision.TextInserter` System object in MATLAB, or theInsert Text block in a Simulink model, to overlay text on video streams. In this Simulink model example, you add a running count of the number of video frames to a video using the Insert Text block. The model contains the From Multimedia File block to import the video into the Simulink model, a Frame Counter block to count the number of frames in the input video, and two Video Viewer blocks to view the original and annotated videos.

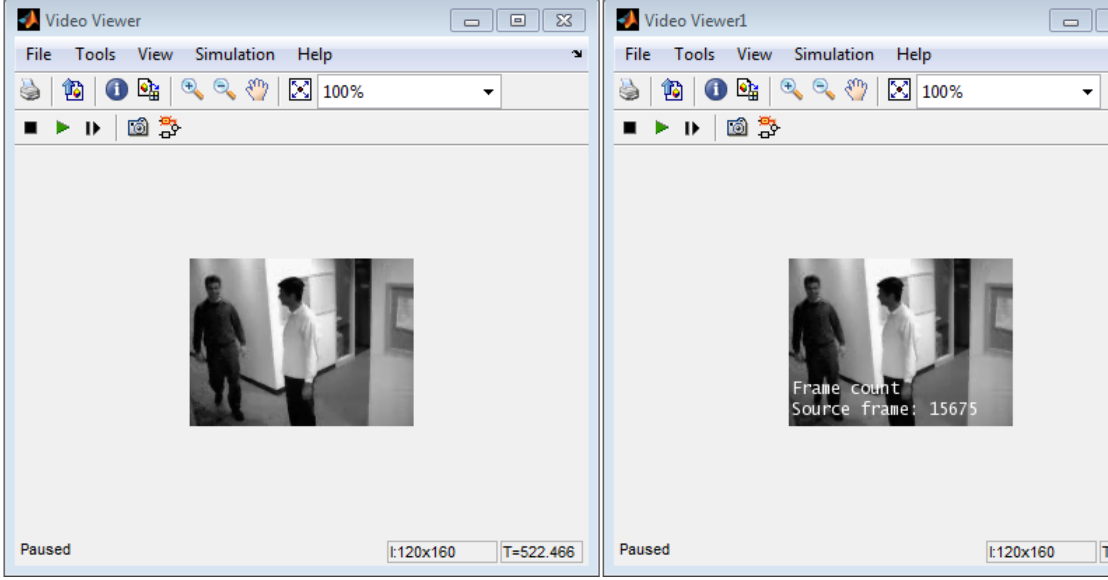
You can open the example model by typing

```
ex_vision_annotate_video_files_with_frame_numbers
```

on the MATLAB command line.

1 Run your model.

2 The model displays the original and annotated videos.



Color Formatting

For this example, the color format for the video was set to `Intensity`, and therefore the color value for the text was set to a scaled value. If instead, you set the color format to `RGB`, then the text value must satisfy this format, and requires a 3-element vector.

Inserting Text

Use the Insert Text block to annotate the video stream with a running frame count. Set the block parameters as follows:

- **Main** pane, **Text** = `['Frame count' sprintf('\n') 'Source frame: %d']`
- **Main** pane, **Color value** = `1`
- **Main** pane, **Location [x y]** = `[2 85]`
- **Font** pane, **Font face** = `LucindaTypewriterRegular`

By setting the **Text** parameter to `['Frame count' sprintf('\n') 'Source frame:  %d']`, you are asking the block to print `Frame count` on one line and the `Source frame:` on a new line. Because you specified `%d`, an ANSI C printf-style format specification, the Variables port appears on the block. The block takes the port input in decimal form and substitutes this input for the `%d` in the string. You used the **Location [x y]** parameter to specify where to print the text. In this case, the location is 85 rows down and 2 columns over from the top-left corner of the image.

Configuration Parameters

Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `inf`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

Registration and Stereo Vision

- "Feature Detection, Extraction, and Matching" on page 3-2
- "Image Registration" on page 3-26
- "Stereo Vision" on page 3-29

Feature Detection, Extraction, and Matching

In this section...

"Detect Edges in Images" on page 3-2

"Detect Lines in Images" on page 3-8

"Detect Corner Features in an Image " on page 3-11

"Find Possible Point Matches Between Two Images" on page 3-12

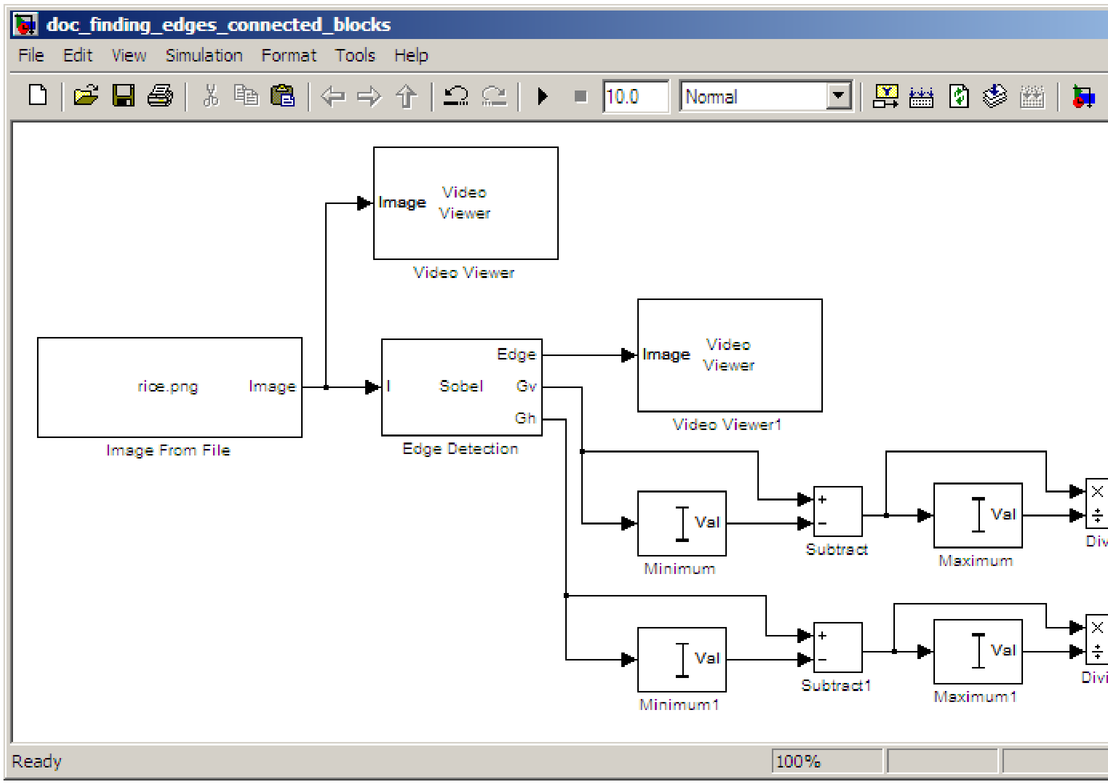
"Measure an Angle Between Lines" on page 3-14

Detect Edges in Images

This example finds the edges of rice grains in an intensity image. It does this by finding the pixel locations where the magnitude of the gradient of intensity exceeds a threshold value. These locations typically occur at the boundaries of objects.

Open the Simulink model

```
ex_vision_detect_edges_in_image
```



Setting block parameters

Block	Parameter setting
Image From File	• **File name** to `rice.png`. • **Output data type** to `single`.
Edge Detection	Use the Edge Detection block to find the edges in the image. • **Output type** = `Binary image and gradient components` • Select the **Edge thinning** check box.

Block	Parameter setting
Video Viewer and **Video Viewer1**	View the original and binary images. Accept the default parameters for both viewers.
2-D Minimum and **2-D Minimum1**	Find the minimum value of Gv and Gh matrices. Set the **Mode** parameters to `Value` for both of these blocks.
Subtract and **Subtract1**	Subtract the minimum values from each element of the Gv and Gh matrices. This process ensures that the minimum value of these matrices is `0`. Accept the default parameters.
2-D Maximum and **2-D Maximum1**	Find the maximum value of the new Gv and Gh matrices. Set the **Mode** parameters to `Value` for both of these blocks.
Divide and **Divide1**	Divide each element of the Gv and Gh matrices by their maximum value. This normalization process ensures that these matrices range between `0` and `1`. Accept the default parameters.
Video Viewer2 and **Video Viewer3**	View the gradient components of the image. Accept the default parameters.

Setting configuration parameters

Open the Configuration dialog box by selecting **Simulation** > **Configuration Parameters**. The parameters are set as follows:

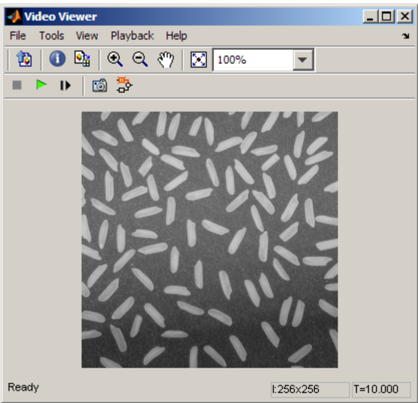
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

- **Diagnostics** pane, **Automatic solver parameter selection:** = none

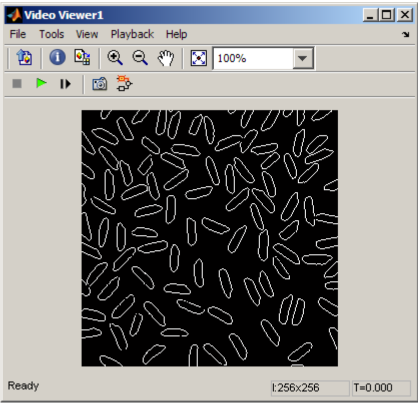
View edge detection results

Run your model.

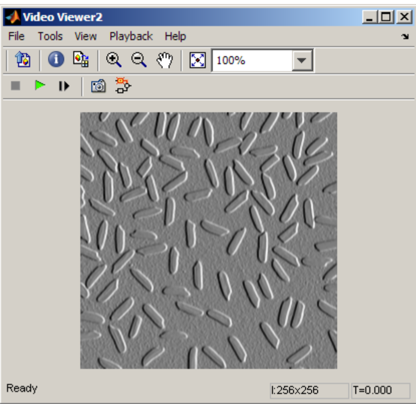
The Video Viewer window displays the original image.



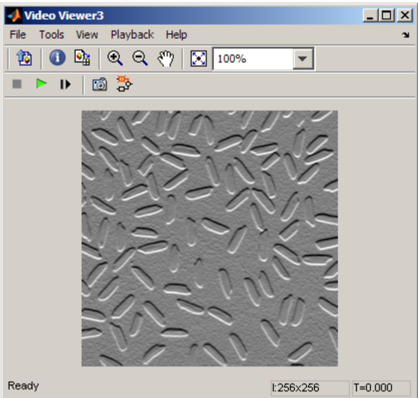
The Video Viewer1 window displays the edges of the rice grains in white and the background in black.



The Video Viewer2 window displays the intensity image of the vertical gradient components of the image. You can see that the vertical edges of the rice grains are darker and more well defined than the horizontal edges.



The Video Viewer3 window displays the intensity image of the horizontal gradient components of the image. In this image, the horizontal edges of the rice grains are more well defined.



The edge detection workflow

The Edge Detection block convolves the input matrix with the Sobel kernel. This calculates the gradient components of the image that correspond to the

horizontal and vertical edge responses. The block outputs these components at the **Gh** and **Gv** ports, respectively. Then the block performs a thresholding operation on the gradient components to find the binary image. The binary image is a matrix filled with 1s and 0s. The nonzero elements of this matrix correspond to the edge pixels and the zero elements correspond to the background pixels. The block outputs the binary image at the **Edge** port.

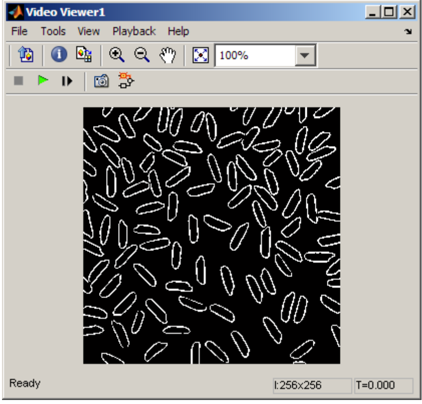
The matrix values at the **Gv** and **Gh** output ports of the Edge Detection block are double-precision floating-point. These matrix values need to be scaled between 0 and 1 in order to display them using the Video Viewer blocks. This is done with the Statistics and Math Operation blocks.

Running the model faster

Double-click the Edge Detection block and clear the **Edge thinning** check box.

Run your model.

Your model runs faster because the Edge Detection block is more efficient when you clear the **Edge thinning** check box. However, the edges of rice grains in the Video Viewer window are wider.



Close the model

```
bdclose('ex_vision_detect_edges_in_image');
```

You have now used the Edge Detection block to find the object boundaries in an image. For more information on this block, see the Edge Detection block reference page in the *Computer Vision System Toolbox Reference*.

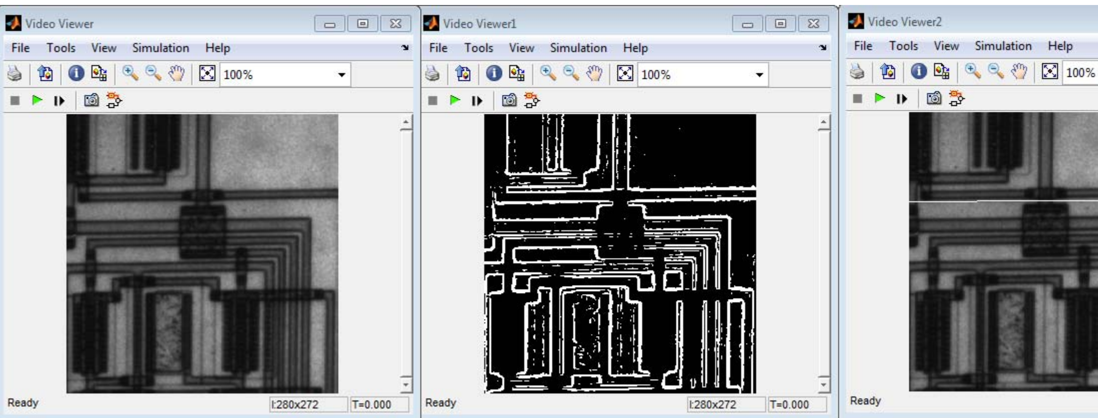
Detect Lines in Images

Finding lines within images enables you to detect, measure, and recognize objects. In this example, you use the Hough Transform, Find Local Maxima, Edge Detectionand Hough Lines blocks to find the longest line in an image.

You can open the model for this example by typing

```
ex_vision_detect_lines_in_image
```

at the MATLAB command line.



The Video Viewer blocks display the original image, the image with all edges found, and the image with the longest line annotated.

The Edge Detection block finds the edges in the intensity image. This process improves the efficiency of the Hough Lines block by reducing the image area over which the block searches for lines. The block also converts the image to a binary image, which is the required input for the Hough Transform block.

For additional examples of the techniques used in this section, see the following list of demos. You can open these demos by typing the demo titles at the MATLAB command prompt:

Demo	MATLAB	Simulink model-based
Lane Departure Warning System	`videoldws`	`vipldws`
Rotation Correction	`videorotationcorrection`	`viphough`

You can find all demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line.

Setting Block Parameters

Block	Parameter setting
Hough Transform	The Hough Transform block computes the Hough matrix by transforming the input image into the rho-theta parameter space. The block also outputs the rho and theta values associated with the Hough matrix. The parameters are set as follows: • **Theta resolution (radians)** = `pi/360` • Select the **Output theta and rho values** check box.
Find Local Maxima	The Find Local Maxima block finds the location of the maximum value in the Hough matrix. The block parameters are set as follows: • **Maximum number of local maxima** = `1` • **Input is Hough matrix spanning full theta range**

Block	Parameter setting
Selector	The Selector blocks separate the indices of the rho and theta values, which the Find Local Maxima block outputs at the **Idx** port. The rho and theta values correspond to the maximum value in the Hough matrix. The Selector blocks parameters are set as follows: • **Number of input dimensions:** `1` • **Index mode** = `One-based` • **Index Option** = `Index vector (port)` • **Input port size** = `2`
Variable Selector	The Variable Selector blocks index into the rho and theta vectors and determine the rho and theta values that correspond to the longest line in the original image. The parameters of the Variable Selector blocks are set as follows: • **Select** = `Columns` • **Index mode** = `One-based`

Block	Parameter setting
Hough Lines	The Hough Lines block determines where the longest line intersects the edges of the original image. • **Sine value computation method** = `Trigonometric function`
Draw Shapes	The Draw Shapes block draws a white line over the longest line on the original image. The coordinates are set to superimpose a line on the original image. The block parameters are set as follows: • **Shape** = `Lines` • **Border color** = `White`

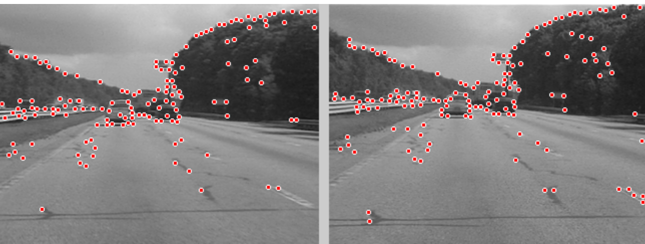
Configuration Parameters

Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `discrete (no continuous states)`
- **Solver** pane, **Fixed-step size (fundamental sample time):** = `0.2`

Detect Corner Features in an Image

Stabilizing a video that was captured from a jittery or moving platform is an important application in computer vision. One way to stabilize a video is to track a salient feature in the image and use this as an anchor point to cancel out all perturbations relative to it. This example uses corner detection around salient image features.



You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line. You can also launch the Video Stabilization Using Point Feature Matching demo model directly, by typing `videostabilize_pm` on the MATLAB command line.

Find Possible Point Matches Between Two Images

Stabilizing a video that was captured from a jittery or moving platform is an important application in computer vision. One way to stabilize a video is to track a salient feature in the image and use this as an anchor point to cancel out all perturbations relative to it. This procedure, however, must be bootstrapped with knowledge of where such a salient feature lies in the first video frame. In this example, we explore a method of video stabilization that works without any such a priori knowledge. It instead automatically searches for the "background plane" in a video sequence, and uses its observed distortion to correct for camera motion.

You can launch this example, Video Stabilization Using Point Feature Matching directly, by typing `videostabilize_pm` on the MATLAB command line.

Video mosaicking is the process of stitching video frames together to form a comprehensive view of the scene. The resulting mosaic image is a compact representation of the video data, which is often used in video compression and surveillance applications. This example illustrates how to use the CornerDetector, GeometricTransformEstimator, AlphaBlender, and the GeometricTransformer System objects to create a mosaic image from a video sequence. First, the example identifies the corners in the first reference frame and second video frame. Then, it calculates the projective transformation matrix that best describes the transformation between corner positions in these frames. Finally, the demo overlays the second image onto the first image. The example repeats this process to create a mosaic image of the video scene.

You can launch this example, Video Mosaicking directly, by typing `videomosaicking` on the MATLAB command line.

You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line.

Measure an Angle Between Lines

The Hough Transform, Find Local Maxima, and Hough Lines blocks enable you to find lines in images. With the Draw Shapes block, you can annotate images. In the following example, you use these capabilities to draw lines on the edges of two beams and measure the angle between them.

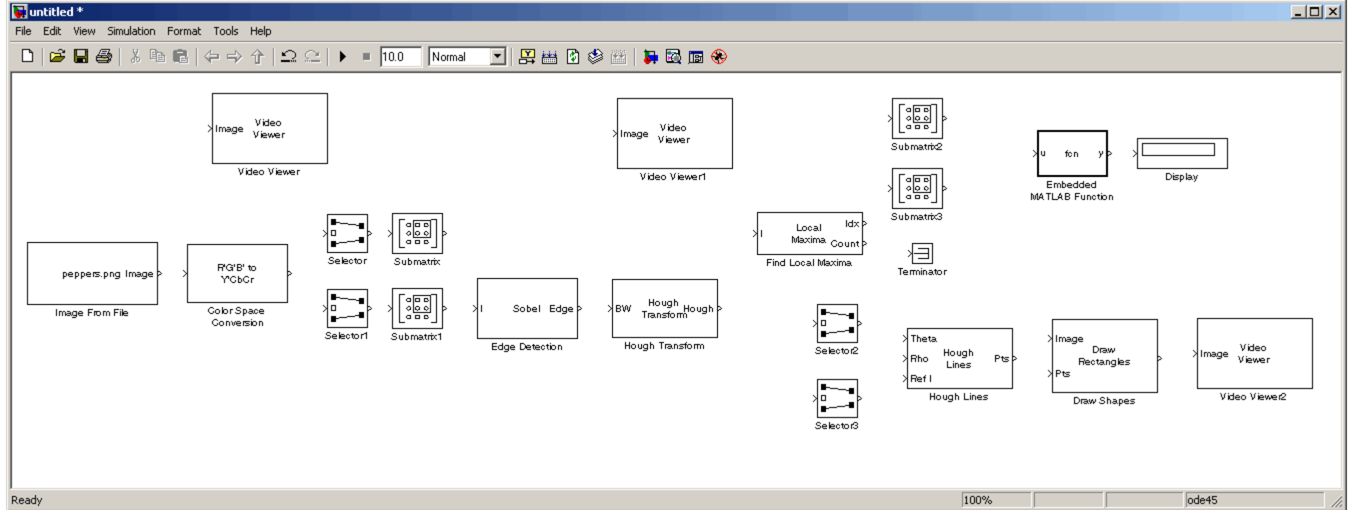
Running this example requires a DSP System Toolbox license.

1 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Color Space Conversion	Computer Vision System Toolbox > Conversions	1

Block	Library	Quantity
Edge Detection	Computer Vision System Toolbox > Analysis & Enhancement	1
Hough Transform	Computer Vision System Toolbox > Transforms	1
Find Local Maxima	Computer Vision System Toolbox > Statistics	1
Draw Shapes	Computer Vision System Toolbox > Text & Graphics	1
Video Viewer	Computer Vision System Toolbox > Sinks	3
Hough Lines	Computer Vision System Toolbox > Transforms	1
Submatrix	DSP System Toolbox > Math Functions > Matrices and Linear Algebra > Matrix Operations	4
Terminator	Simulink > Sinks	1
Selector	Simulink > Signal Routing	4
MATLAB Function	Simulink > User-Defined Functions	1
Display	Simulink > Sinks	1

2 Position the blocks as shown in the following figure.



3 Use the Image From File block to import an image into the Simulink model. Set the parameters as follows:

- **File name** = `gantrycrane.png`
- **Sample time** = `1`

4 Use the Color Space Conversion block to convert the RGB image into the Y'CbCr color space. You perform this conversion to separate the luma information from the color information. Accept the default parameters.

Note In this example, you segment the image using a thresholding operation that performs best on the Cb channel of the Y'CbCr color space.

---

5 Use the Selector and Selector1 blocks to separate the Y' (luminance) and Cb (chrominance) components from the main signal.

The Selector block separates the Y' component from the entire signal. Set its block parameters as follows:

- **Number of input dimensions** = `3`
- **Index mode** = `One-based`

- **1**
  - **Index Option** = `Select all`
- **2**
  - **Index Option** = `Select all`
- **3**
  - **Index Option** = `Index vector (dialog)`
  - **Index** = `1`

The Selector1 block separates the Cb component from the entire signal. Set its block parameters as follows:

- **Number of input dimensions** = `3`
- **Index mode** = `One-based`
- **1**
  - **Index Option** = `Select all`
- **2**
  - **Index Option** = `Select all`
- **3**
  - **Index Option** = `Index vector (dialog)`
  - **Index** = `2`

6. Use the Submatrix and Submatrix1 blocks to crop the Y' and Cb matrices to a particular region of interest (ROI). This ROI contains two beams that are at an angle to each other. Set the parameters as follows:

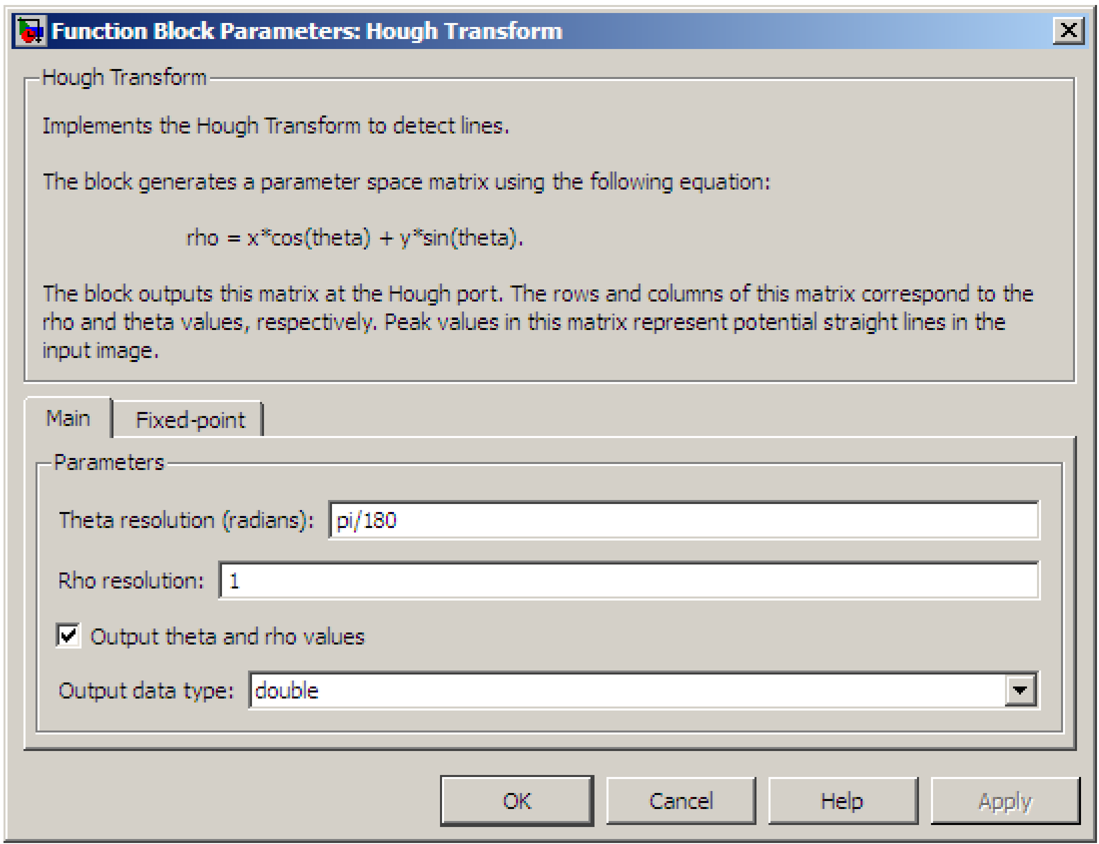
   - **Starting row** = `Index`
   - **Starting row index** = `66`
   - **Ending row** = `Index`
   - **Ending row index** = `150`
   - **Starting column** = `Index`
   - **Starting column index** = `325`

- **Ending column** = `Index`
- **Ending column index** = `400`

7 Use the Edge Detection block to find the edges in the Cb portion of the image. This block outputs a binary image. Set the **Threshold scale factor** parameter to `1`.

8 Use the Hough Transform block to calculate the Hough matrix, which gives you an indication of the presence of lines in an image. Select the **Output theta and rho values** check box as shown in the following figure.

Note In step 11, you find the theta and rho values that correspond to the peaks in the Hough matrix.



9 Use the Find Local Maxima block to find the peak values in the Hough matrix. These values represent potential lines in the input image. Set the parameters as follows:

- **Neighborhood size** = [11 11]
- **Input is Hough matrix spanning full theta range** = selected

Because you are expecting two lines, leave the **Maximum number of local maxima (N)** parameter set to 2, and connect the Count port to the Terminator block.

10. Use the Submatrix2 block to find the indices that correspond to the theta values of the two peak values in the Hough matrix. Set the parameters as follows:

    - **Starting row** = `Index`
    - **Starting row index** = `2`
    - **Ending row** = `Index`
    - **Ending row index** = `2`

    The Idx port of the Find Local Maxima block outputs a matrix whose second row represents the One-based indices of the theta values that correspond to the peaks in the Hough matrix. Now that you have these indices, you can use a Selector block to extract the corresponding theta values from the vector output of the Hough Transform block.

11. Use the Submatrix3 block to find the indices that correspond to the rho values of the two peak values in the Hough matrix. Set the parameters as follows:

    - **Ending row** = `Index`
    - **Ending row index** = `1`

    The Idx port of the Find Local Maxima block outputs a matrix whose first row represents the One-based indices of the rho values that correspond to the peaks in the Hough matrix. Now that you have these indices, you can use a Selector block to extract the corresponding rho values from the vector output of the Hough Transform block.

12. Use the Selector2 and Selector3 blocks to find the theta and rho values that correspond to the peaks in the Hough matrix. These values, output by the Hough Transform block, are located at the indices output by the Submatrix2 and Submatrix3 blocks. Set both block parameters as follows:

    - **Index mode** = `One-based`
    - **1**
      - **Index Option** = `Index vector (port)`
    - **Input port size** = `-1`

You set the **Index mode** to `One-based` because the Find Local Maxima block outputs One-based indices at the Idx port.

13 Use the Hough Lines block to find the Cartesian coordinates of lines that are described by rho and theta pairs. Set the **Sine value computation method** parameter to `Trigonometric function`.

14 Use the Draw Shapes block to draw the lines on the luminance portion of the ROI. Set the parameters as follows:

- **Shape** = `Lines`
- **Border color** = `White`

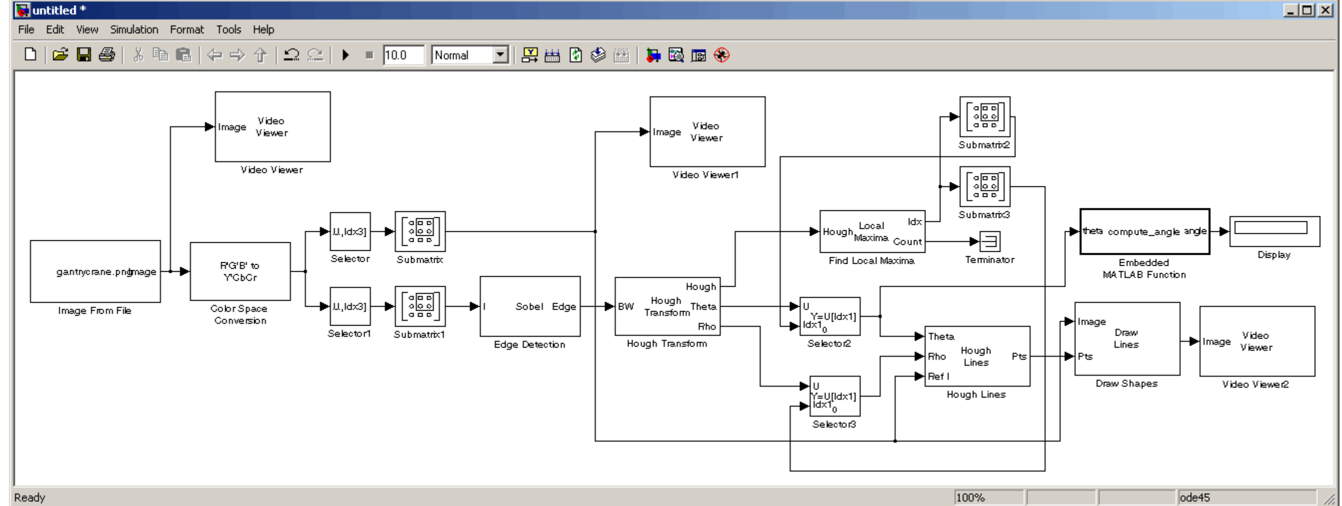
15 Use the MATLAB Function block to calculate the angle between the two lines. Copy and paste the following code into the block:

```
function angle = compute_angle(theta)

%Compute the angle value in degrees
angle = abs(theta(1)-theta(2))*180/pi;
%Always return an angle value less than 90 degrees
if (angle>90)
    angle = 180-angle;
end
```

16 Use the Display block to view the angle between the two lines. Accept the default parameters.

17 Use the Video Viewer blocks to view the original image, the ROI, and the annotated ROI. Accept the default parameters.

18 Connect the blocks as shown in the following figure.

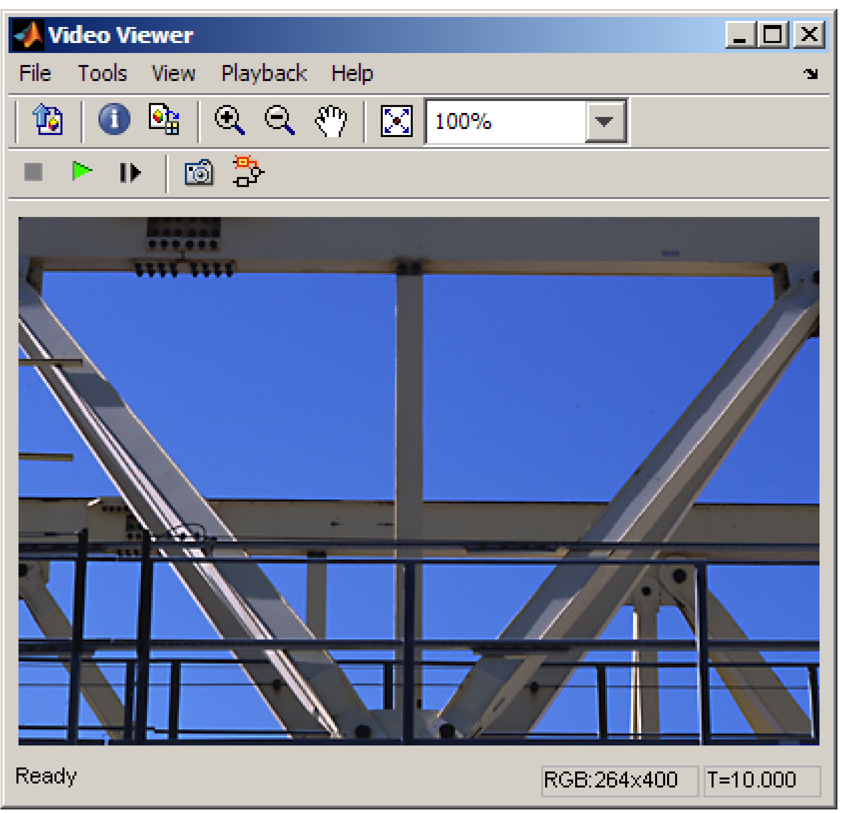


19 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

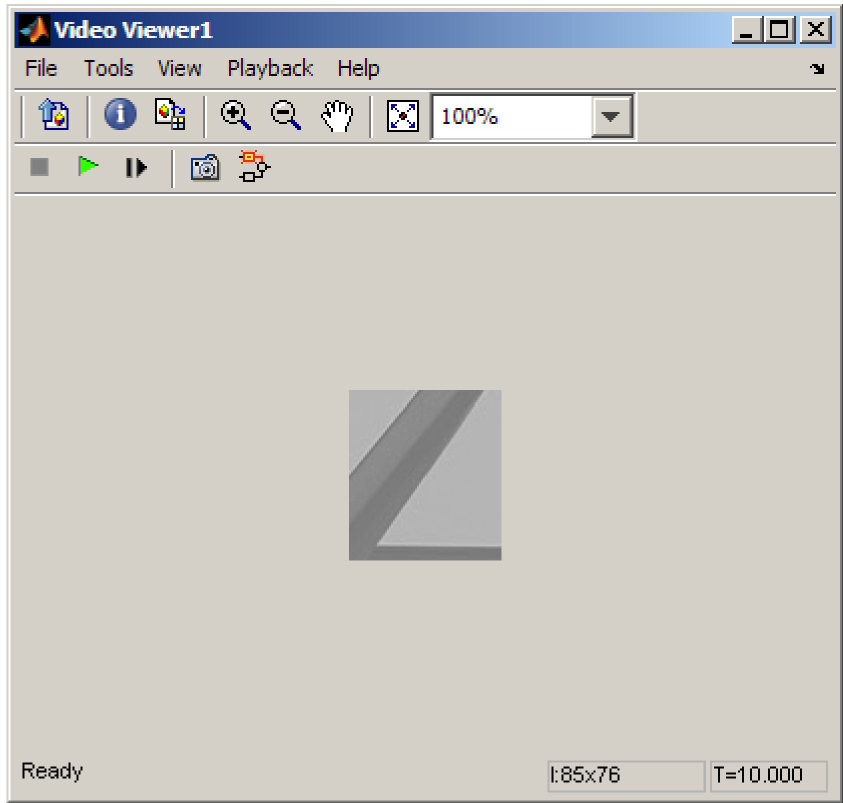
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

20 Run the model.

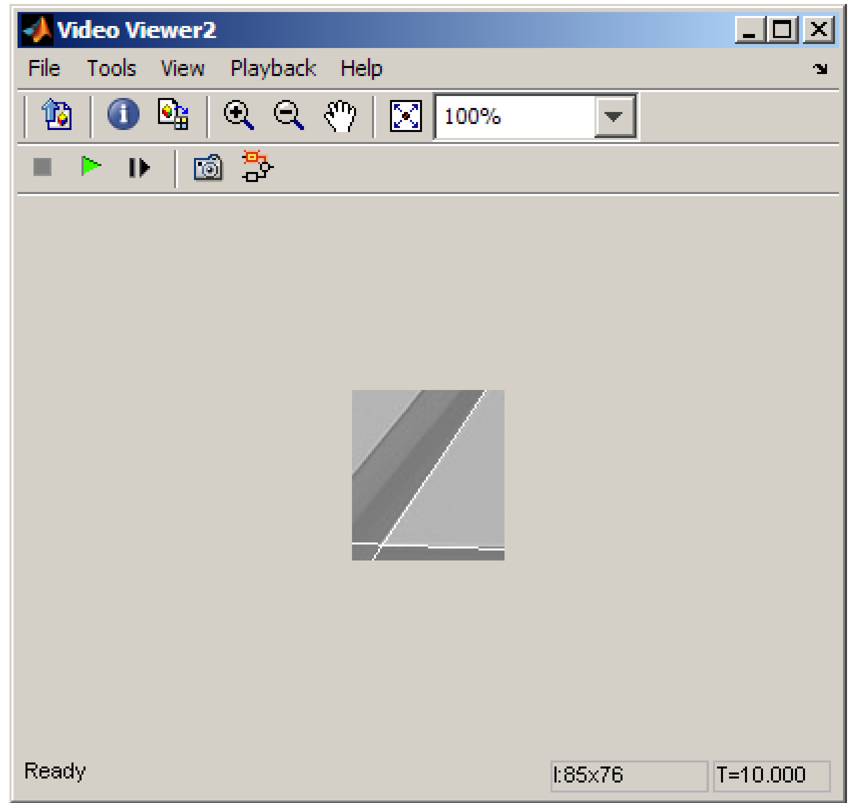
The Video Viewer window displays the original image.



The Video Viewer1 window displays the ROI where two beams intersect.



The Video Viewer2 window displays the ROI that has been annotated with two white lines.



The Display block shows a value of 58, wich is the angle in degrees between the two lines on the annotated ROI.

You have now annotated an image with two lines and measured the angle between them. For additional information, see the Hough Transform, Find Local Maxima, Hough Lines, and Draw Shapes block reference pages.

Image Registration

In this section...

"Automatically Determine Geometric Transform for Image Registration" on page 3-26

"Transform Images and Display Registration Results" on page 3-27

"Remove the Effect of Camera Motion from a Video Stream." on page 3-28

Automatically Determine Geometric Transform for Image Registration

Stabilizing a video that was captured from a jittery or moving platform is an important application in computer vision. One way to stabilize a video is to track a salient feature in the image and use this as an anchor point to cancel out all perturbations relative to it. This procedure, however, must be bootstrapped with knowledge of where such a salient feature lies in the first video frame. In this example, we explore a method of video stabilization that works without any such a priori knowledge. It instead automatically searches for the "background plane" in a video sequence, and uses its observed distortion to correct for camera motion.

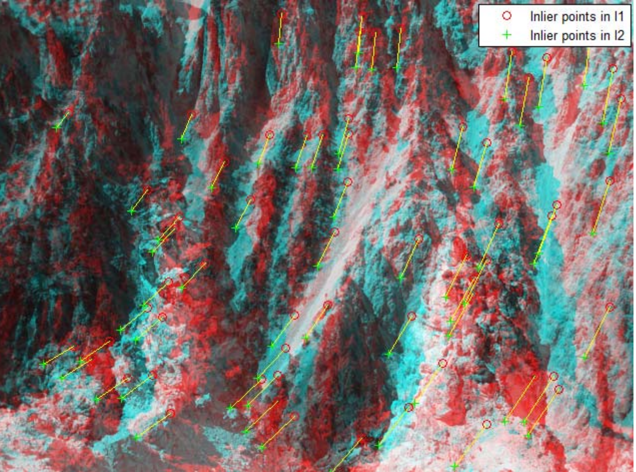
This stabilization algorithm involves two steps. First, we determine the affine image transformations between all neighboring frames of a video sequence using a Random Sampling and Consensus (RANSAC) [1] procedure applied to point correspondences between two images. Second, we warp the video frames to achieve a stabilized video. We use System objects in the Computer Vision System Toolbox™, both for the algorithm and for display.

You can launch this example, Video Stabilization Using Point Feature Matching directly, by typing `videostabilize_pm` on the MATLAB command line.

Transform Images and Display Registration Results

Rectification is the process of transforming stereo images, such that the corresponding points have the same row coordinates in the two images. It is a useful procedure in stereo vision, as the 2-D stereo correspondence problem is reduced to a 1-D problem when rectified image pairs are used.

The Image Rectification demo automatically registers and rectifies stereo images. This example detects corners in stereo images, matches the corners, computes the fundamental matrix, and then rectifies the images.

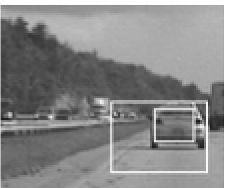


You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line. You can also launch the Image

Rectification demo model directly, by typing `videorectification` on the MATLAB command line.

Remove the Effect of Camera Motion from a Video Stream.

The video stabilization demo tracks a license plate of a vehicle while reducing the effect of camera motion from a video stream.



In the first video frame, the model defines the target to track. In this case, it is the back of a car and the license plate. It also establishes a dynamic search region, where the last known target location determines the position.

You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line. You can launch the Video Stabilization model directly by typing `vipstabilize` on the MATLAB command line.

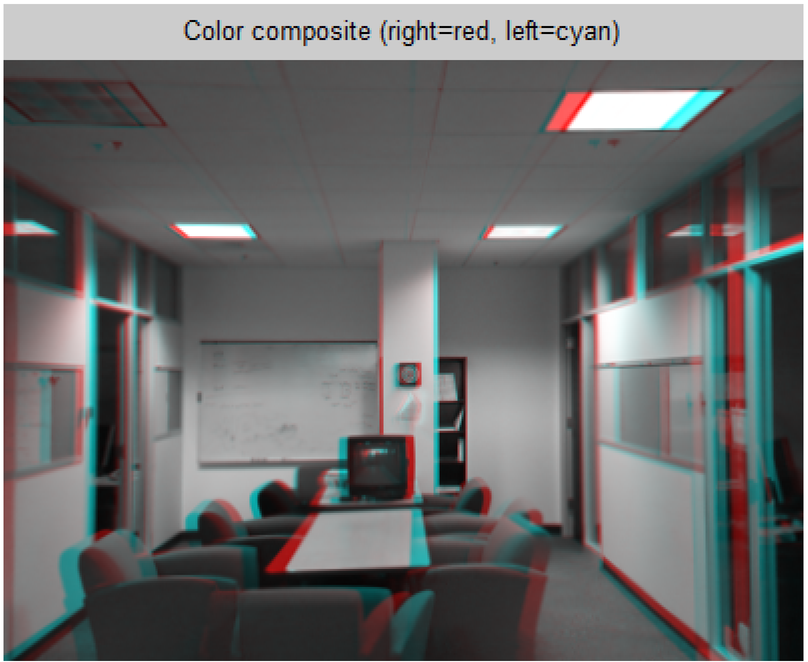
Stereo Vision

In this section...
“Compute Disparity Depth Map” on page 3-29
“Find Fundamental Matrix Describing Epipolar Geometry” on page 3-30
“Rectify Stereo Images” on page 3-32

Compute Disparity Depth Map

Stereo vision is the process of recovering depth from camera images by comparing two or more views of the same scene. Simple, binocular stereo uses only two images, typically taken with parallel cameras that were separated by a horizontal distance known as the "baseline." The output of the stereo computation is a disparity map (which is translatable to a range image) which tells how far each point in the physical scene was from the camera.

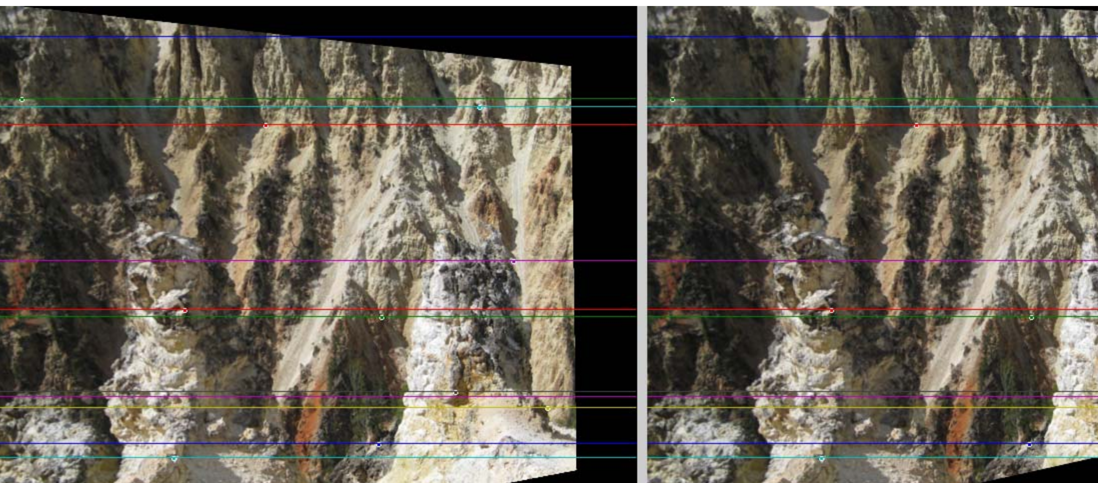
In this example, we use MATLAB® and the Computer Vision System Toolbox™ to compute the depth map between two rectified stereo images.



You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line. You can also launch the Stereo Vision demo model directly, by typing `videostereo` on the MATLAB command line.

Find Fundamental Matrix Describing Epipolar Geometry

In computer vision, the fundamental matrix is a 3×3 matrix which relates corresponding points in stereo images. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. Two images of the same scene are related by epipolar geometry.



This example takes two stereo images, computes the fundamental matrix from their corresponding points, displays the original stereo images, corresponding points, and epipolar lines for this and for the rectified images.

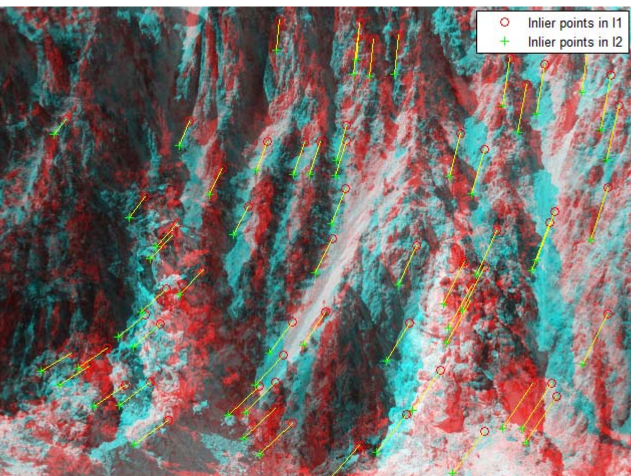
```
% Load the stereo images and convert them to double precision.
    I1 = imread('yellowstone_left.png');
    I2 = imread('yellowstone_right.png');

    % Load the points which are already matched.
    load yellowstone_matched_points;

    % Compute the fundamental matrix from the corresponding points.
    f= estimateFundamentalMatrix(matched_points1, matched_points2,...
      'Method', 'Norm8Point');

    % Display the original stereo images, corresponding points, and
    % epipolar lines.
    cvexShowStereoImages('Original image 1', 'Original image 2', ...
      I1, I2, matched_points1, matched_points2, f);
```

You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line.

Rectify Stereo Images

Rectification is the process of transforming stereo images, such that the corresponding points have the same row coordinates in the two images. It is a useful procedure in stereo vision, as the 2-D stereo correspondence problem is reduced to a 1-D problem when rectified image pairs are used.

The Image Rectification demo automatically registers and rectifies stereo images. This example detects corners in stereo images, matches the corners, computes the fundamental matrix, and then rectifies the images.



You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line. You can also launch the Image Rectification demo model directly, by typing `videorectification` on the MATLAB command line.

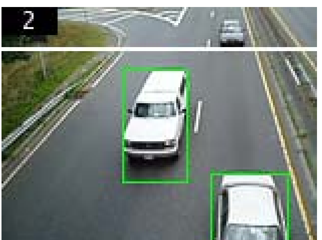
4

Motion Estimation and Tracking

- “Detect and Track Moving Objects Using Gaussian Mixture Models” on page 4-2
- “Video Mosaicking” on page 4-3
- “Track an Object Using Correlation” on page 4-4
- “Create a Panoramic Scene” on page 4-12

Detect and Track Moving Objects Using Gaussian Mixture Models

This example illustrates how to detect cars in a video sequence using foreground detection based on Gaussian mixture models (GMMs). After foreground detection, the example processes the binary foreground images using blob analysis. Finally, bounding boxes are drawn around the detected cars.



You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line. You can also launch the Tracking Cars Using Gaussian Mixture Models demo model directly, by typing `videotrafficgmm` at the MATLAB command line.

Video Mosaicking

Video mosaicking is the process of stitching video frames together to form a comprehensive view of the scene. The resulting mosaic image is a compact representation of the video data, which is often used in video compression and surveillance applications.

You can find demos for the Computer Vision System Toolbox by typing `visiondemos` at the MATLAB command line. You can also launch the Video Mosaicking demo model directly, by typing `videomosaicking` at the MATLAB command line.

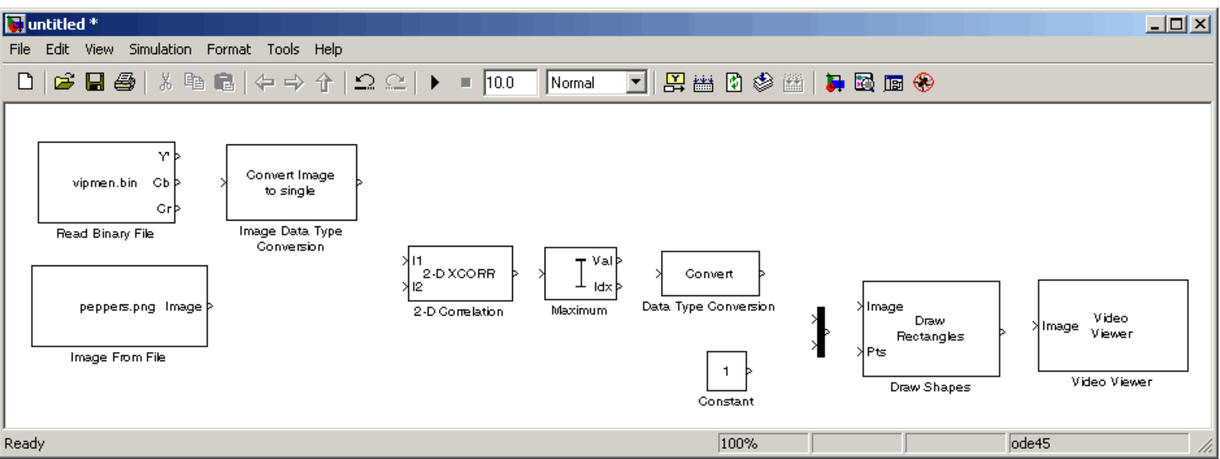
Track an Object Using Correlation

In this example, you use the 2-D Correlation, 2-D Maximum, and Draw Shapes blocks to find and indicate the location of a sculpture in each video frame:

1 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Read Binary File	Computer Vision System Toolbox > Sources	1
Image Data Type Conversion	Computer Vision System Toolbox > Conversions	1
Image From File	Computer Vision System Toolbox > Sources	1
2-D Correlation	Computer Vision System Toolbox > Statistics	1
2-D Maximum	Computer Vision System Toolbox > Statistics	1
Draw Shapes	Computer Vision System Toolbox > Text & Graphics	1
Video Viewer	Computer Vision System Toolbox > Sinks	1
Data Type Conversion	Simulink > Signal Attributes	1
Constant	Simulink > Sources	1
Mux	Simulink > Signal Routing	1

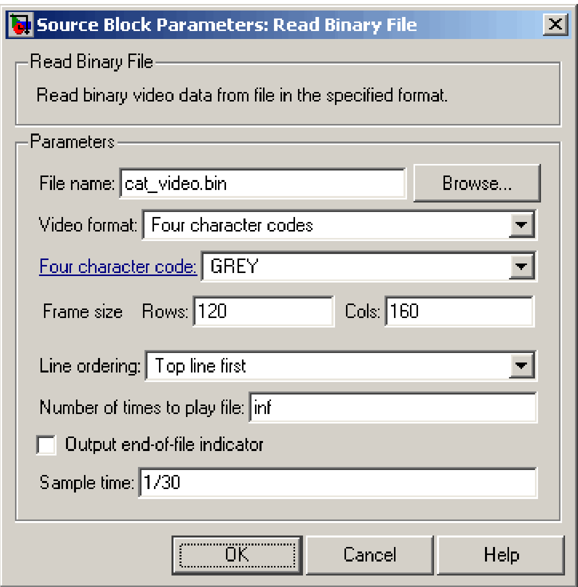
2 Position the blocks as shown in the following figure.



You are now ready to set your block parameters by double-clicking the blocks, modifying the block parameter values, and clicking **OK**.

3 Use the Read Binary File block to import a binary file into the model. Set the block parameters as follows:

- **File name** = `cat_video.bin`
- **Four character code** = `GREY`
- **Number of times to play file** = `inf`
- **Sample time** = `1/30`



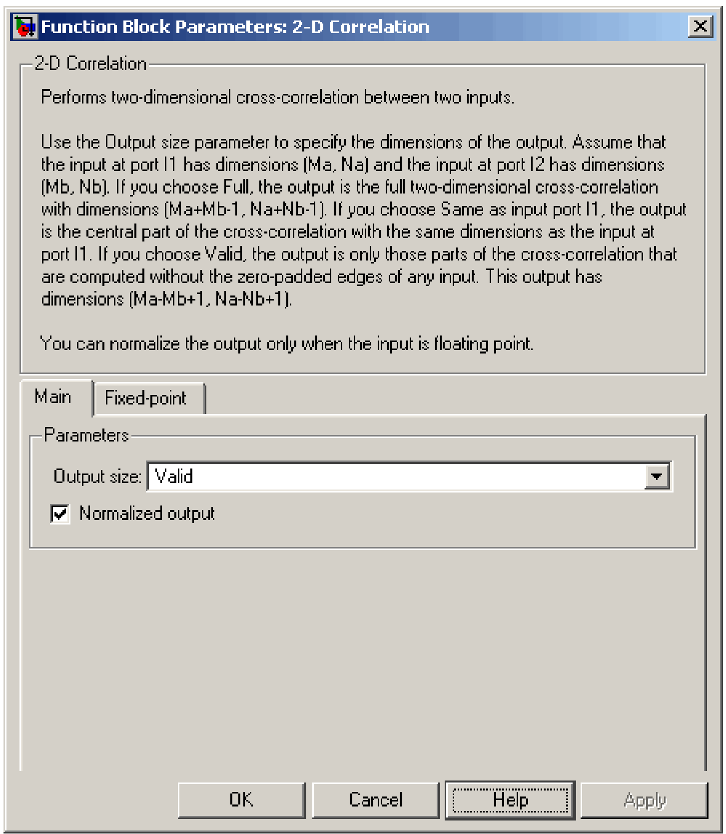
4 Use the Image Data Type Conversion block to convert the data type of the video to single-precision floating point. Accept the default parameter.

5 Use the Image From File block to import the image of the cat sculpture, which is the object you want to track. Set the block parameters as follows:

- **Main** pane, **File name** = `cat_target.png`
- **Data Types** pane, **Output data type** = `single`

6 Use the 2-D Correlation block to determine the portion of each video frame that best matches the image of the cat sculpture. Set the block parameters as follows:

- **Output size** = `Valid`
- Select the **Normalized output** check box.



Because you chose `Valid` for the **Output size** parameter, the block outputs only those parts of the correlation that are computed without the zero-padded edges of any input.

7 Use the 2-D Maximum block to find the index of the maximum value in each input matrix. Set the **Mode** parameter to `Index`.

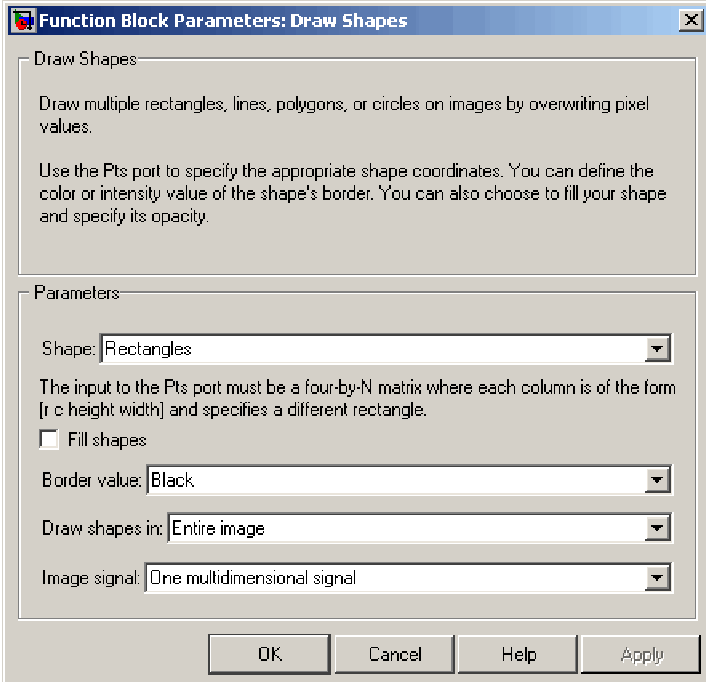
The block outputs the zero-based location of the maximum value as a two-element vector of 32-bit unsigned integers at the Idx port.

8 Use the Data Type Conversion block to change the index values from 32-bit unsigned integers to single-precision floating-point values. Set the **Output data type** parameter to `single`.

9 Use the Constant block to define the size of the image of the cat sculpture. Set the **Constant value** parameter to `single([41 41])`.

10 Use the Mux block to concatenate the location of the maximum value and the size of the image of the cat sculpture into a single vector. You use this vector to define a rectangular region of interest (ROI) that you pass to the Draw Shapes block.

11 Use the Draw Shapes block to draw a rectangle around the portion of each video frame that best matches the image of the cat sculpture. Accept the default parameters.

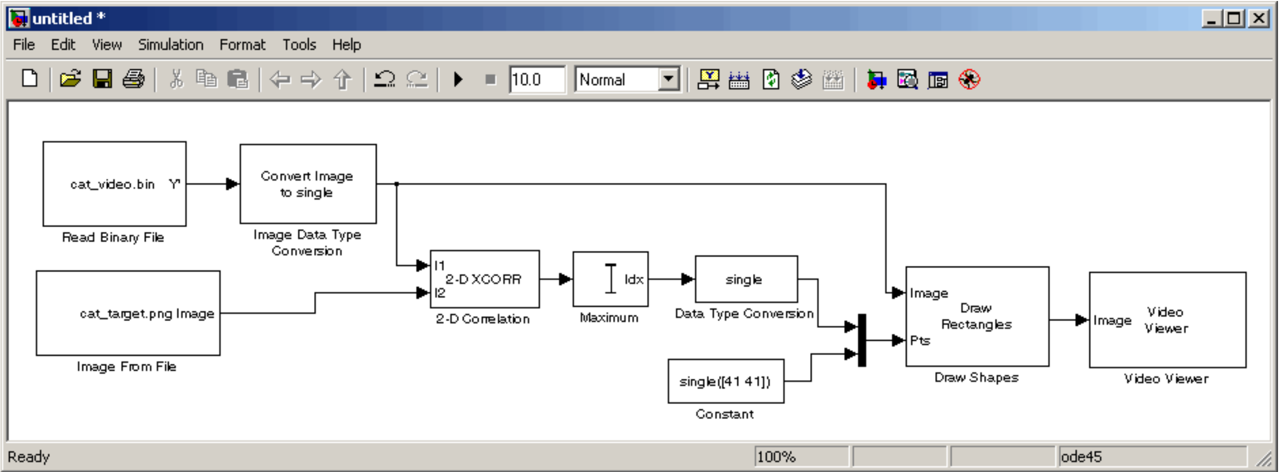


12 Use the Video Viewer block to display the video stream with the ROI displayed on it. Accept the default parameters.

The Video Viewer block automatically displays the video in the Video Viewer window when you run the model. Because the image is represented

by single-precision floating-point values, a value of 0 corresponds to black and a value of 1 corresponds to white.

13 Connect the blocks as shown in the following figure.

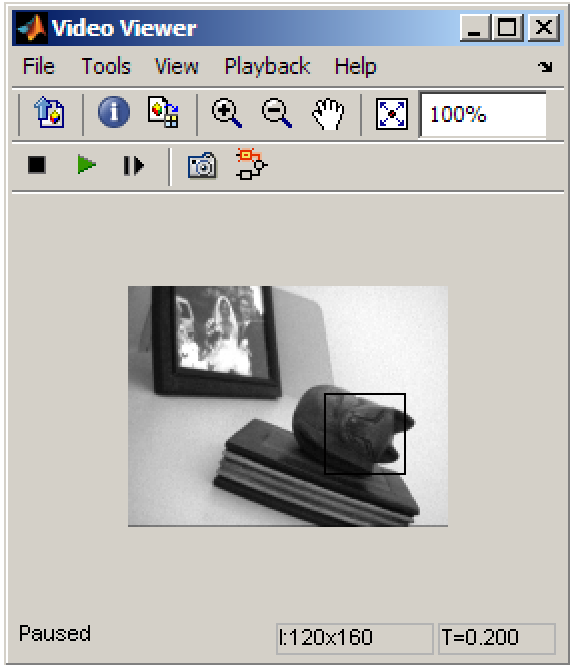


14 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

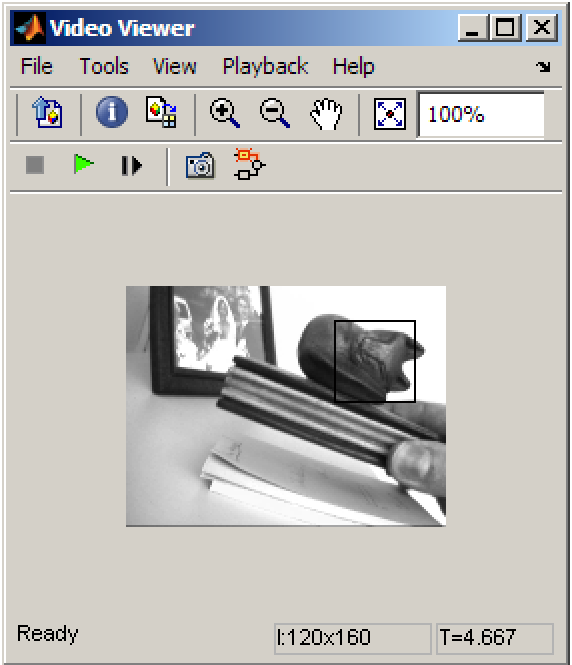
- **Solver** pane, **Stop time** = `inf`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

15 Run the simulation.

The video is displayed in the Video Viewer window and a rectangular box appears around the cat sculpture. To view the video at its true size, right-click the window and select **Set Display To True Size**.



As the video plays, you can watch the rectangular ROI follow the sculpture as it moves.



In this example, you used the 2-D Correlation, 2-D Maximum, and Draw Shapes blocks to track the motion of an object in a video stream. For more information about these blocks, see the 2-D Correlation, 2-D Maximum, and Draw Shapes block reference pages.

Note This example model does not provide an indication of whether or not the sculpture is present in each video frame. For an example of this type of model, type `vippattern` at the MATLAB command prompt.

Create a Panoramic Scene

The motion estimation subsystem for the Panorama Creation demo uses template matching. This model uses the Template Matching block to estimate the motion between consecutive video frames. It then computes the motion vector of a particular block in the current frame with respect to the previous frame. The model uses this motion vector to align consecutive frames of the video to form a panoramic picture.

You can find demos for the Computer Vision System Toolbox by typing `visiondemos` on the MATLAB command line. You can launch the Panorama model directly by typing `vippanorama` on the MATLAB command line.

5

Geometric Transformations

- "Rotate an Image" on page 5-2
- "Resize an Image" on page 5-10
- "Crop an Image" on page 5-16
- "Interpolation Methods" on page 5-22
- "Automatically Determine Geometric Transform for Image Registration" on page 5-26

Rotate an Image

You can use the Rotate block to rotate your image or video stream by a specified angle. In this example, you learn how to use the Rotate block to continuously rotate an image.

Running this example requires a DSP System Toolbox license.

1 Define an RGB image in the MATLAB workspace. At the MATLAB command prompt, type

```
I = checker_board;
```

`I` is a 100-by-100-by-3 array of double-precision values. Each plane of the array represents the red, green, or blue color values of the image.

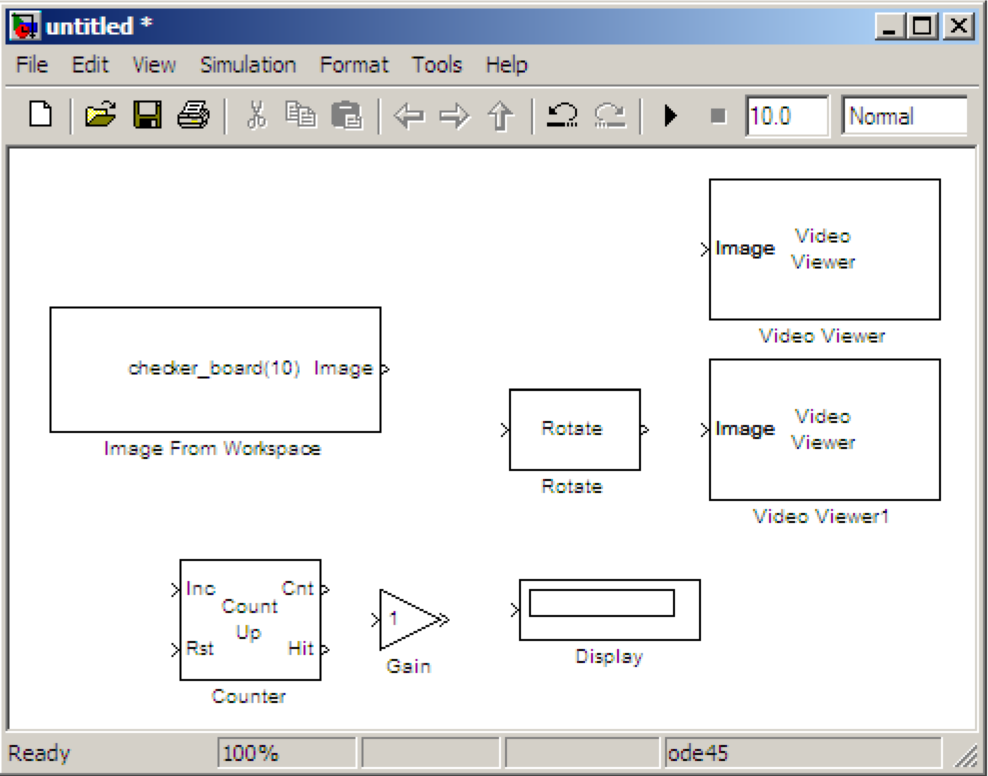
2 To view the image this matrix represents, at the MATLAB command prompt, type

```
imshow(I)
```

3 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From Workspace	Computer Vision System Toolbox > Sources	1
Rotate	Computer Vision System Toolbox > Geometric Transformations	1
Video Viewer	Computer Vision System Toolbox > Sinks	2
Gain	Simulink > Math Operations	1
Display	DSP System Toolbox > Sinks	1
Counter	DSP System Toolbox > Signal Management > Switches and Counters	1

4. Position the blocks as shown in the following figure.



You are now ready to set your block parameters by double-clicking the blocks, modifying the block parameter values, and clicking **OK**.

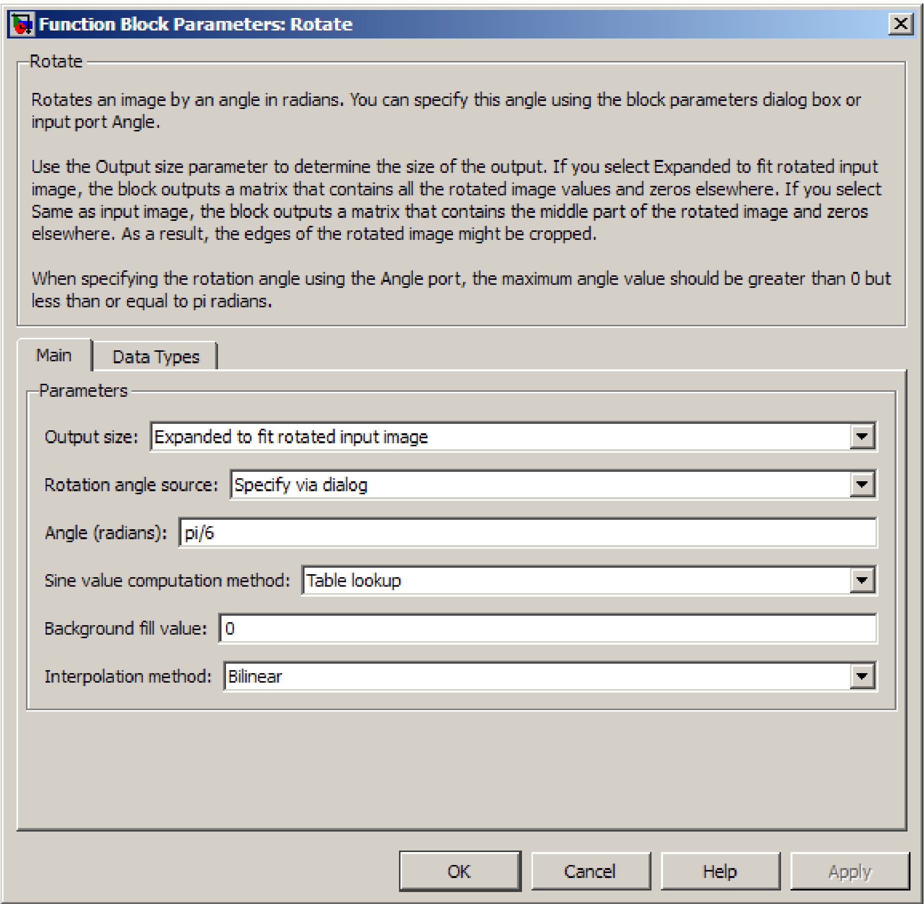
5 Use the Image From Workspace block to import the RGB image from the MATLAB workspace. On the Main pane, set the **Value** parameter to `I`.Each plane of the array represents the red, green, or blue color values of the image.

6 Use the Video Viewer block to display the original image. Accept the default parameters.

   The Video Viewer block automatically displays the original image in the Video Viewer window when you run the model. Because the image is represented by double-precision floating-point values, a value of 0 corresponds to black and a value of 1 corresponds to white.

7 Use the Rotate block to rotate the image. Set the block parameters as follows:

- **Rotation angle source** = `Input port`
- **Sine value computation method** = `Trigonometric function`



The Angle port appears on the block. You use this port to input a steadily increasing angle. Setting the **Output size** parameter to `Expanded to fit rotated input image` ensures that the block does not crop the output.

8 Use the Video Viewer1 block to display the rotating image. Accept the default parameters.

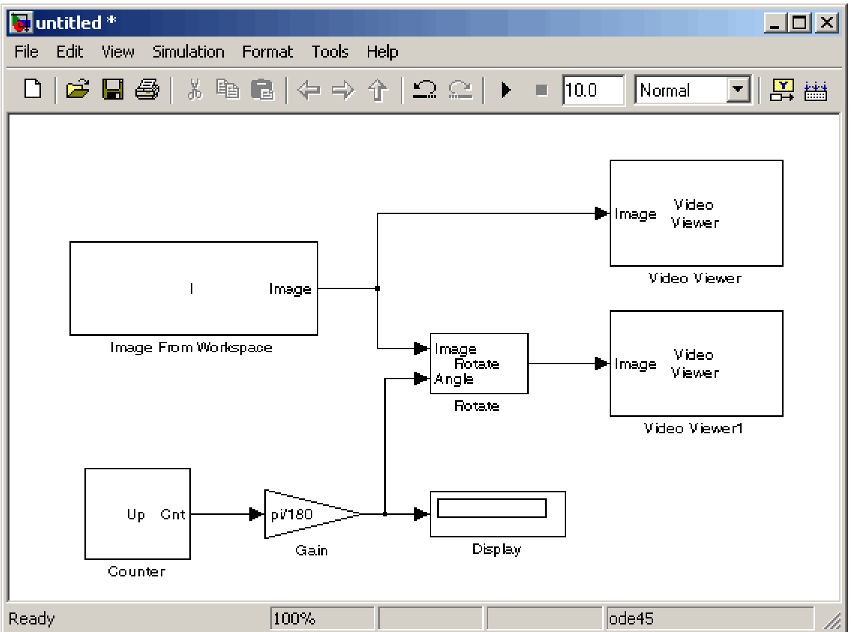
9 Use the Counter block to create a steadily increasing angle. Set the block parameters as follows:

- **Count event** = `Free running`
- **Counter size** = `16 bits`
- **Output** = `Count`
- Clear the **Reset input** check box.
- **Sample time** = `1/30`

The Counter block counts upward until it reaches the maximum value that can be represented by 16 bits. Then, it starts again at zero. You can view its output value on the Display block while the simulation is running. The Counter block's **Count data type** parameter enables you to specify it's output data type.

10 Use the Gain block to convert the output of the Counter block from degrees to radians. Set the **Gain** parameter to `pi/180`.

11 Connect the blocks as shown in the following figure.

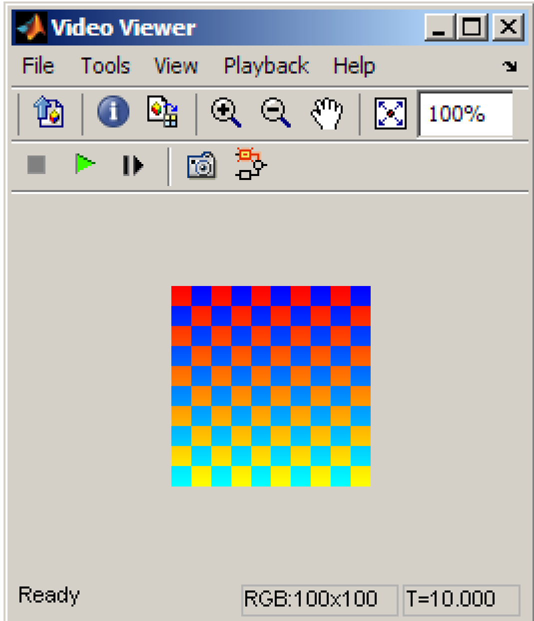


12 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

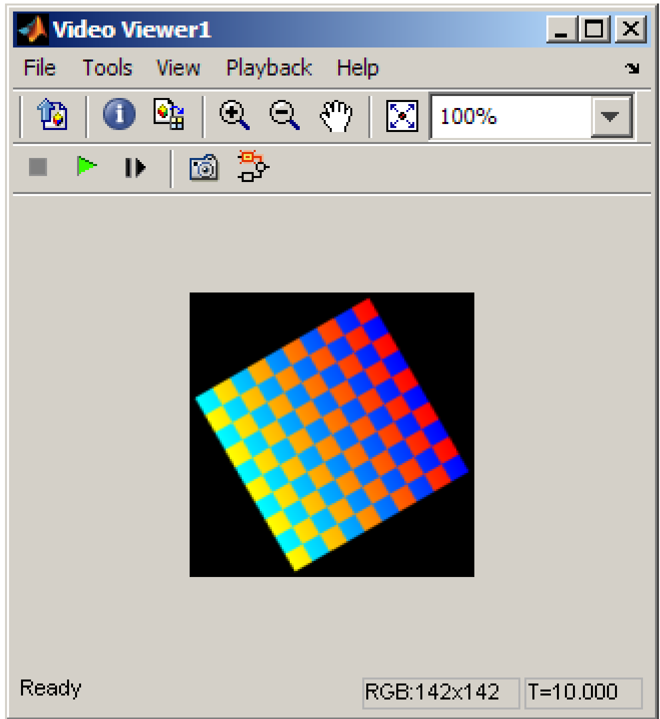
- **Solver** pane, **Stop time** = `inf`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

13 Run the model.

The original image appears in the Video Viewer window.



The rotating image appears in the Video Viewer1 window.



In this example, you used the Rotate block to continuously rotate your image. For more information about this block, see the Rotate block reference page in the *Computer Vision System Toolbox Reference*. For more information about other geometric transformation blocks, see the Resize and Shear block reference pages.

Note If you are on a Windows operating system, you can replace the Video Viewer block with the To Video Display block, which supports code generation.

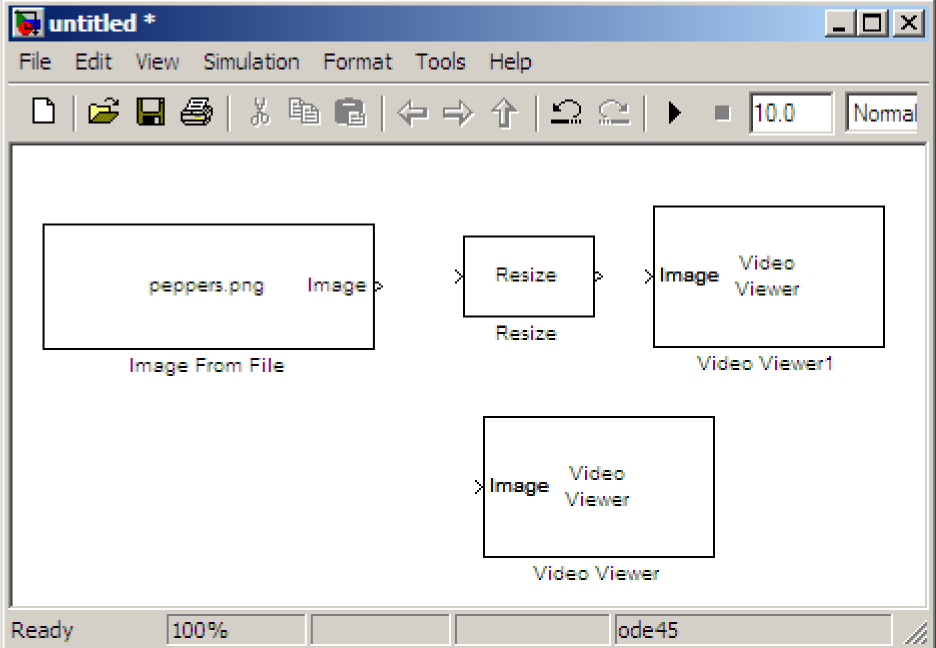
Resize an Image

You can use the Resize block to change the size of your image or video stream. In this example, you learn how to use the Resize block to reduce the size of an image:

1. Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Resize	Computer Vision System Toolbox > Geometric Transformations	1
Video Viewer	Computer Vision System Toolbox > Sinks	2

2. Position the blocks as shown in the following figure.

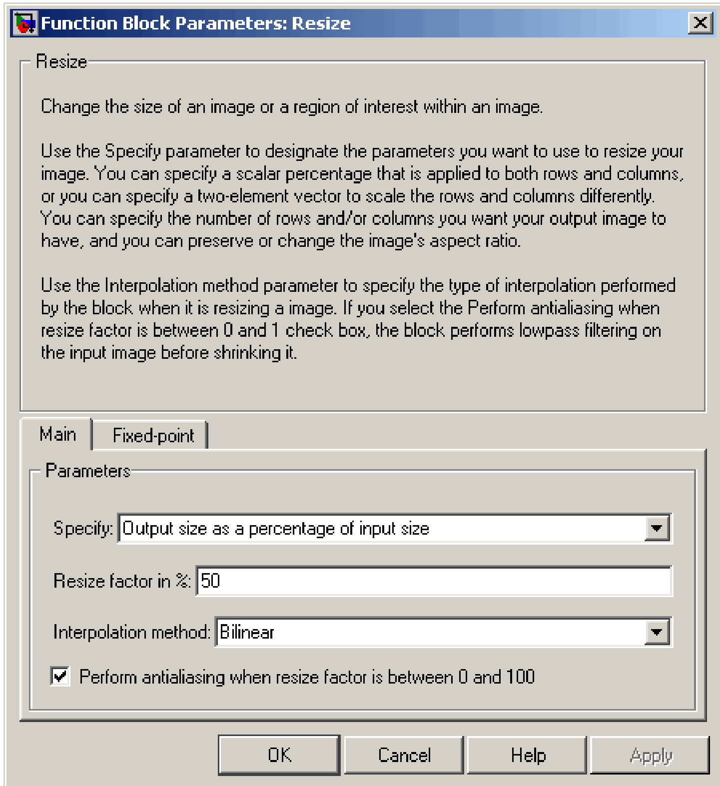


3 Use the Image From File block to import the intensity image. Set the **File name** parameter to `moon.tif`. The tif file is a 537-by-358 matrix of 8-bit unsigned integer values.

4 Use the Video Viewer block to display the original image. Accept the default parameters.

The Video Viewer block automatically displays the original image in the Video Viewer window when you run the model.

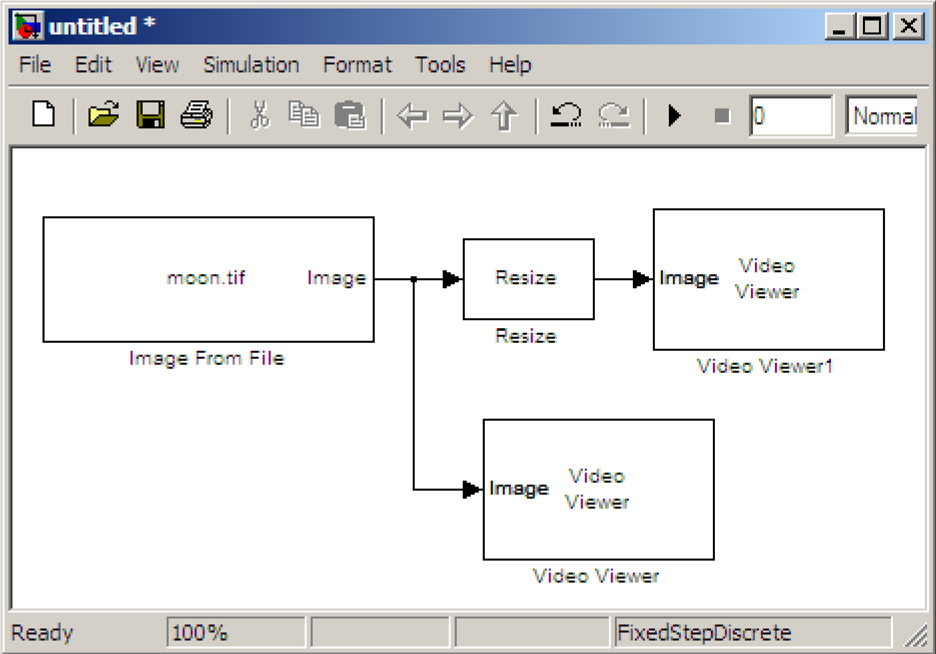
5 Use the Resize block to shrink the image. Set the **Resize factor in %** parameter to `50`.



The Resize block shrinks the image to half its original size.

6 Use the Video Viewer1 block to display the modified image. Accept the default parameters.

7 Connect the blocks as shown in the following figure.

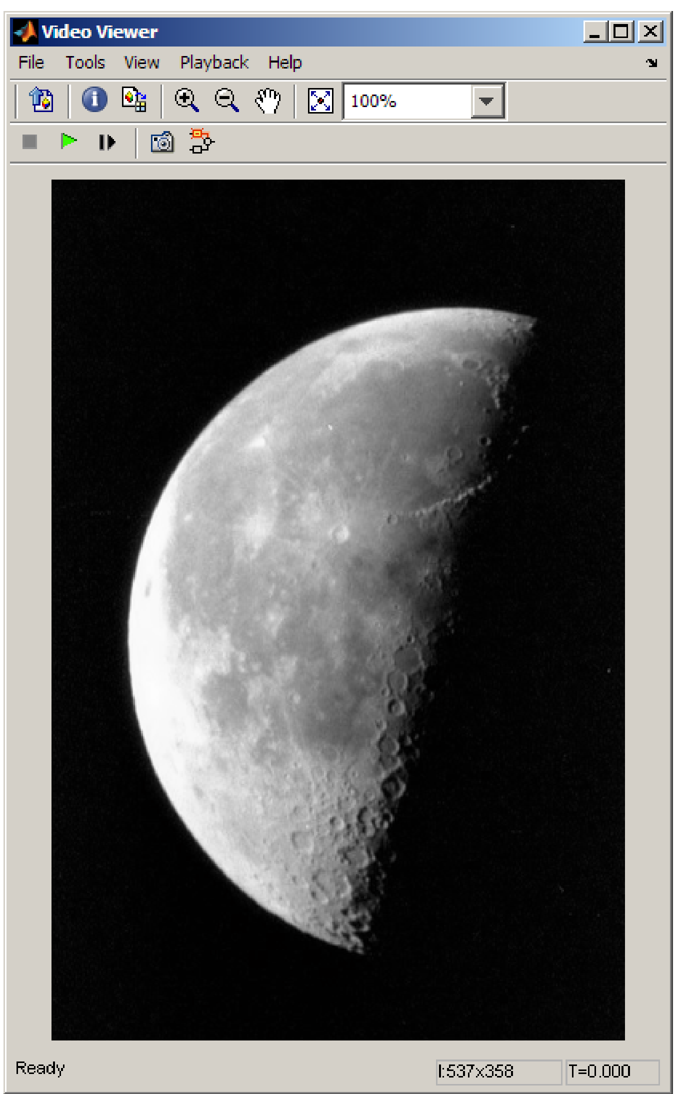


8 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

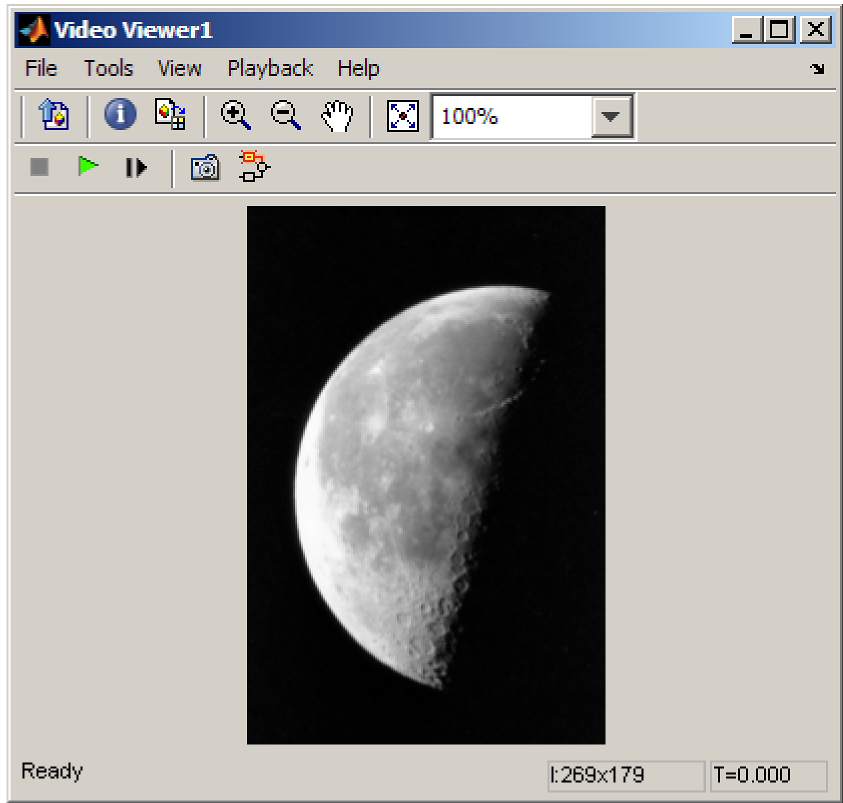
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

9 Run the model.

The original image appears in the Video Viewer window.



The reduced image appears in the Video Viewer1 window.



In this example, you used the Resize block to shrink an image. For more information about this block, see the Resize block reference page. For more information about other geometric transformation blocks, see the Rotate, Shear, and Translate block reference pages.

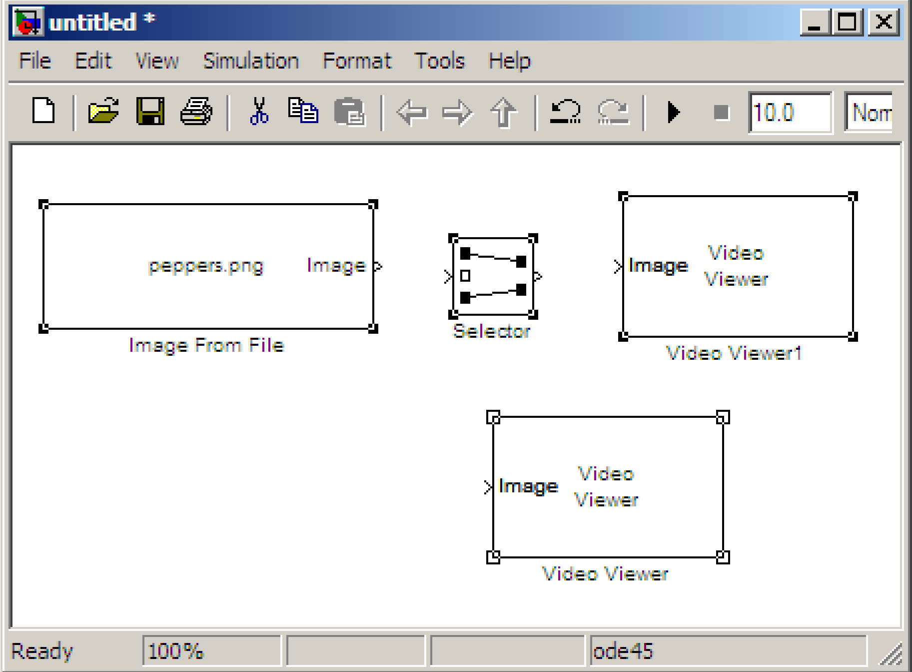
Crop an Image

You can use the Selector block to crop your image or video stream. In this example, you learn how to use the Selector block to trim an image down to a particular region of interest:

1. Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Video Viewer	Computer Vision System Toolbox > Sinks	2
Selector	Simulink > Signal Routing	1

2. Position the blocks as shown in the following figure.



3 Use the Image From File block to import the intensity image. Set the **File name** parameter to `coins.png`. The image is a 246-by-300 matrix of 8-bit unsigned integer values.

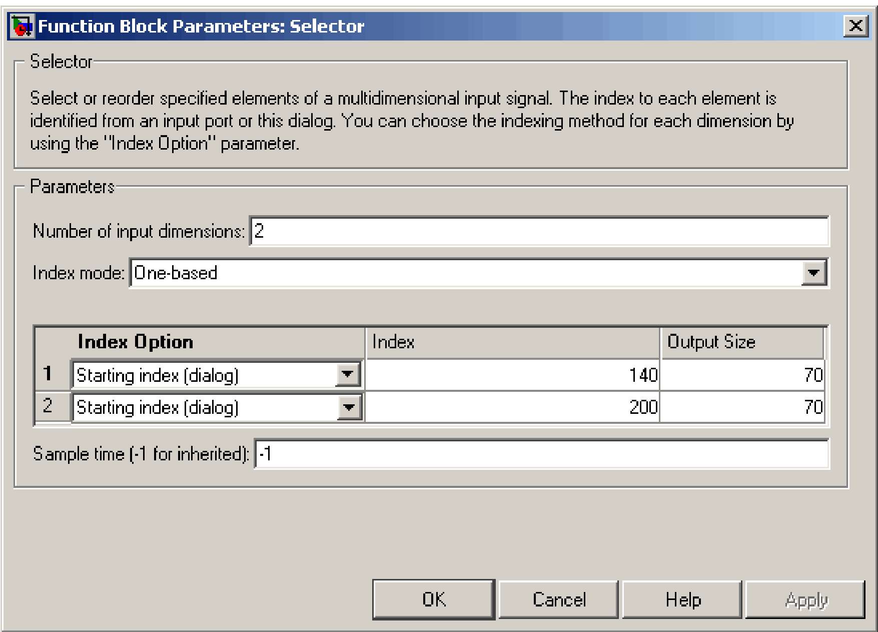
4 Use the Video Viewer block to display the original image. Accept the default parameters.

The Video Viewer block automatically displays the original image in the Video Viewer window when you run the model.

5 Use the Selector block to crop the image. Set the block parameters as follows:

- **Number of input dimensions** = 2
- **1**
  - **Index Option** = `Starting index (dialog)`
  - **Index** = 140

  - **Output Size** = 70
- **2**
  - **Index Option** = `Starting index (dialog)`
  - **Index** = 200
  - **Output Size** = 70

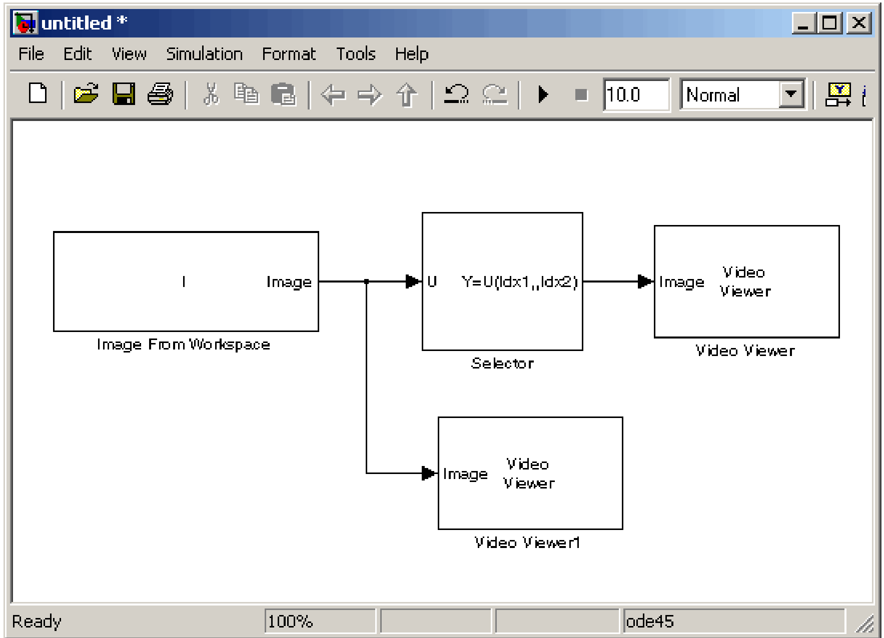


The Selector block starts at row 140 and column 200 of the image and outputs the next 70 rows and columns of the image.

6. Use the Video Viewer1 block to display the cropped image.

   The Video Viewer1 block automatically displays the modified image in the Video Viewer window when you run the model.

7. Connect the blocks as shown in the following figure.



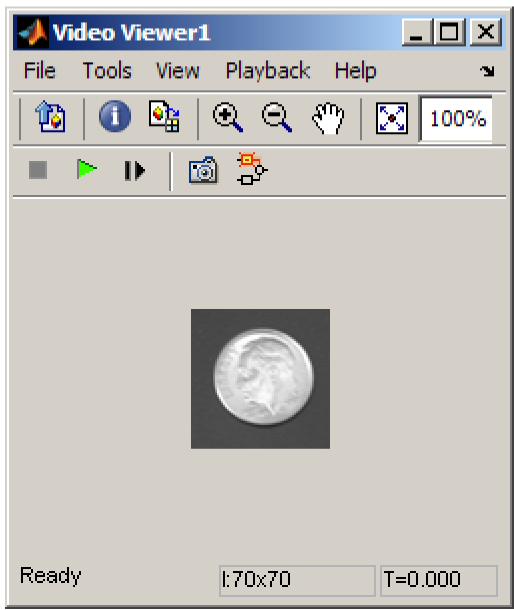
8 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

9 Run the model.

The original image appears in the Video Viewer window.

The cropped image appears in the Video Viewer window. The following image is shown at its true size.



In this example, you used the Selector block to crop an image. For more information about the Selector block, see the Simulink documentation. For information about the `imcrop` function, see the Image Processing Toolbox documentation.

Interpolation Methods

In this section...

"Nearest Neighbor Interpolation" on page 5-22
"Bilinear Interpolation" on page 5-23
"Bicubic Interpolation" on page 5-24

Nearest Neighbor Interpolation

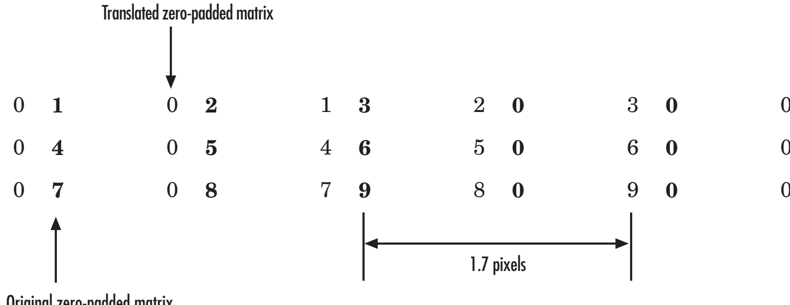
For nearest neighbor interpolation, the block uses the value of nearby translated pixel values for the output pixel values.

For example, suppose this matrix,

```
1   2   3
4   5   6
7   8   9
```

represents your input image. You want to translate this image 1.7 pixels in the positive horizontal direction using nearest neighbor interpolation. The Translate block's nearest neighbor interpolation algorithm is illustrated by the following steps:

1. Zero pad the input matrix and translate it by 1.7 pixels to the right.



2. Create the output matrix by replacing each input pixel value with the translated value nearest to it. The result is the following matrix:

```
0  0  1  2  3
0  0  4  5  6
0  0  7  8  9
```

Note You wanted to translate the image by 1.7 pixels, but this method translated the image by 2 pixels. Nearest neighbor interpolation is computationally efficient but not as accurate as bilinear or bicubic interpolation.

Bilinear Interpolation

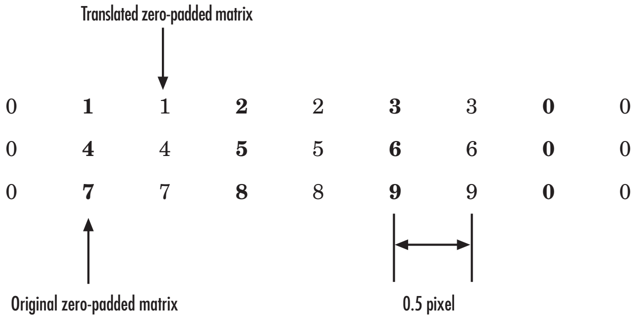
For bilinear interpolation, the block uses the weighted average of two translated pixel values for each output pixel value.

For example, suppose this matrix,

```
1  2  3
4  5  6
7  8  9
```

represents your input image. You want to translate this image 0.5 pixel in the positive horizontal direction using bilinear interpolation. The Translate block's bilinear interpolation algorithm is illustrated by the following steps:

1. Zero pad the input matrix and translate it by 0.5 pixel to the right.



2 Create the output matrix by replacing each input pixel value with the weighted average of the translated values on either side. The result is the following matrix where the output matrix has one more column than the input matrix:

0.5	1.5	2.5	1.5
2	4.5	5.5	3
3.5	7.5	8.5	4.5

Bicubic Interpolation

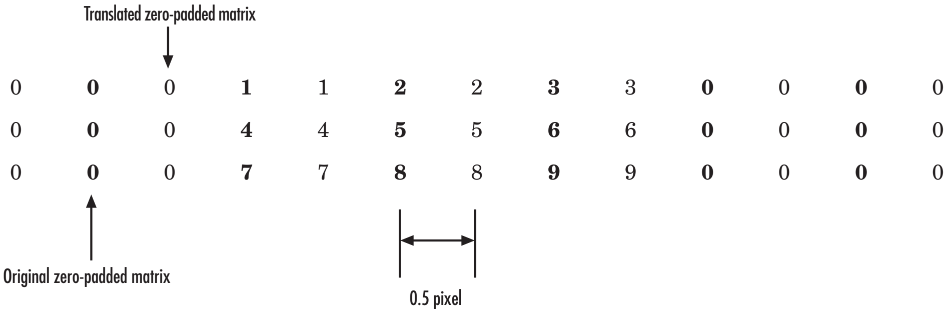
For bicubic interpolation, the block uses the weighted average of four translated pixel values for each output pixel value.

For example, suppose this matrix,

1	2	3
4	5	6
7	8	9

represents your input image. You want to translate this image 0.5 pixel in the positive horizontal direction using bicubic interpolation. The Translate block's bicubic interpolation algorithm is illustrated by the following steps:

1 Zero pad the input matrix and translate it by 0.5 pixel to the right.

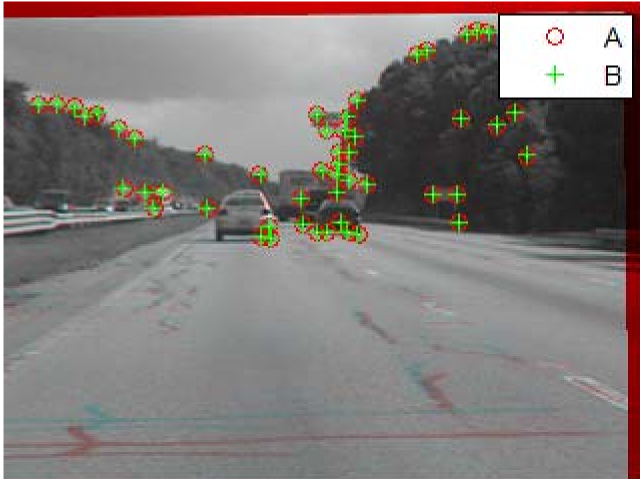


2 Create the output matrix by replacing each input pixel value with the weighted average of the two translated values on either side. The result is the following matrix where the output matrix has one more column than the input matrix:

0.375	1.5	3	1.625
1.875	4.875	6.375	3.125
3.375	8.25	9.75	4.625

Automatically Determine Geometric Transform for Image Registration

Stabilizing a video that was captured from a jittery or moving platform is an important application in computer vision. One way to stabilize a video is to track a salient feature in the image and use this as an anchor point to cancel out all perturbations relative to it. This procedure, however, must be bootstrapped with knowledge of where such a salient feature lies in the first video frame. In this example, we explore a method of video stabilization that works without any such a priori knowledge. It instead automatically searches for the "background plane" in a video sequence, and uses its observed distortion to correct for camera motion.



This stabilization algorithm involves two steps. First, we determine the affine image transformations between all neighboring frames of a video sequence using a Random Sampling and Consensus (RANSAC) [1] procedure applied to point correspondences between two images. Second, we warp the video frames to achieve a stabilized video. We use System objects in the Computer Vision System Toolbox™, both for the algorithm and for display.

You can launch this example, Video Stabilization Using Point Feature Matching directly, by typing `videostabilize_pm` on the MATLAB command line.

Filters, Transforms, and Enhancements

- "Adjust the Contrast of Intensity Images" on page 6-2
- "Adjust the Contrast of Color Images" on page 6-8
- "Remove Periodic Noise from a Video" on page 6-14
- "Remove Salt and Pepper Noise from Images" on page 6-23
- "Sharpen an Image" on page 6-30

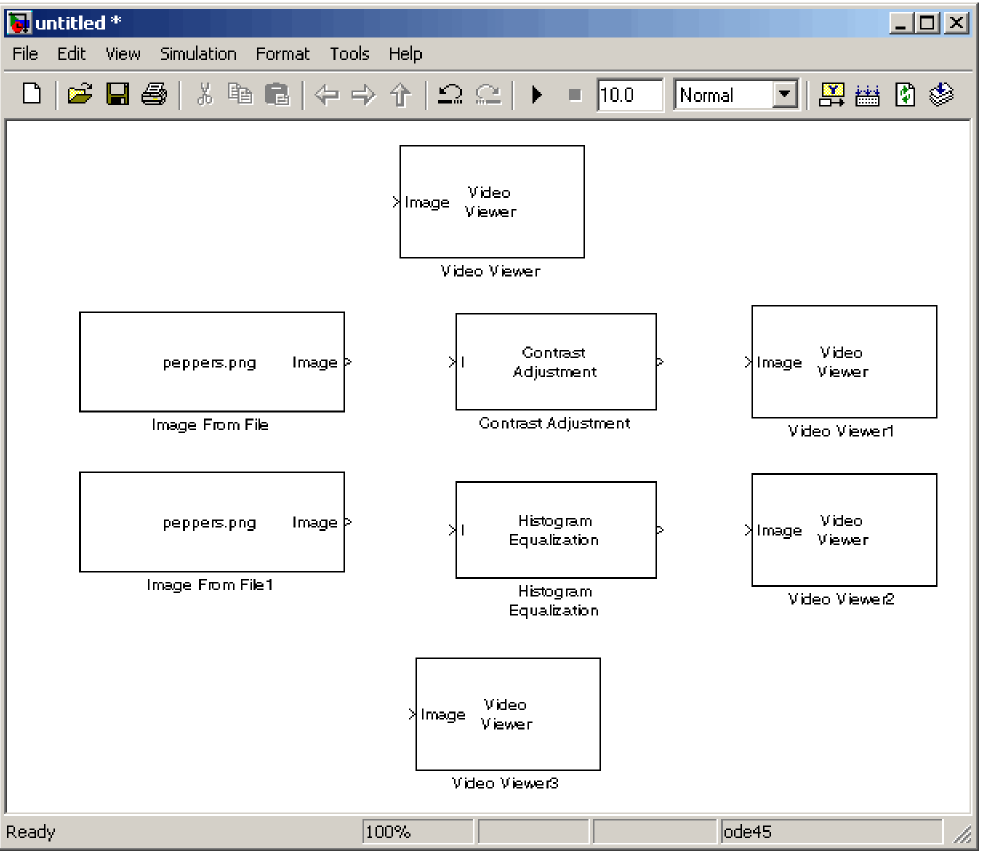
Adjust the Contrast of Intensity Images

This example shows you how to modify the contrast in two intensity images using the Contrast Adjustment and Histogram Equalization blocks.

1 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	2
Contrast Adjustment	Computer Vision System Toolbox > Analysis & Enhancement	1
Histogram Equalization	Computer Vision System Toolbox > Analysis & Enhancement	1
Video Viewer	Computer Vision System Toolbox > Sinks	4

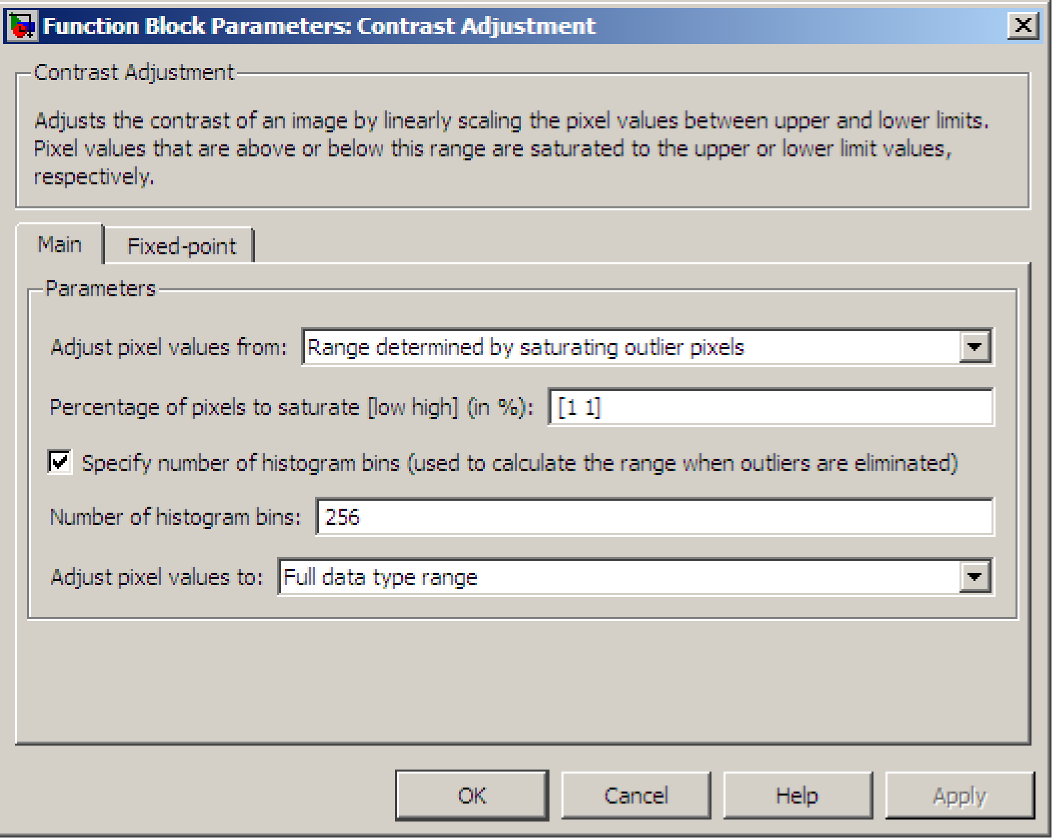
2 Place the blocks so that your model resembles the following figure.



3 Use the Image From File block to import the first image into the Simulink model. Set the **File name** parameter to `pout.tif`.

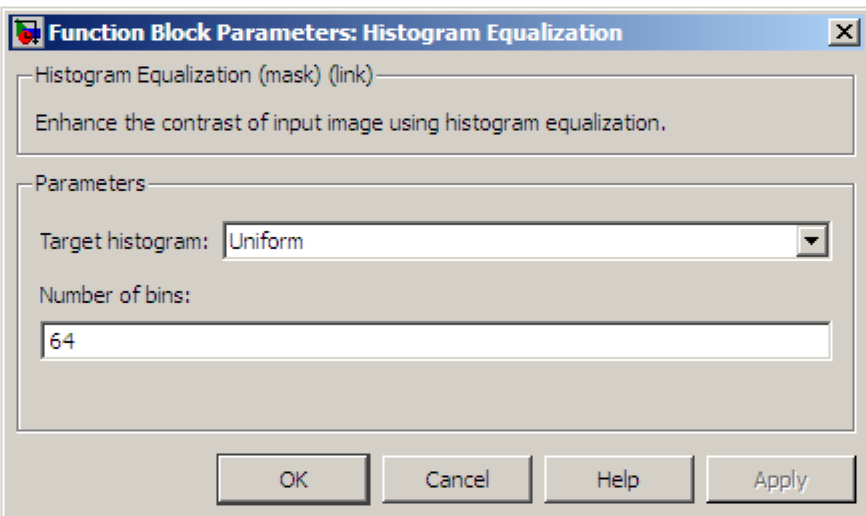
4 Use the Image From File1 block to import the second image into the Simulink model. Set the **File name** parameter to `tire.tif`.

5 Use the Contrast Adjustment block to modify the contrast in `pout.tif`. Set the **Adjust pixel values from** parameter to `Range determined by saturating outlier pixels`, as shown in the following figure.



This block adjusts the contrast of the image by linearly scaling the pixel values between user-specified upper and lower limits.

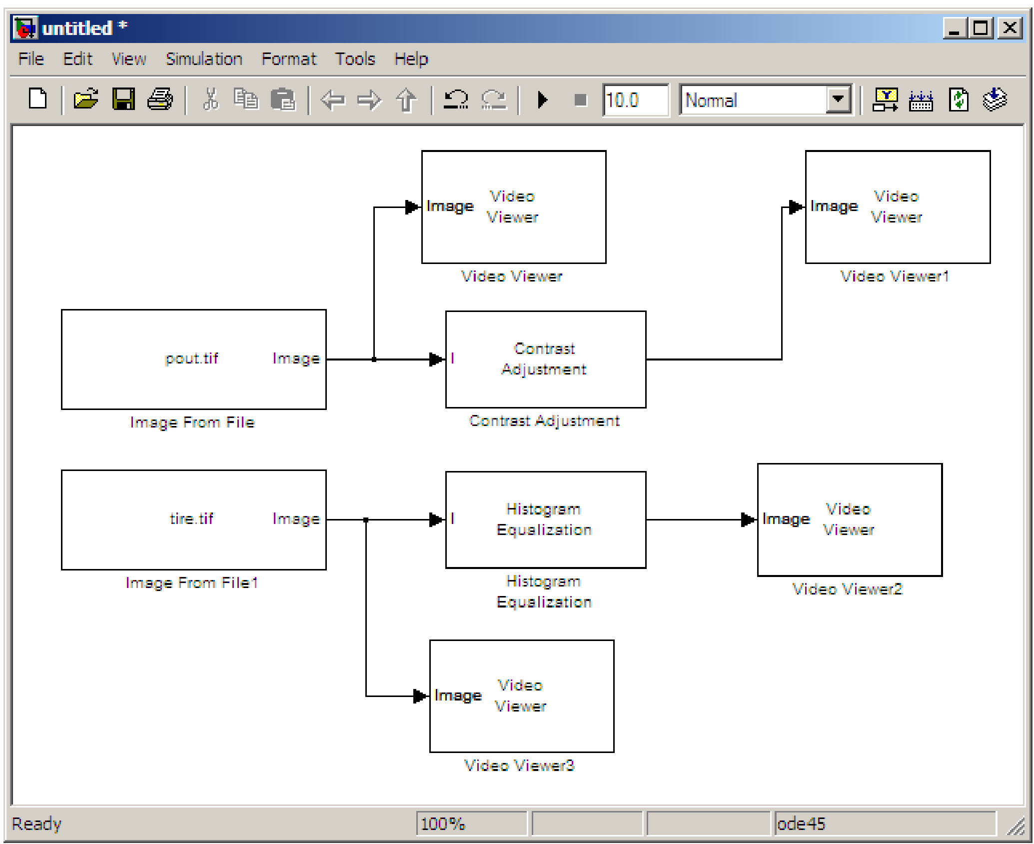
6 Use the Histogram Equalization block to modify the contrast in `tire.tif`. Accept the default parameters.



This 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.

7. Use the Video Viewer blocks to view the original and modified images. Accept the default parameters.

8. Connect the blocks as shown in the following figure.

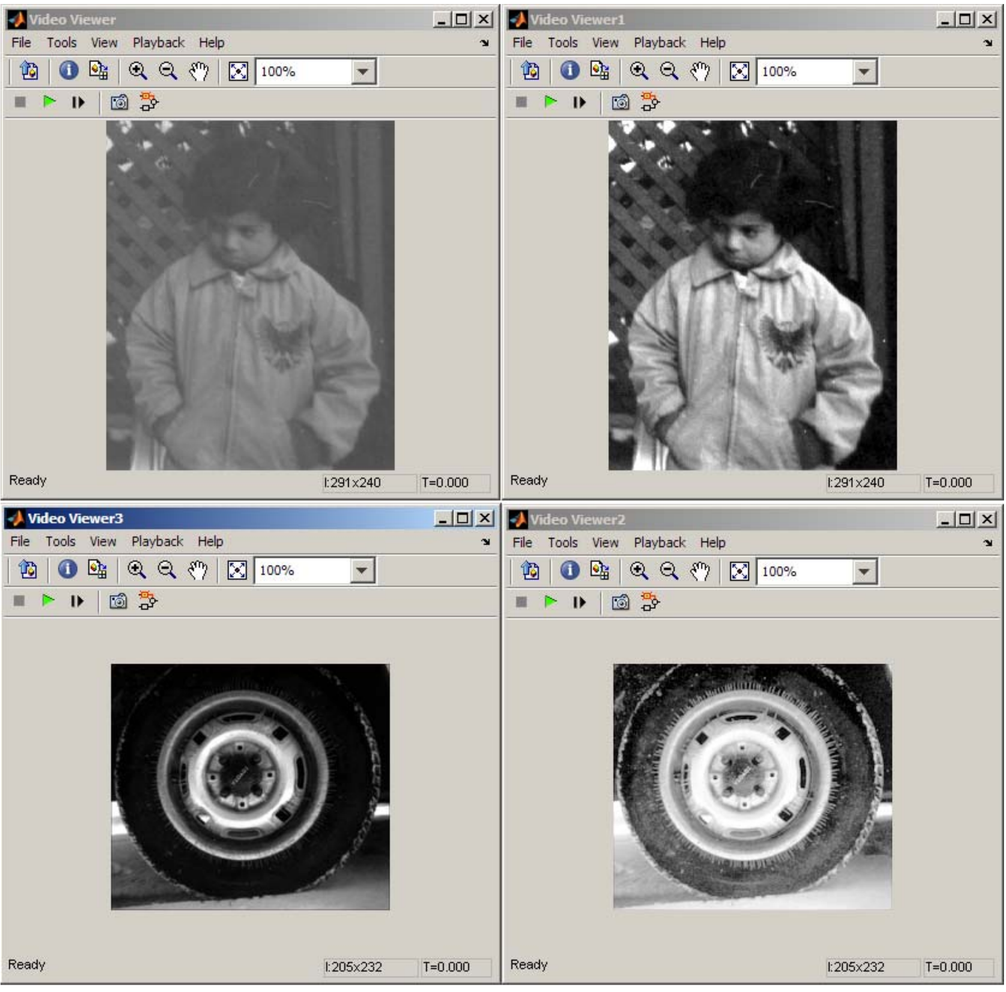


9 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

10 Run the model.

The results appear in the Video Viewer windows.



In this example, you used the Contrast Adjustment block to linearly scale the pixel values in `pout.tif` between new upper and lower limits. You used the Histogram Equalization block to transform the values in `tire.tif` so that the histogram of the output image approximately matches a uniform histogram. For more information, see the Contrast Adjustment and Histogram Equalization reference pages.

Adjust the Contrast of Color Images

This example shows you how to modify the contrast in color images using the Histogram Equalization block.

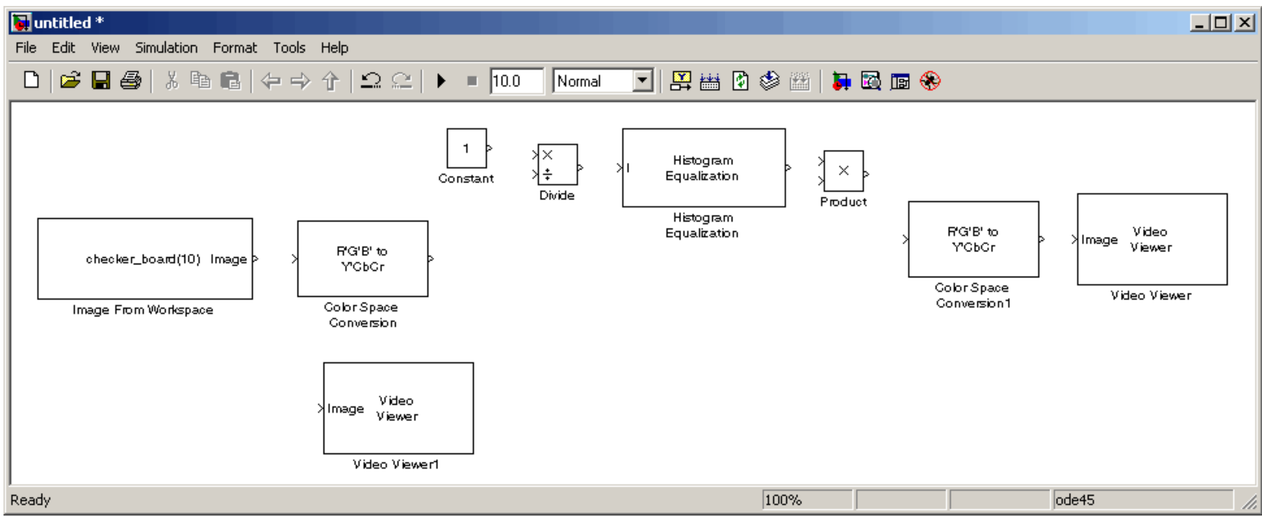
1 Use the following code to read in an indexed RGB image, `shadow.tif`, and convert it to an RGB image.

```
[X map] = imread('shadow.tif');
shadow = ind2rgb(X,map);
```

2 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From Workspace	Computer Vision System Toolbox > Sources	1
Color Space Conversion	Computer Vision System Toolbox > Conversions	2
Histogram Equalization	Computer Vision System Toolbox > Analysis & Enhancement	1
Video Viewer	Computer Vision System Toolbox > Sinks	2
Constant	Simulink > Sources	1
Divide	Simulink > Math Operations	1
Product	Simulink > Math Operations	1

3 Place the blocks so that your model resembles the following figure.



4 Use the Image From Workspace block to import the RGB image from the MATLAB workspace into the Simulink model. Set the block parameters as follows:

- **Value** = `shadow`
- **Image signal** = `Separate color signals`

5 Use the Color Space Conversion block to separate the luma information from the color information. Set the block parameters as follows:

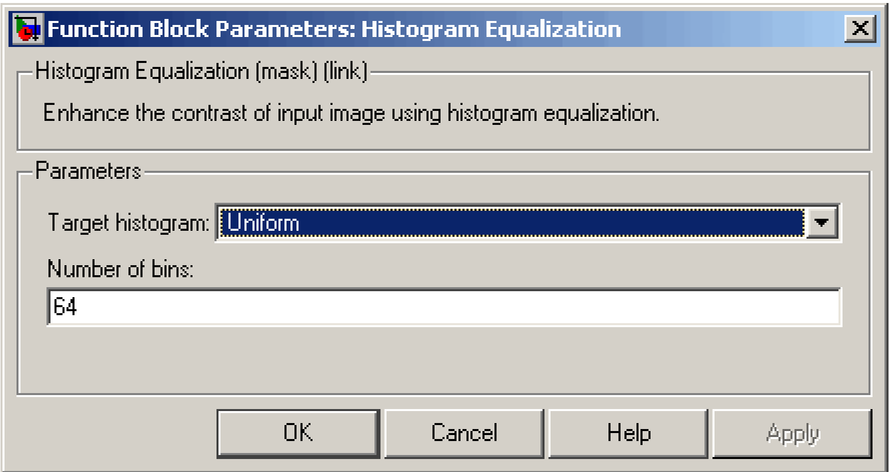
- **Conversion** = `sR'G'B' to L*a*b*`
- **Image signal** = `Separate color signals`

Because the range of the L* values is between `0` and `100`, you must normalize them to be between zero and one before you pass them to the Histogram Equalization block, which expects floating point input in this range.

6 Use the Constant block to define a normalization factor. Set the **Constant value** parameter to `100`.

7 Use the Divide block to normalize the L* values to be between `0` and `1`. Accept the default parameters.

8 Use the Histogram Equalization block to modify the contrast in the image. Accept the default parameters.



This block enhances the contrast of images by transforming the luma values in the color image so that the histogram of the output image approximately matches a specified histogram.

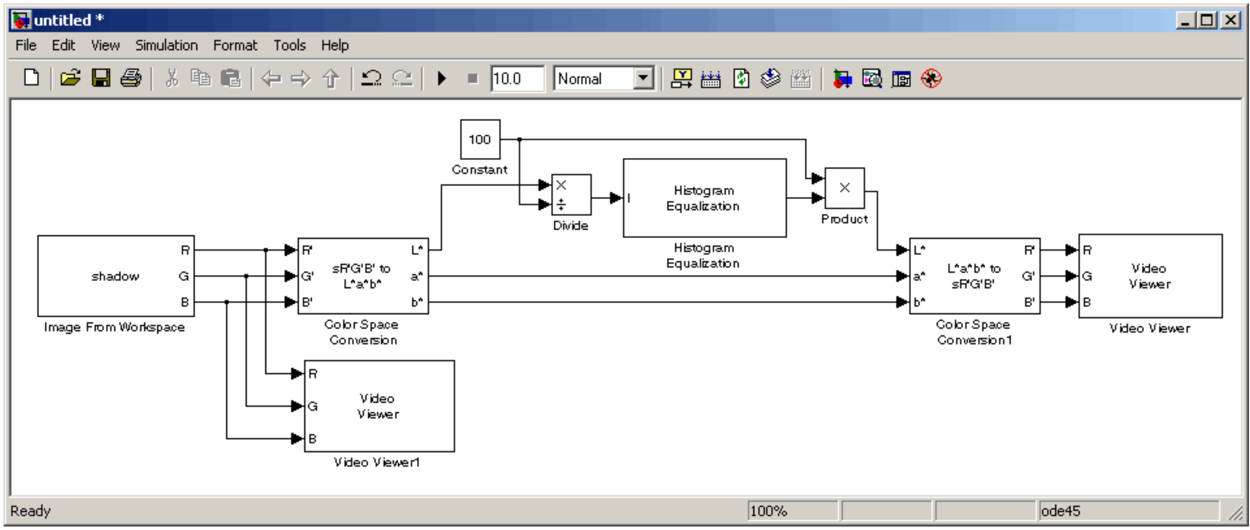
9 Use the Product block to scale the values back to be between the `0` to `100` range. Accept the default parameters.

10 Use the Color Space Conversion1 block to convert the values back to the sR'G'B' color space. Set the block parameters as follows:

- **Conversion** = `L*a*b* to sR'G'B'`
- **Image signal** = `Separate color signals`

11 Use the Video Viewer blocks to view the original and modified images. For each block, set the **Image signal** parameter to `Separate color signals` from the file menu.

12 Connect the blocks as shown in the following figure.

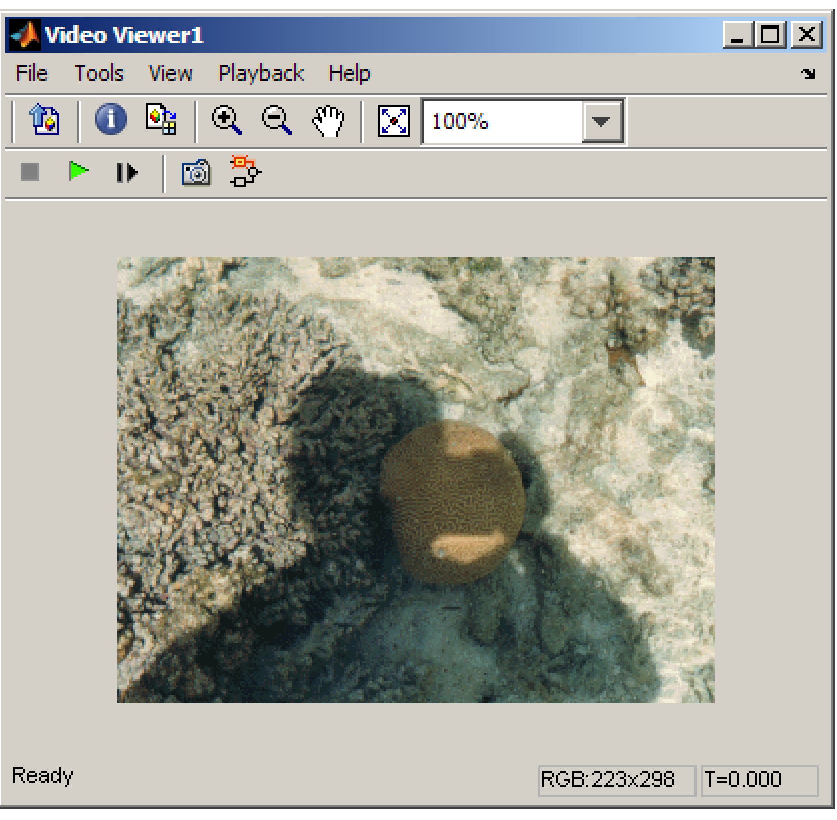


13 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

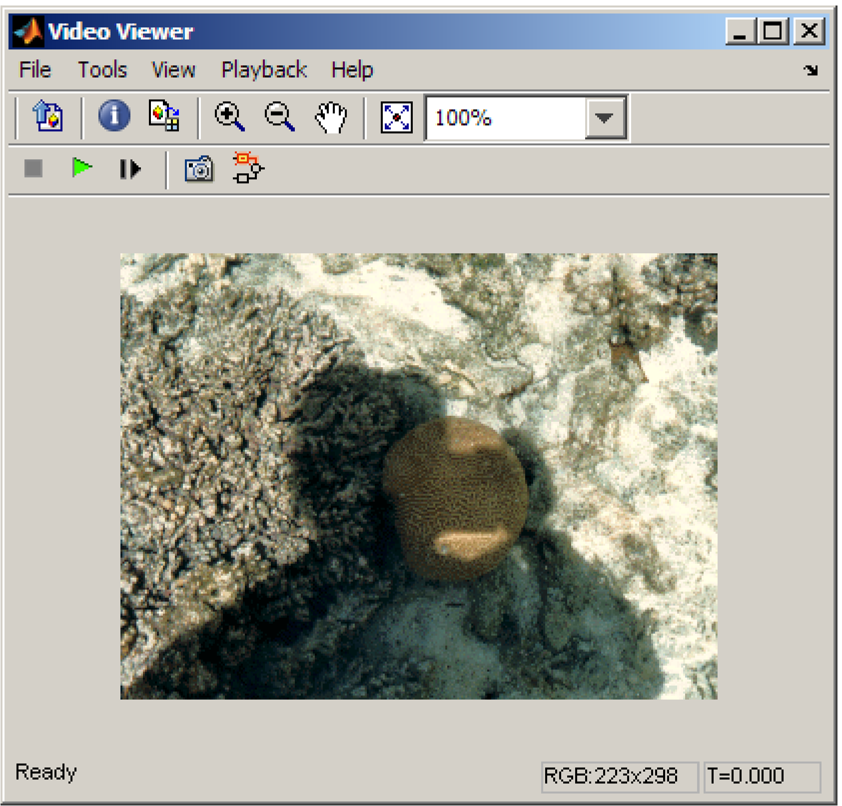
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

14 Run the model.

As shown in the following figure, the model displays the original image in the Video Viewer1 window.



As the next figure shows, the model displays the enhanced contrast image in the Video Viewer window.



In this example, you used the Histogram Equalization block to transform the values in a color image so that the histogram of the output image approximately matches a uniform histogram. For more information, see the Histogram Equalization reference page.

Remove Periodic Noise from a Video

Periodic noise can be introduced into a video stream during acquisition or transmission due to electrical or electromechanical interference. In this example, you remove periodic noise from an intensity video using the 2-D FIR Filter block. You can use this technique to remove noise from other images or video streams, but you might need to modify the filter coefficients to account for the noise frequency content present in your signal:

1 Create a Simulink model, and add the blocks shown in the following table.

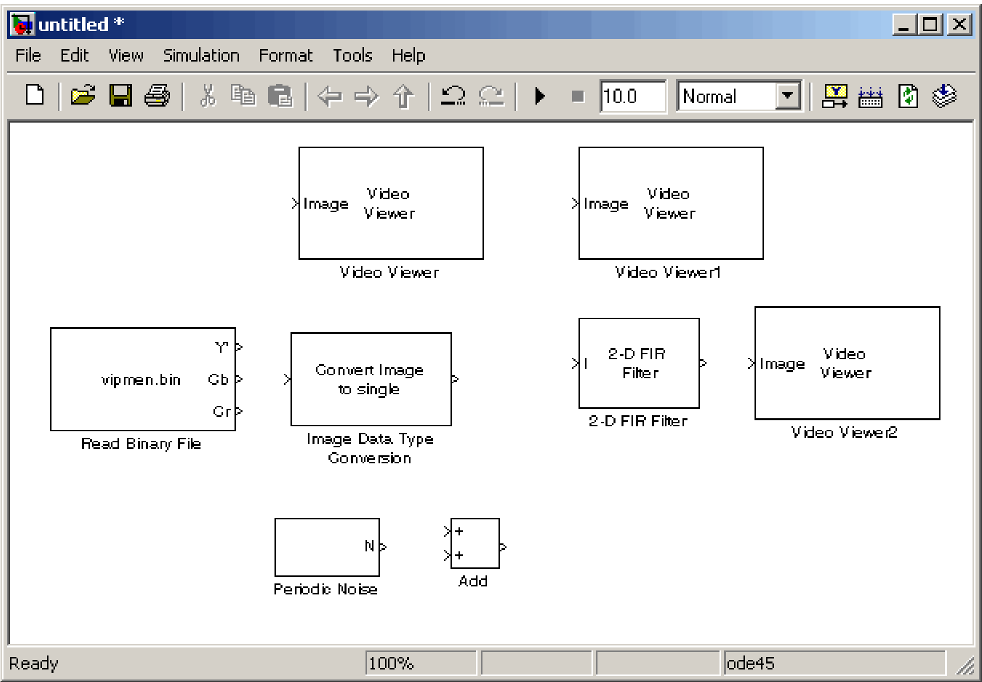
Block	Library	Quantity
Read Binary File	Computer Vision System Toolbox > Sources	1
Image Data Type Conversion	Computer Vision System Toolbox > Conversions	1
2-D FIR Filter	Computer Vision System Toolbox > Filtering	1
Video Viewer	Computer Vision System Toolbox > Sinks	3
Add	Simulink > Math Operations	1

2 Open the Periodic noise reduction demo by typing `vipstripes` at the MATLAB command prompt.

3 Click-and-drag the Periodic Noise block into your model.

   The block outputs a sinusoid with a normalized frequency that ranges between 0.61π and 0.69π radians per sample and a phase that varies between zero and three radians. You are using this sinusoid to represent periodic noise.

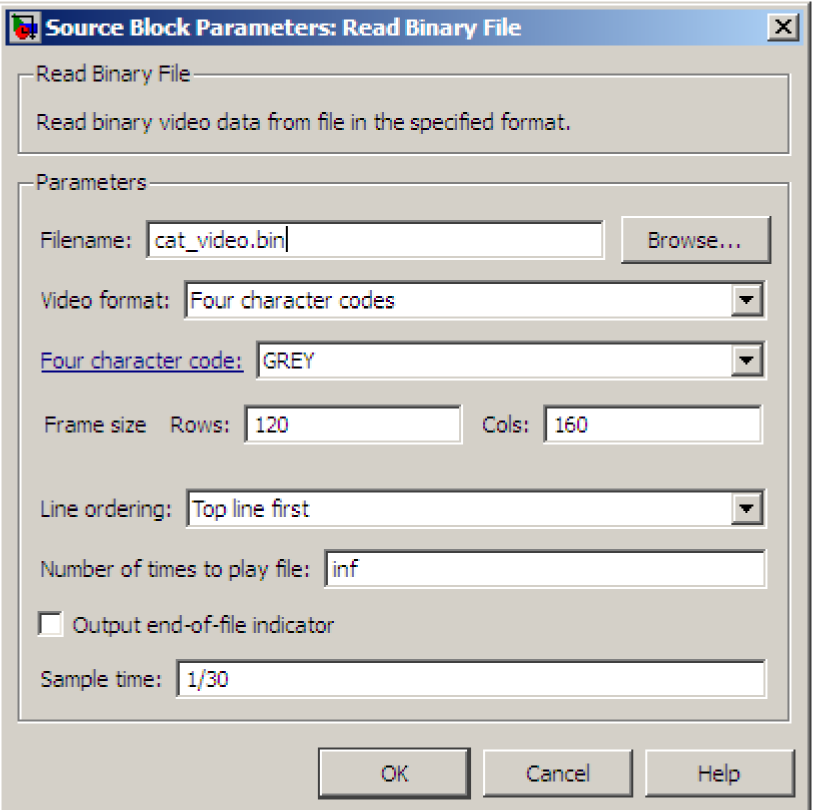
4 Place the blocks so that your model resembles the following figure. The unconnected ports disappear when you set block parameters.



You are now ready to set your block parameters by double-clicking the blocks, modifying the block parameter values, and clicking **OK**.

5 Use the Read Binary File block to import a binary file into the model. Set the block parameters as follows:

- **File name** = `cat_video.bin`
- **Four character code** = `GREY`
- **Number of times to play file** = `inf`
- **Sample time** = `1/30`



6. Use the Image Data Type Conversion block to convert the data type of the video to single-precision floating point. Accept the default parameter.

7. Use the Video Viewer block to view the original video. Accept the default parameters.

8. Use the Add block to add the noise video to the original video. Accept the default parameters.

9. Use the Video Viewer1 block to view the noisy video. Accept the default parameters.

10. Define the filter coefficients in the MATLAB workspace. Type the following code at the MATLAB command prompt:

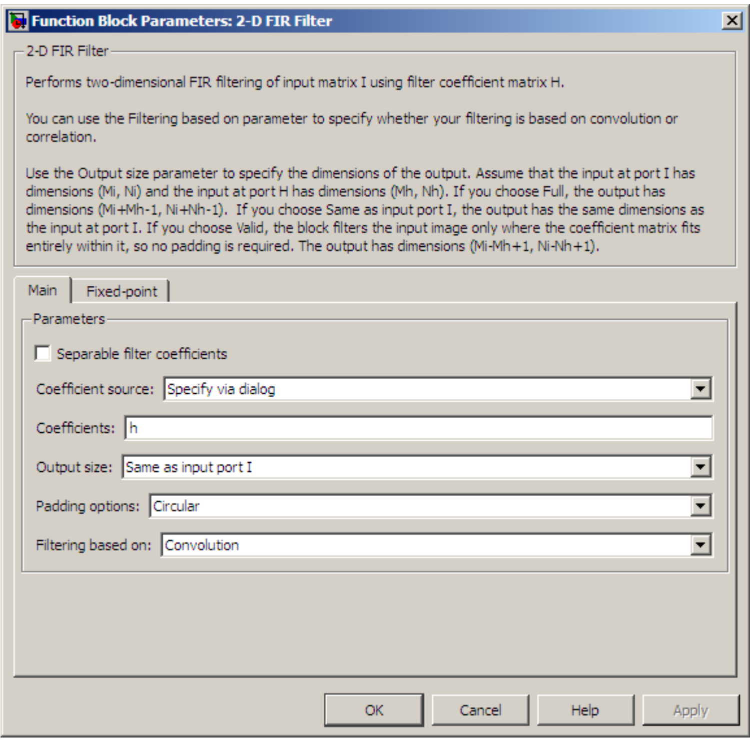
```
vipdh_stripes
```

The variable `h`, as well as several others, are loaded into the MATLAB workspace. The variable `h` represents the coefficients of the band reject filter capable of removing normalized frequencies between 0.61π and 0.69π radians per sample. The coefficients were created using the Filter Design and Analysis Tool (FDATool) and the `ftrans2` function.

11 Use the 2-D FIR Filter block to model a band-reject filter capable of removing the periodic noise from the video. Set the block parameters as follows:

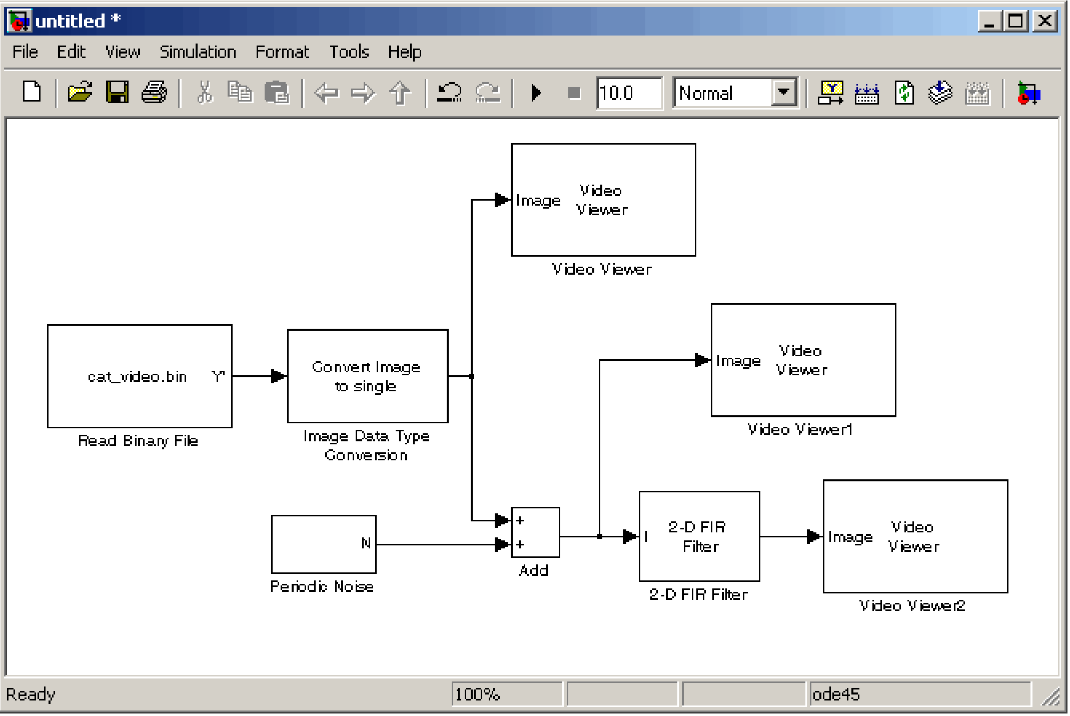
- **Coefficients** = `h`
- **Output size** = `Same as input port I`
- **Padding options** = `Circular`

Choose a type of padding that minimizes the effect of the pixels outside the image on the processing of the image. In this example, circular padding produces the best results because it is most effective at replicating the sinusoidal noise outside the image.



12 Use the Video Viewer2 block to view the approximation of the original video. Accept the default parameters.

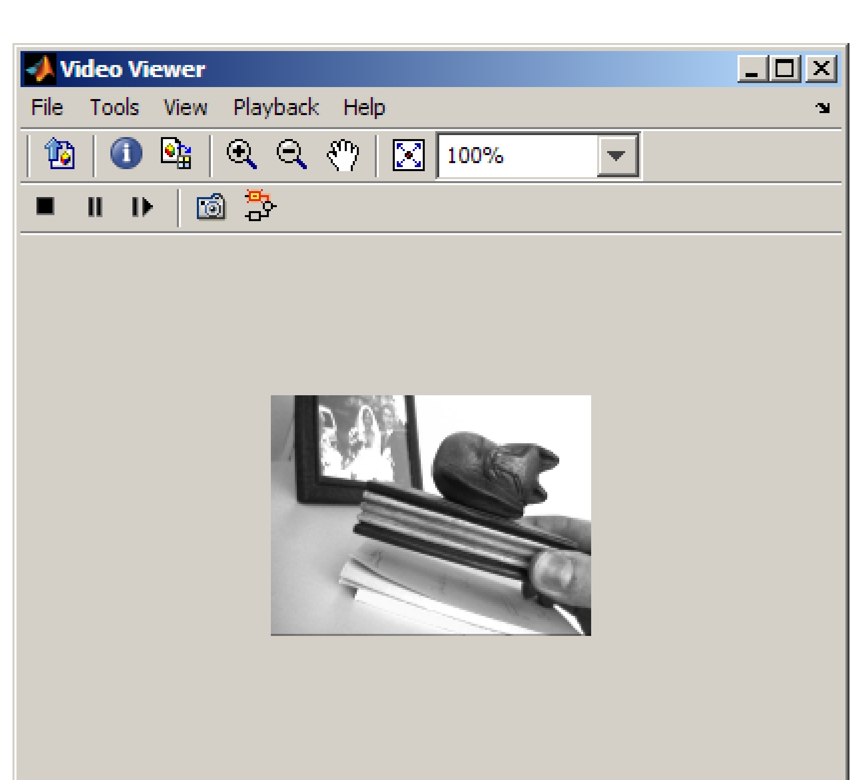
13 Connect the block as shown in the following figure.



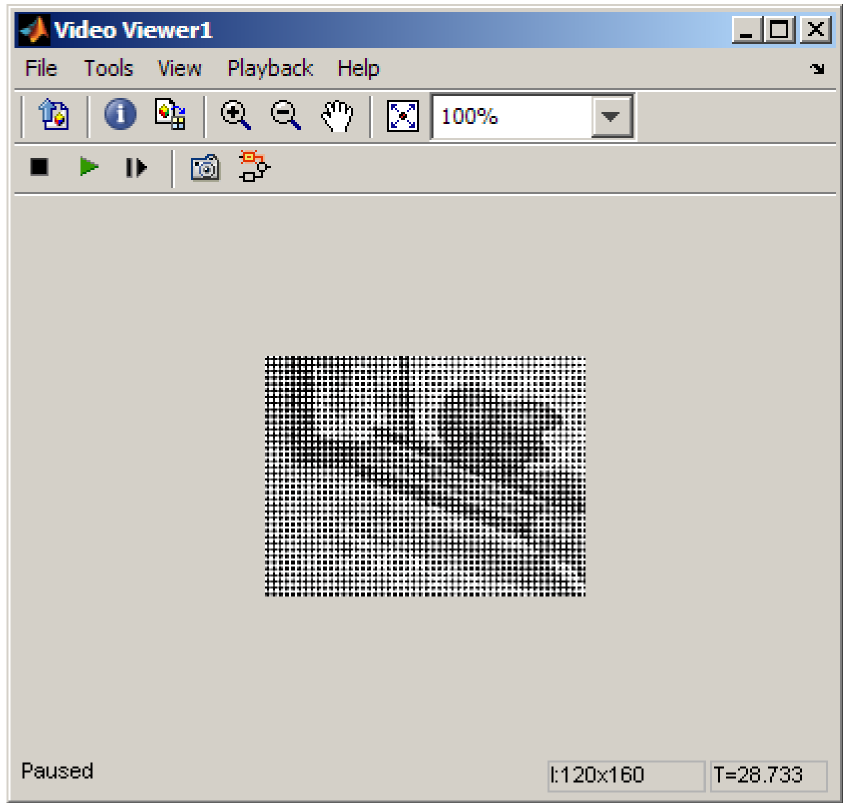
14 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `inf`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

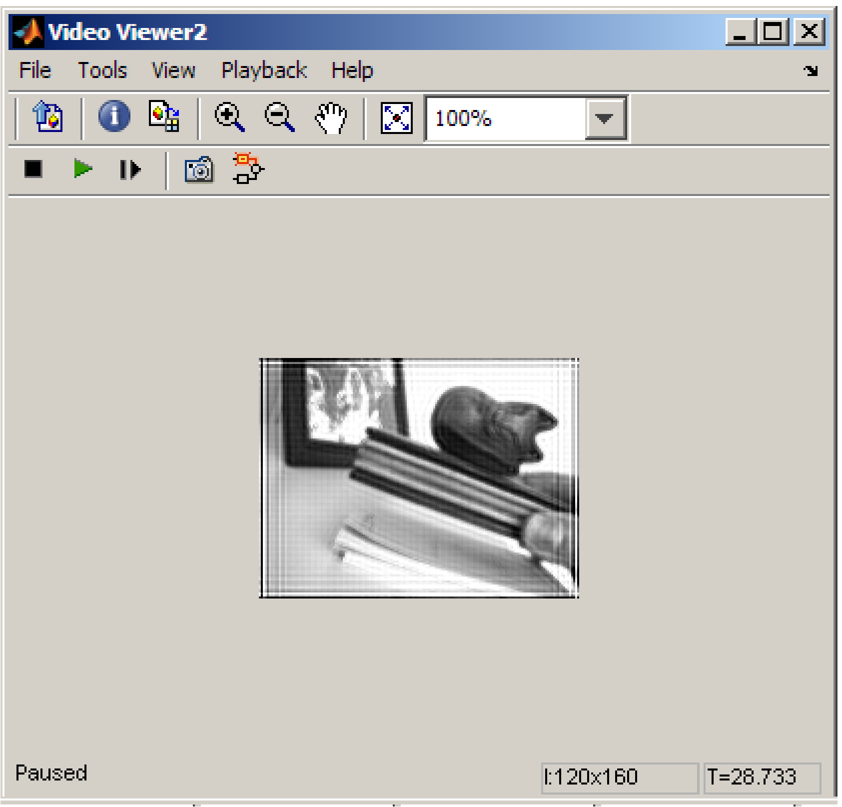
15 Run the model.



The noisy video appears in the Video Viewer1 window. The following video is shown at its true size.



The approximation of the original video appears in the Video Viewer2 window, and the artifacts of the processing appear near the edges of the video. The following video is shown at its true size.



You have used the Read Binary File block to import a binary video into your model, the 2-D FIR Filter to remove periodic noise from this video, and the Video Viewer block to display the results. For more information about these blocks, see the Read Binary File, 2-D FIR Filter, and Video Viewer block reference pages. For more information about the Filter Design and Analysis Tool (FDATool), see the Signal Processing Toolbox documentation. For information about the `ftrans2` function, see the Image Processing Toolbox documentation.

Remove Salt and Pepper Noise from Images

Median filtering is a common image enhancement technique for removing salt and pepper noise. Because this filtering is less sensitive than linear techniques to extreme changes in pixel values, it can remove salt and pepper noise without significantly reducing the sharpness of an image. In this topic, you use the Median Filter block to remove salt and pepper noise from an intensity image:

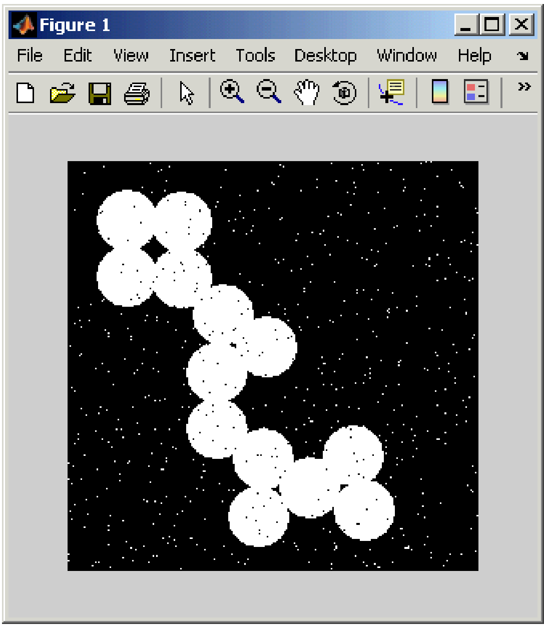
1 Define an intensity image in the MATLAB workspace and add noise to it by typing the following at the MATLAB command prompt:

```
I= double(imread('circles.png'));
I= imnoise(I,'salt & pepper',0.02);
```

`I` is a 256-by-256 matrix of 8-bit unsigned integer values.

2 To view the image this matrix represents, at the MATLAB command prompt, type

```
imshow(I)
```

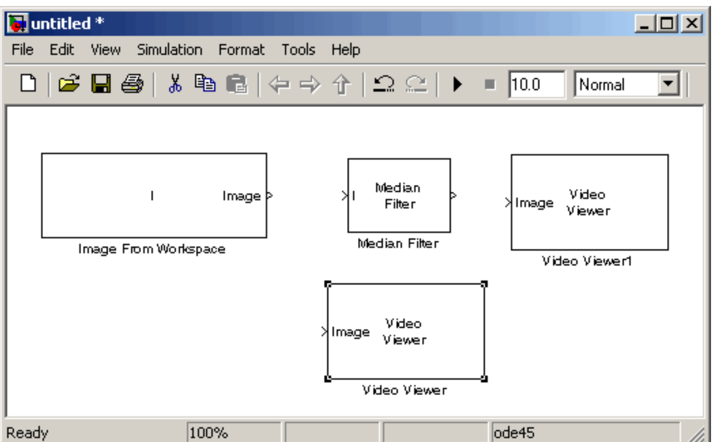


The intensity image contains noise that you want your model to eliminate.

3 Create a Simulink model, and add the blocks shown in the following table.

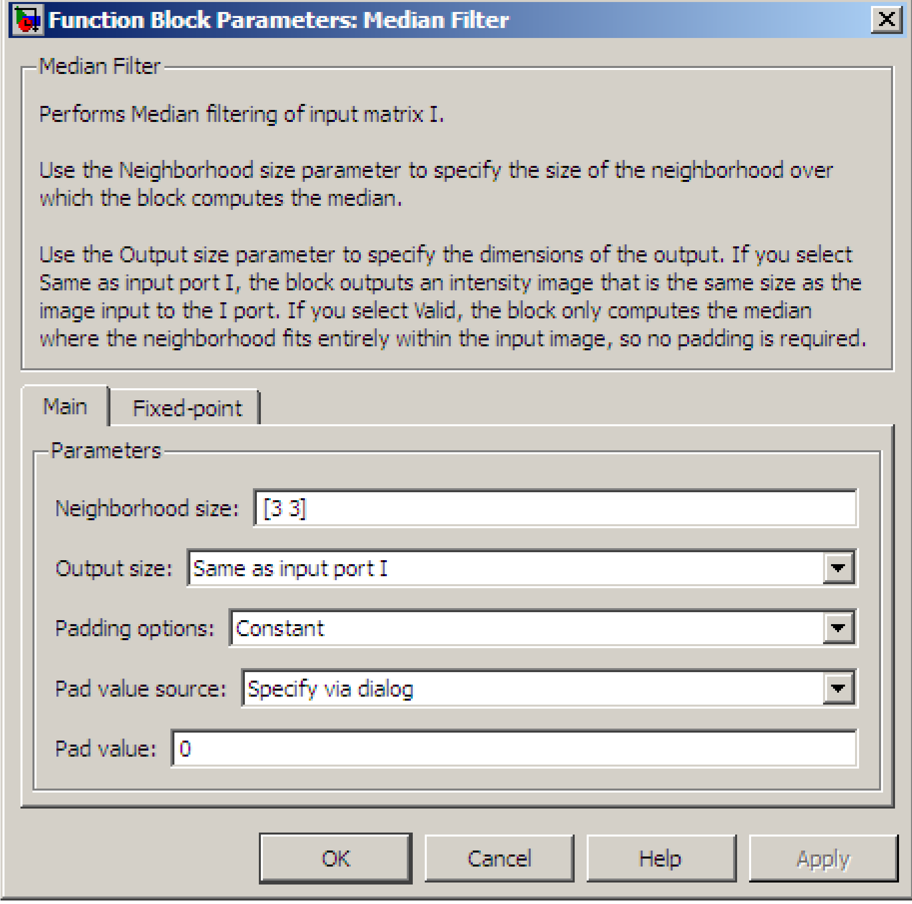
Block	Library	Quantity
Image From Workspace	Computer Vision System Toolbox > Sources	1
Median Filter	Computer Vision System Toolbox > Filtering	1
Video Viewer	Computer Vision System Toolbox > Sinks	2

4 Position the blocks as shown in the following figure.



5 Use the Image From Workspace block to import the noisy image into your model. Set the **Value** parameter to `I`.

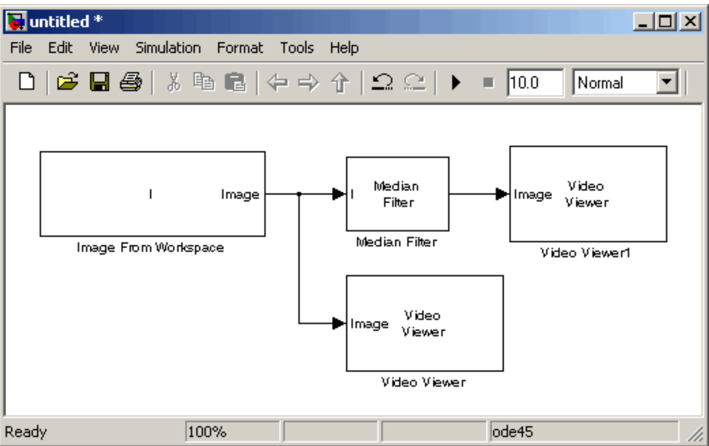
6 Use the Median Filter block to eliminate the black and white speckles in the image. Use the default parameters.



The Median Filter block replaces the central value of the 3-by-3 neighborhood with the median value of the neighborhood. This process removes the noise in the image.

7 Use the Video Viewer blocks to display the original noisy image, and the modified image. Images are represented by 8-bit unsigned integers. Therefore, a value of 0 corresponds to black and a value of 255 corresponds to white. Accept the default parameters.

8 Connect the blocks as shown in the following figure.

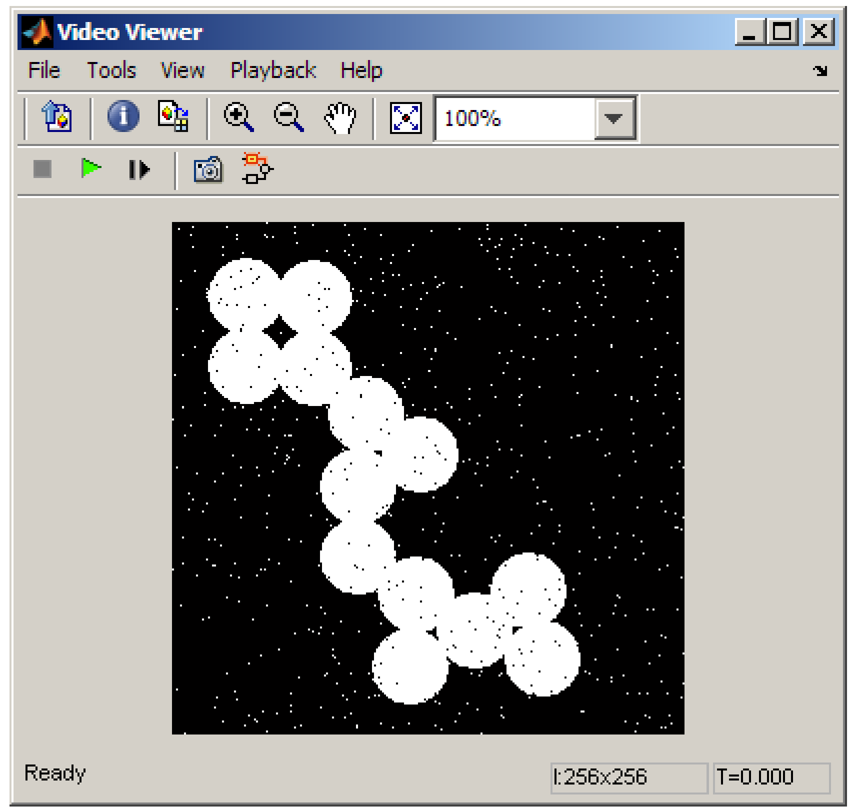


9 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

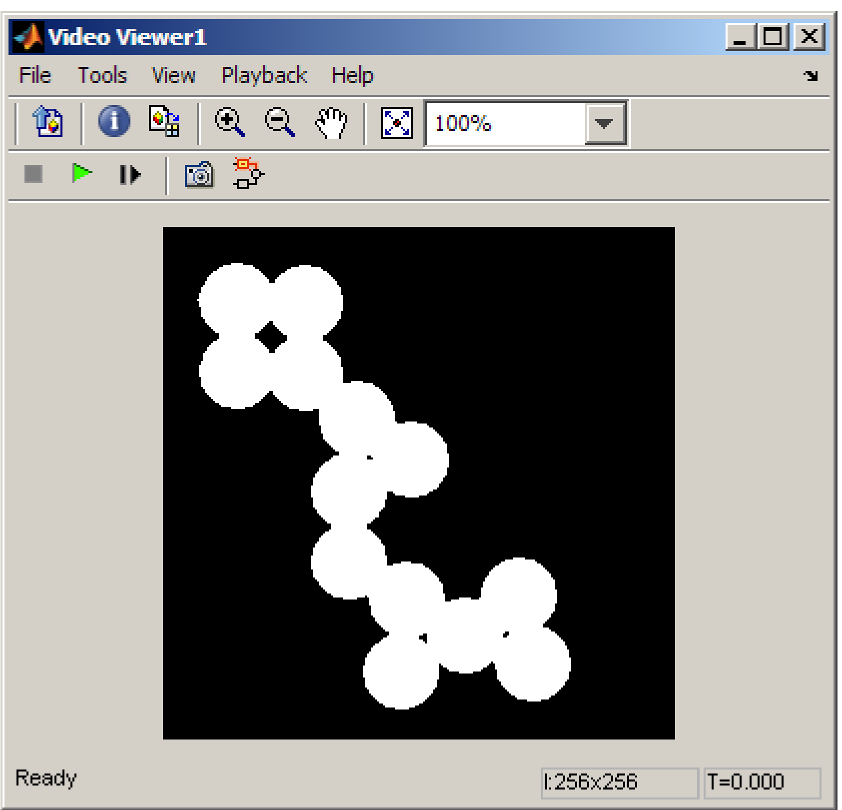
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

10 Run the model.

The original noisy image appears in the Video Viewer window. To view the image at its true size, right-click the window and select **Set Display To True Size**.



The cleaner image appears in the Video Viewer1 window. The following image is shown at its true size.



You have used the Median Filter block to remove noise from your image. For more information about this block, see the Median Filter block reference page in the *Computer Vision System Toolbox Reference*.

Sharpen an Image

To sharpen a color image, you need to make the luma intensity transitions more acute, while preserving the color information of the image. To do this, you convert an R'G'B' image into the Y'CbCr color space and apply a highpass filter to the luma portion of the image only. Then, you transform the image back to the R'G'B' color space to view the results. To blur an image, you apply a lowpass filter to the luma portion of the image. This example shows how to use the 2-D FIR Filter block to sharpen an image. The prime notation indicates that the signals are gamma corrected.

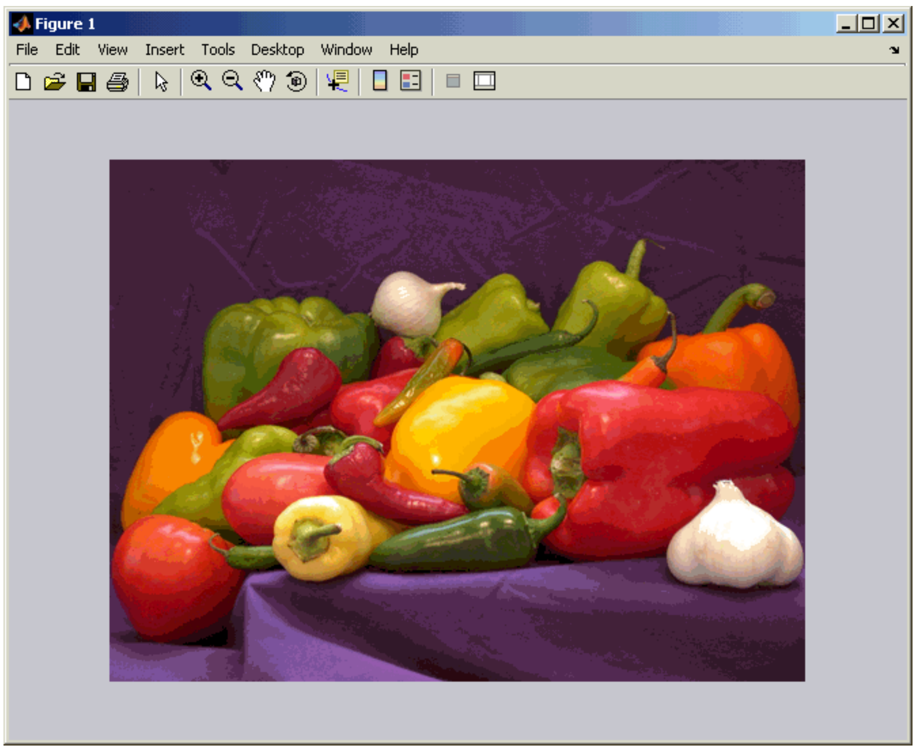
1. Define an R'G'B' image in the MATLAB workspace. To read in an R'G'B' image from a PNG file and cast it to the double-precision data type, at the MATLAB command prompt, type

   ```
   I= im2double(imread('peppers.png'));
   ```

   `I` is a 384-by-512-by-3 array of double-precision floating-point values. Each plane of this array represents the red, green, or blue color values of the image.

2. To view the image this array represents, at the MATLAB command prompt, type

   ```
   imshow(I)
   ```



Now that you have defined your image, you can create your model.

3 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From Workspace	Computer Vision System Toolbox > Sources	1
Color Space Conversion	Computer Vision System Toolbox > Conversions	2
2-D FIR Filter	Computer Vision System Toolbox > Filtering	1
Video Viewer	Computer Vision System Toolbox > Sinks	1

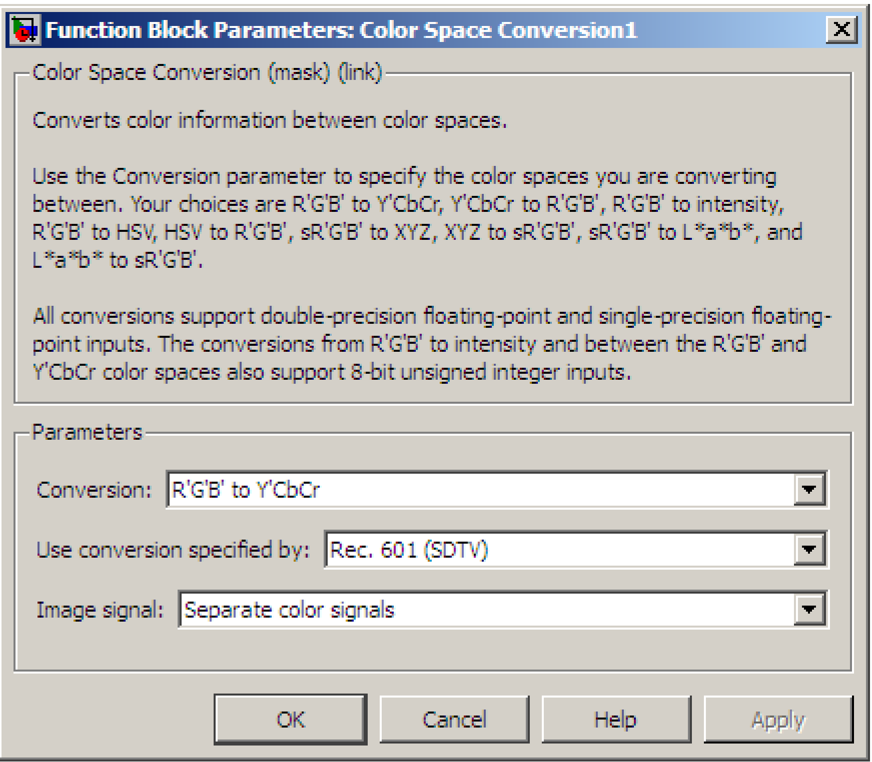
4 Position the blocks as shown in the following figure.

5 Use the Image From Workspace block to import the R'G'B' image from the MATLAB workspace. Set the parameters as follows:

- **Main** pane, **Value** = `I`
- **Main** pane, **Image signal** = `Separate color signals`

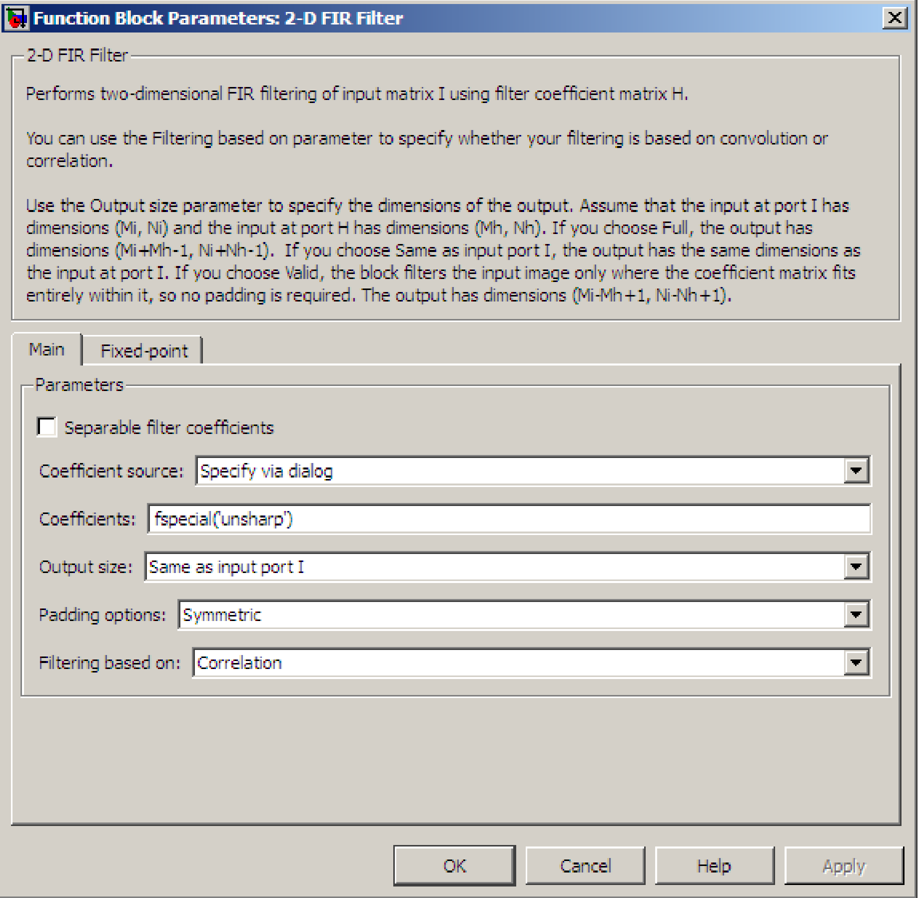
The block outputs the R', G', and B' planes of the `I` array at the output ports.

6 The first Color Space Conversion block converts color information from the R'G'B' color space to the Y'CbCr color space. Set the **Image signal** parameter to `Separate color signals`



7 Use the 2-D FIR Filter block to filter the luma portion of the image. Set the block parameters as follows:

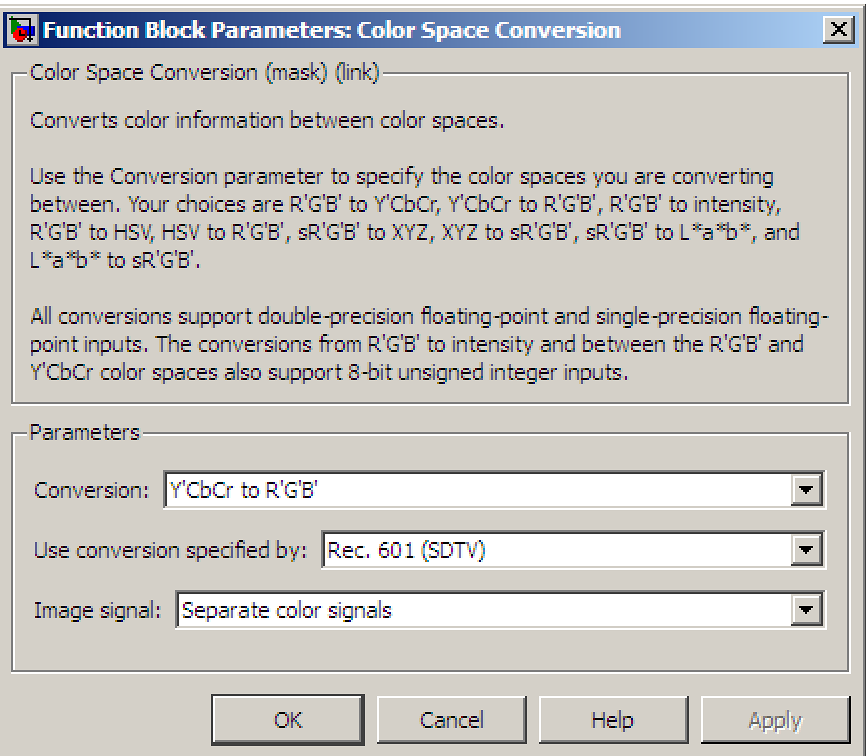
- **Coefficients** = `fspecial('unsharp')`
- **Output size** = `Same as input port I`
- **Padding options** = `Symmetric`
- **Filtering based on** = `Correlation`



The `fspecial('unsharp')` command creates two-dimensional highpass filter coefficients suitable for correlation. This highpass filter sharpens the image by removing the low frequency noise in it.

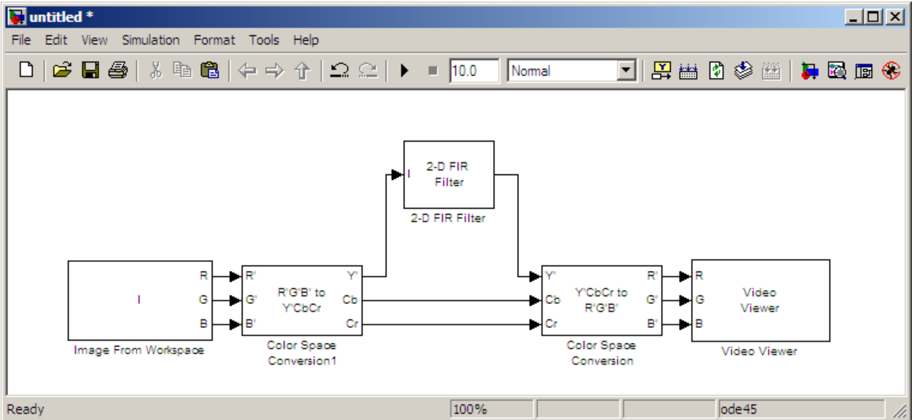
8 The second Color Space Conversion block converts the color information from the Y'CbCr color space to the R'G'B' color space. Set the block parameters as follows:

- **Conversion** = `Y'CbCr to R'G'B'`
- **Image signal** = `Separate color signals`



9 Use the Video Viewer block to automatically display the new, sharper image in the Video Viewer window when you run the model. Set the **Image signal** parameter to `Separate color signals`, by selecting **File > Image Signal**.

10 Connect the blocks as shown in the following figure.

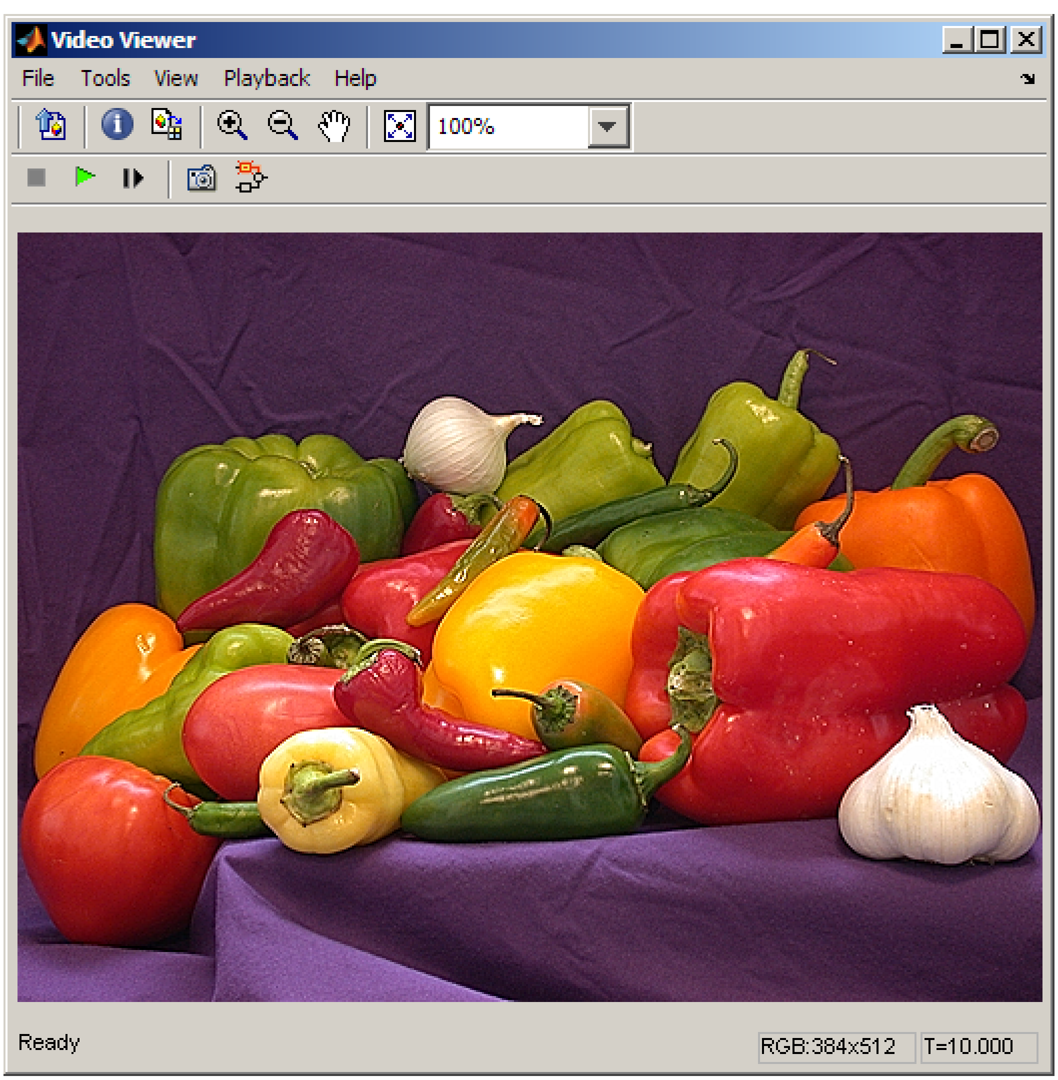


11 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

12 Run the model.

A sharper version of the original image appears in the Video Viewer window.



To blur the image, double-click the 2-D FIR Filter block. Set **Coefficients** parameter to `fspecial('gaussian',[15 15],7)` and then click **OK**. The `fspecial('gaussian',[15 15],7)` command creates two-dimensional Gaussian lowpass filter coefficients. This lowpass filter blurs the image by removing the high frequency noise in it.

In this example, you used the Color Space Conversion and 2-D FIR Filter blocks to sharpen an image. For more information, see the Color Space Conversion and 2-D FIR Filter, and `fspecial` reference pages.

7

Statistics and Morphological Operations

- "Find the Histogram of an Image" on page 7-2
- "Correct Nonuniform Illumination" on page 7-9
- "Count Objects in an Image" on page 7-17

Find the Histogram of an Image

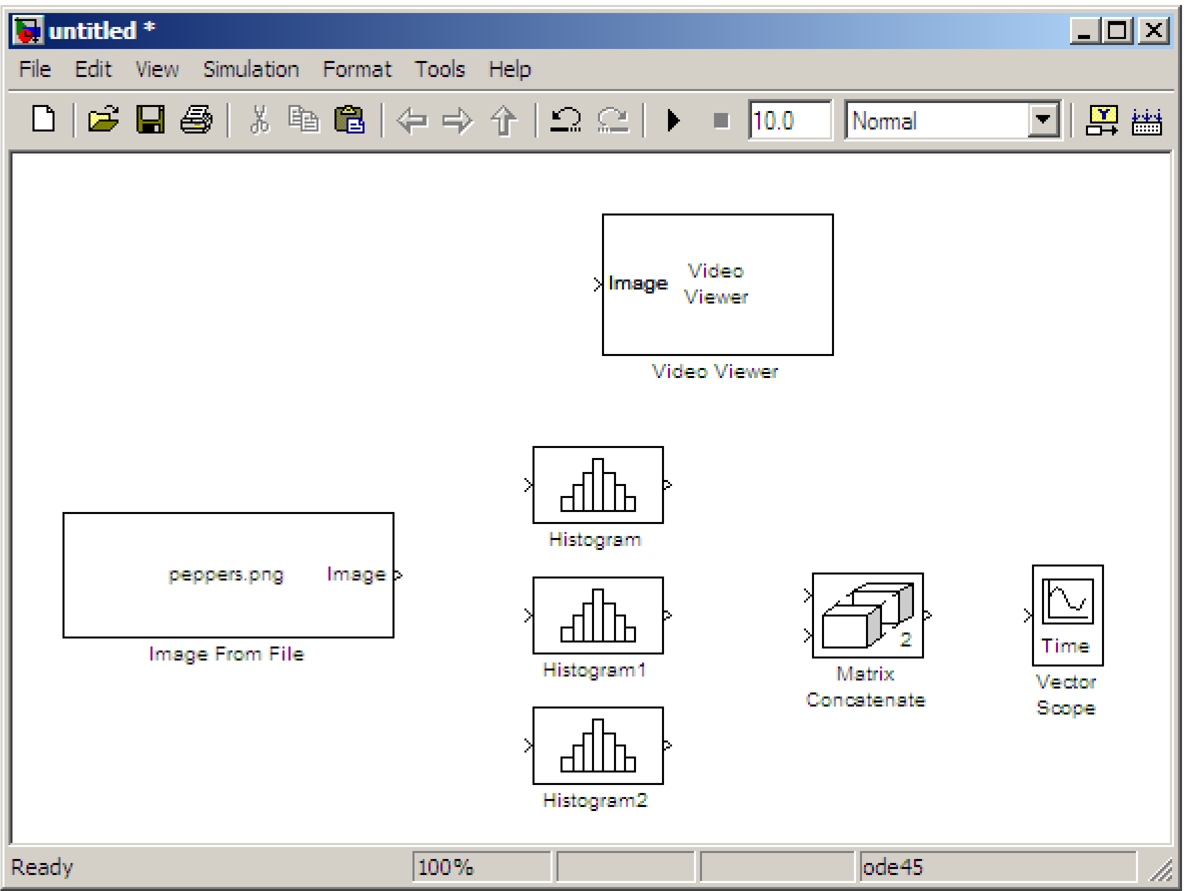
The Histogram block computes the frequency distribution of the elements in each input image by sorting the elements into a specified number of discrete bins. You can use the Histogram block to calculate the histogram of the R, G, and/or B values in an image. This example shows you how to accomplish this task:

Running this example requires a DSP System Toolbox license. However, you can easily replace the DSP System Toolbox Histogram block with the Computer Vision System Toolbox 2-D Histogram block.

1 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Video Viewer	Computer Vision System Toolbox > Sinks	1
Matrix Concatenate	Simulink > Math Operations	1
Vector Scope	DSP System Toolbox > Sinks	1
Histogram	DSP System Toolbox > Statistics	3

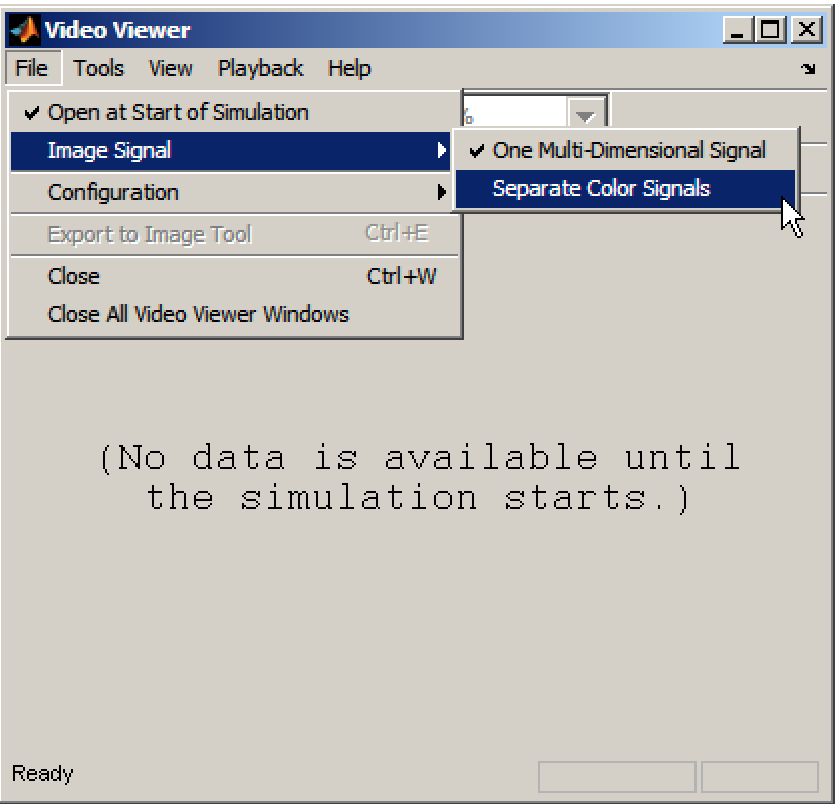
2 Place the blocks so that your model resembles the following figure.



3 Use the Image From File block to import an RGB image. Set the block parameters as follows:

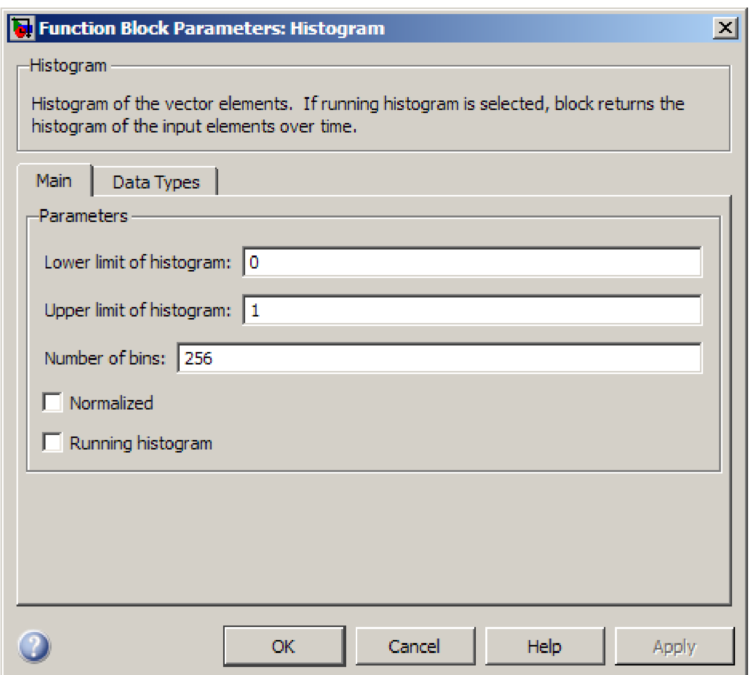
- **Sample time** = `inf`
- **Image signal** = `Separate color signals`
- **Output port labels:** = `R|G|B`
- **Output data type:** = `double`

4 Use the Video Viewer block to automatically display the original image in the viewer window when you run the model. Set the **Image signal** parameter to `Separate color signals`.



5 Use the Histogram blocks to calculate the histogram of the R, G, and B values in the image. Set the Main tab block parameters for the three Histogram blocks as follows:

- **Lower limit of histogram:** 0
- **Upper limit of histogram:** 1
- **Number of bins:** = 256



The **R**, **G**, and **B** input values to the Histogram block are double-precision floating point and range between `0` and `1`. The block creates 256 bins between the maximum and minimum input values and counts the number of R, G, and B values in each bin.

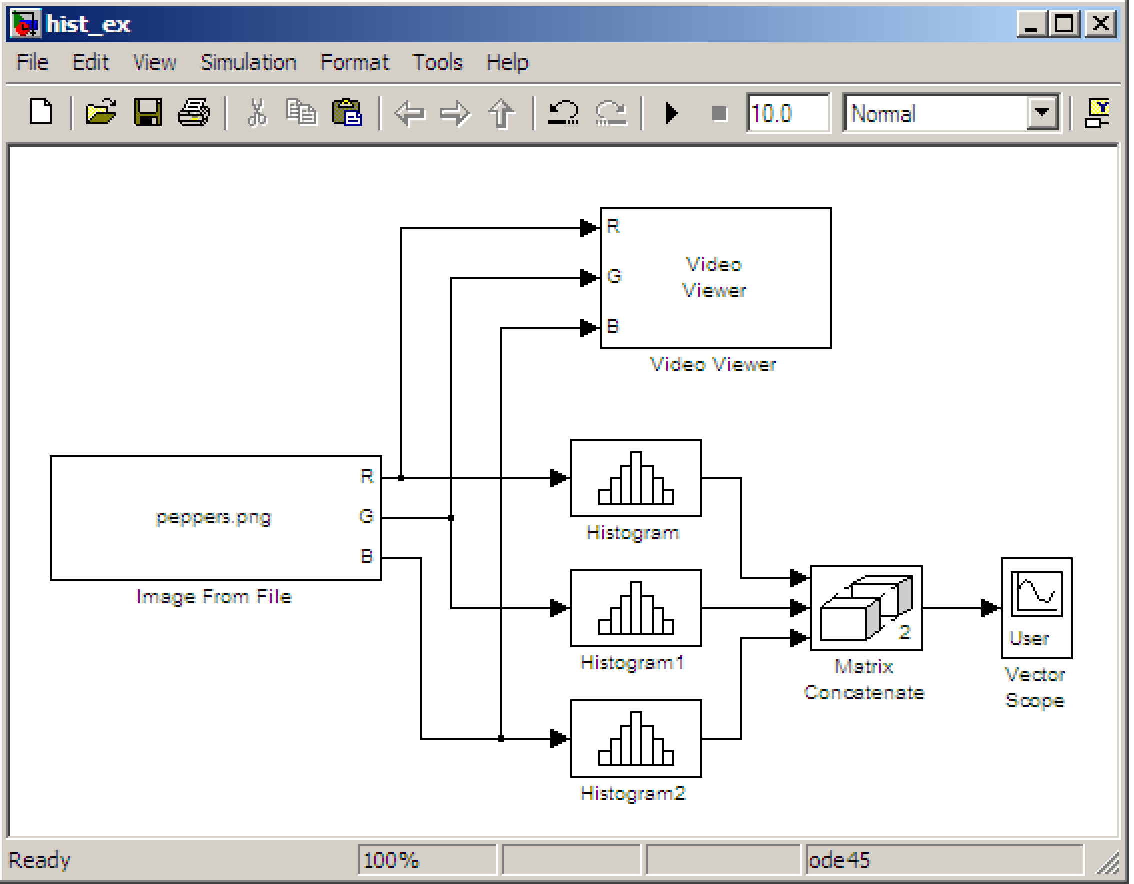
6 Use the Matrix Concatenate block to concatenate the R, G, and B column vectors into a single matrix so they can be displayed using the Vector Scope block. Set the **Number of inputs** parameter to `3`.

7 Use the Vector Scope block to display the histograms of the R, G, and B values of the input image. Set the block parameters as follows:

- **Scope Properties** pane, **Input domain** = `User-defined`
- **Display Properties** pane, clear the **Frame number** check box
- **Display Properties** pane, select the **Channel legend** check box
- **Display Properties** pane, select the **Compact display** check box

- **Axis Properties** pane, clear the **Inherit sample increment from input** check box.
- **Axis Properties** pane, **Minimum Y-limit** = `0`
- **Axis Properties** pane, **Maximum Y-limit** = `1`
- **Axis Properties** pane, **Y-axis label** = `Count`
- **Line Properties** pane, **Line markers** = `.|s|d`
- **Line Properties** pane, **Line colors** = `[1 0 0]|[0 1 0]|[0 0 1]`

8 Connect the blocks as shown in the following figure.

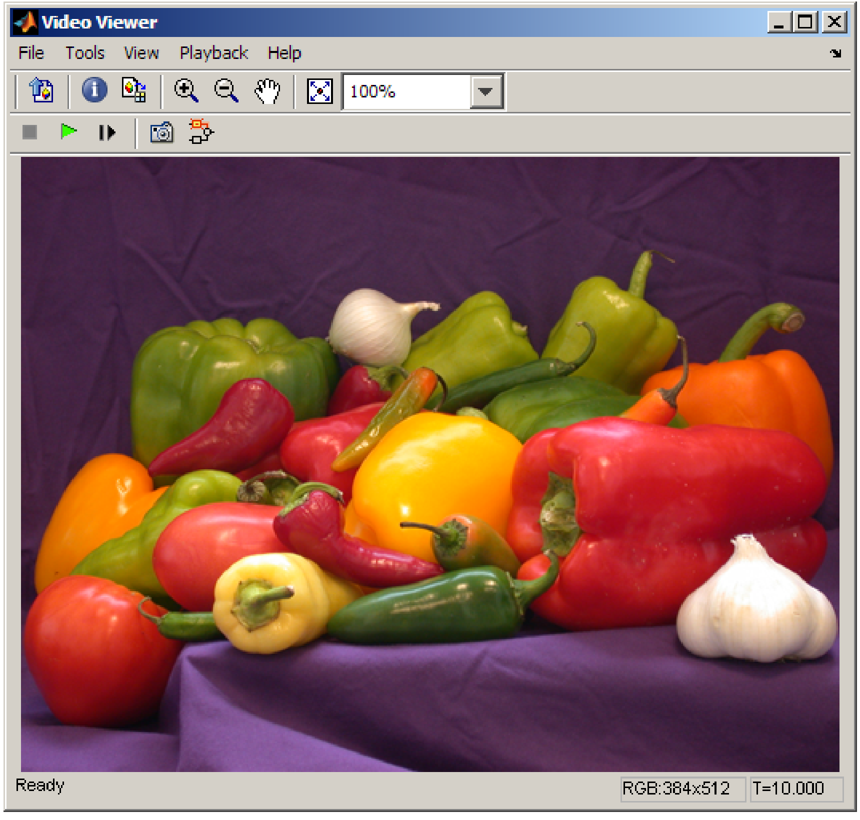


9 Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

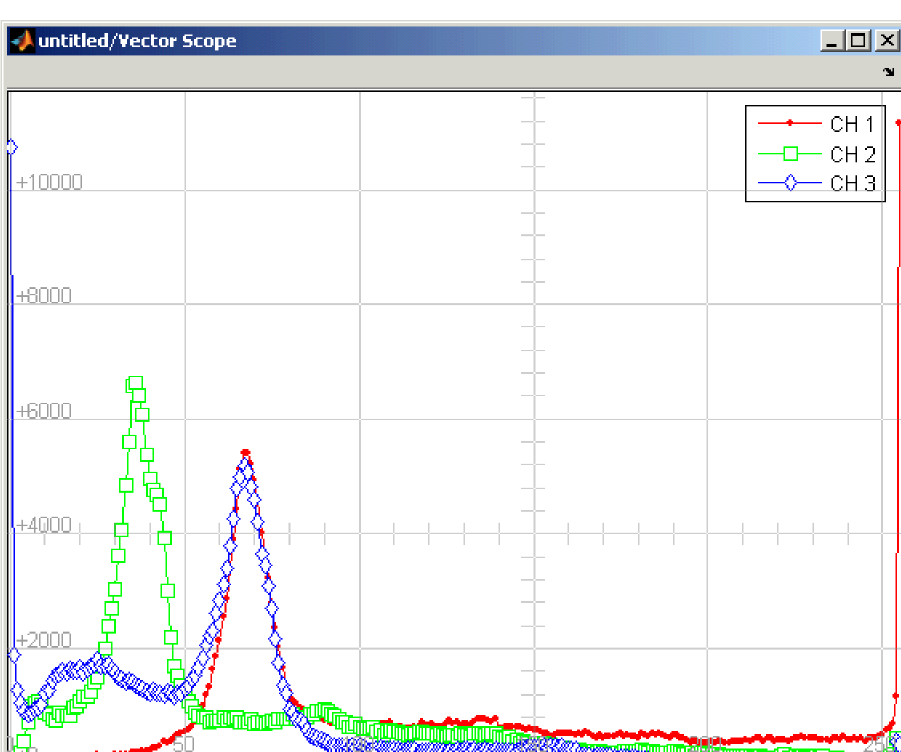
10 Run the model using either the simulation button, or by selecting Simulation > Start.

The original image appears in the Video Viewer window.



11 Right-click in the Vector Scope window and select **Autoscale**.

The scaled histogram of the image appears in the Vector Scope window.



You have now used the Histogram block to calculate the histogram of the R, G, and B values in an RGB image. For more information about this block, see the Histogram or 2D-Histogram reference pages. To open a demo model that illustrates how to use this block to calculate the histogram of the R, G, and B values in an RGB video stream, type `viphistogram` at the MATLAB command prompt.

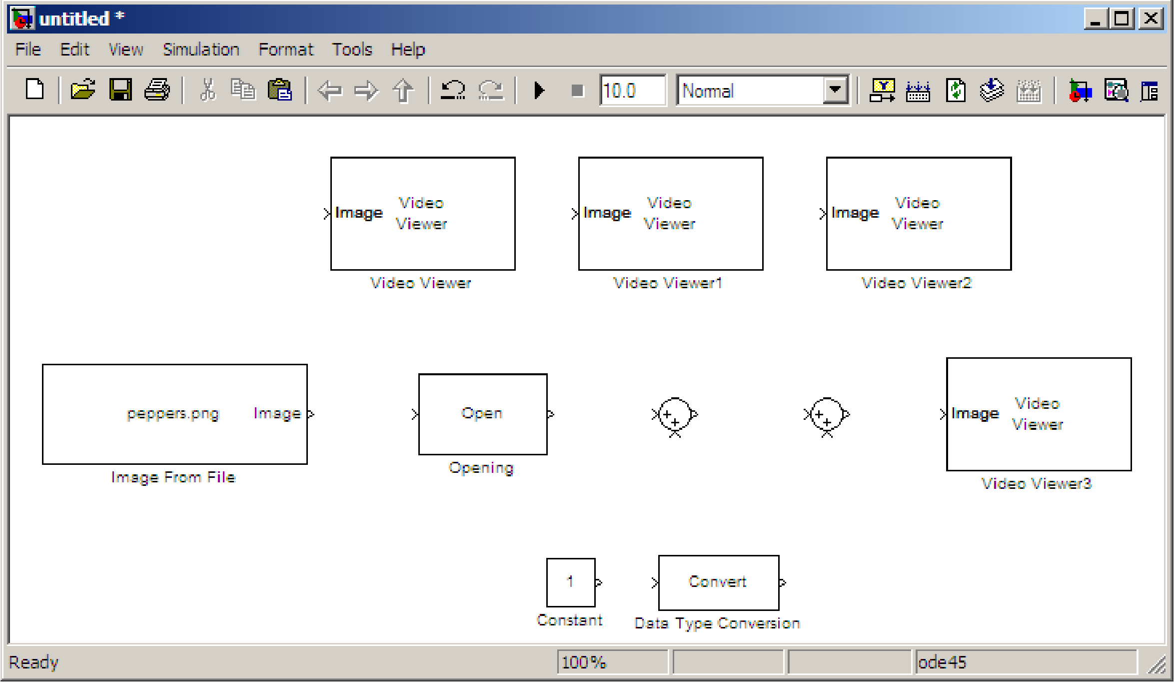
Correct Nonuniform Illumination

Global threshold techniques, which are often the first step in object measurement, cannot be applied to unevenly illuminated images. To correct this problem, you can change the lighting conditions and take another picture, or you can use morphological operators to even out the lighting in the image. Once you have corrected for nonuniform illumination, you can pick a global threshold that delineates every object from the background. In this topic, you use the Opening block to correct for uneven lighting in an intensity image:

1 Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Opening	Computer Vision System Toolbox > Morphological Operations	1
Video Viewer	Computer Vision System Toolbox > Sinks	4
Constant	Simulink > Sources	1
Sum	Simulink > Math Operations	2
Data Type Conversion	Simulink > Signal Attributes	1

2 Position the blocks as shown in the following figure.

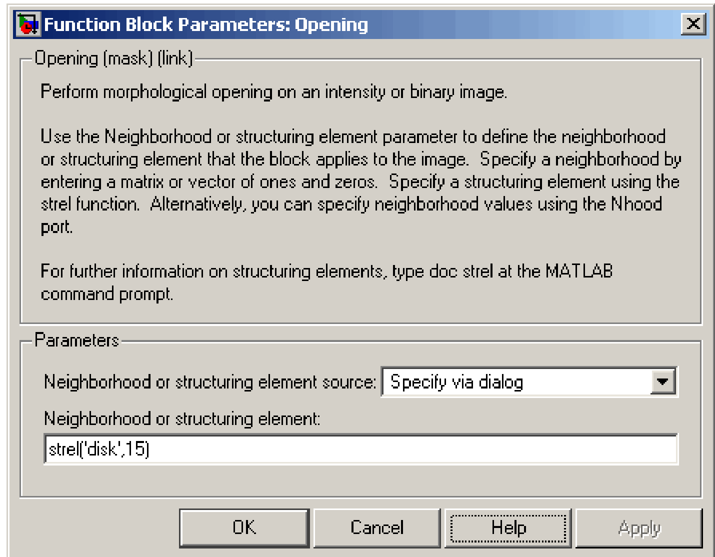


Once you have assembled the blocks required to correct for uneven illumination, you need to set your block parameters. To do this, double-click the blocks, modify the block parameter values, and click **OK**.

3 Use the Image From File block to import the intensity image. Set the **File name** parameter to `rice.png`. This image is a 256-by-256 matrix of 8-bit unsigned integer values.

4 Use the Video Viewer block to view the original image. Accept the default parameters.

5 Use the Opening block to estimate the background of the image. Set the **Neighborhood or structuring element** parameter to `strel('disk',15)`.



The `strel` function creates a circular STREL object with a radius of 15 pixels. When working with the Opening block, pick a STREL object that fits within the objects you want to keep. It often takes experimentation to find the neighborhood or STREL object that best suits your application.

6. Use the Video Viewer1 block to view the background estimated by the Opening block. Accept the default parameters.

7. Use the first Sum block to subtract the estimated background from the original image. Set the block parameters as follows:
   - **Icon shape** = `rectangular`
   - **List of signs** = `-+`

8. Use the Video Viewer2 block to view the result of subtracting the background from the original image. Accept the default parameters.

9. Use the Constant block to define an offset value. Set the **Constant value** parameter to `80`.

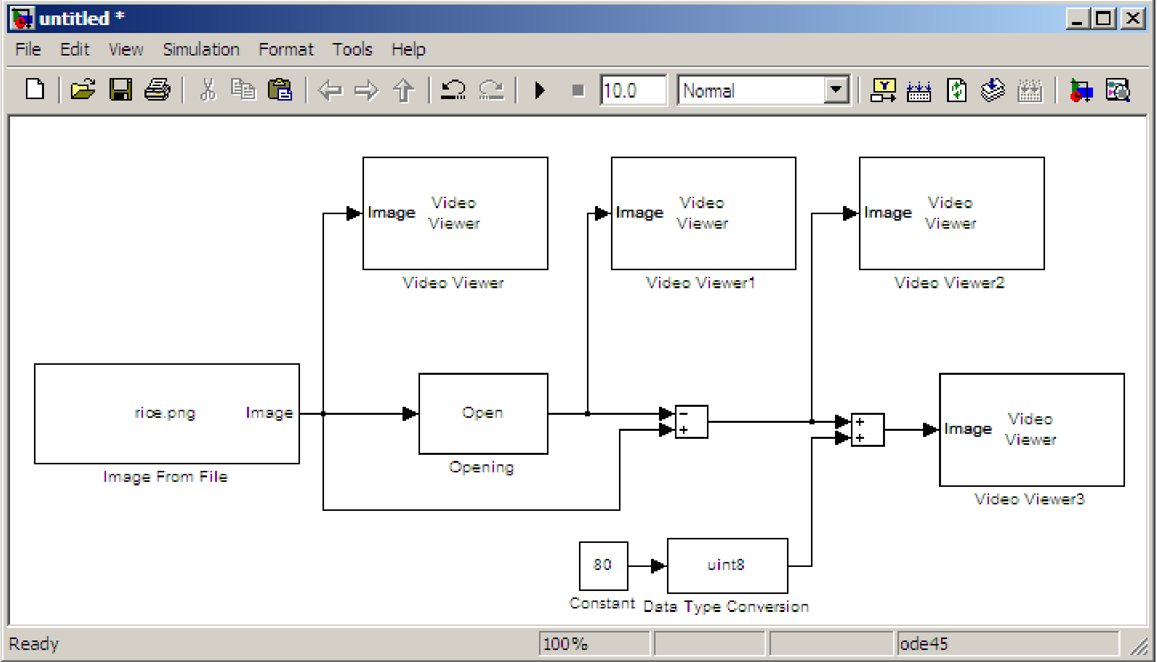
10. Use the Data Type Conversion block to convert the offset value to an 8-bit unsigned integer. Set the **Output data type mode** parameter to `uint8`.

11 Use the second Sum block to lighten the image so that it has the same brightness as the original image. Set the block parameters as follows:

- **Icon shape** = `rectangular`
- **List of signs** = `++`

12 Use the Video Viewer3 block to view the corrected image. Accept the default parameters.

13 Connect the blocks as shown in the following figure.

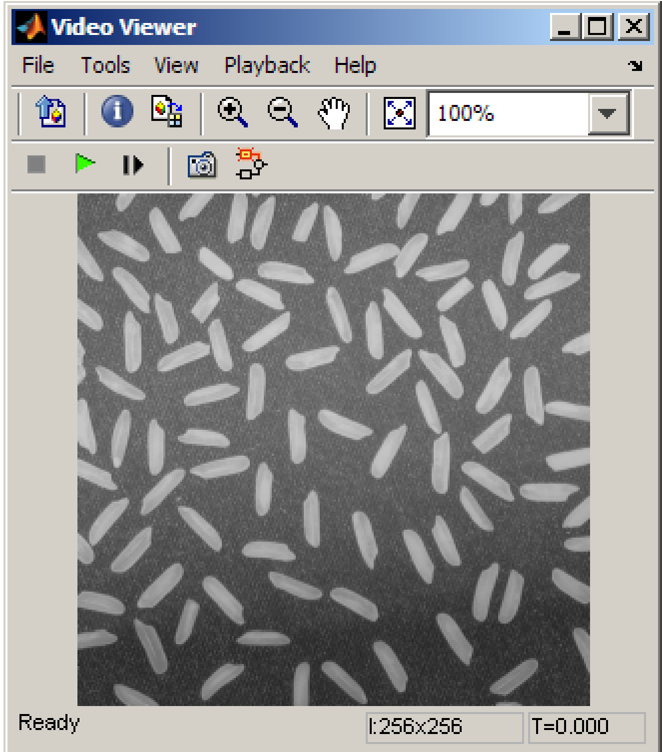


14 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

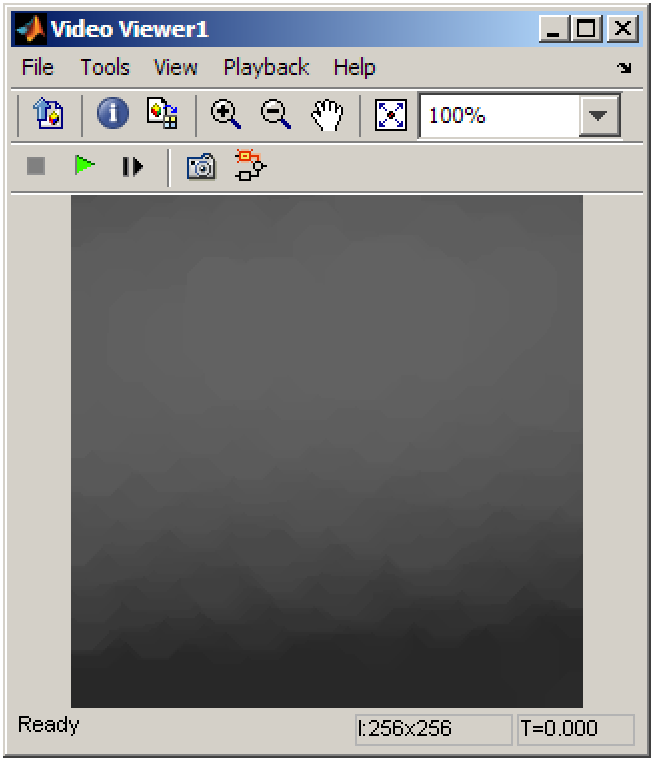
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `discrete (no continuous states)`

15 Run the model.

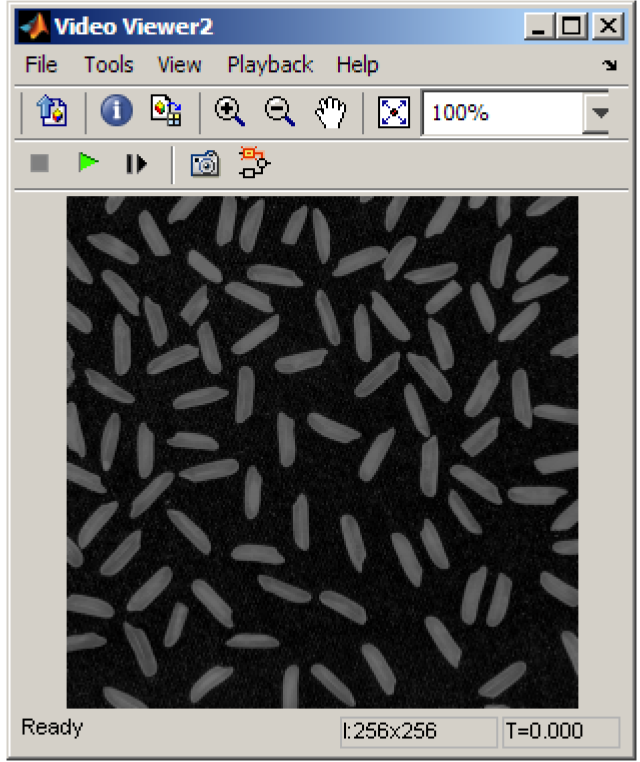
The original image appears in the Video Viewer window.



The estimated background appears in the Video Viewer1 window.

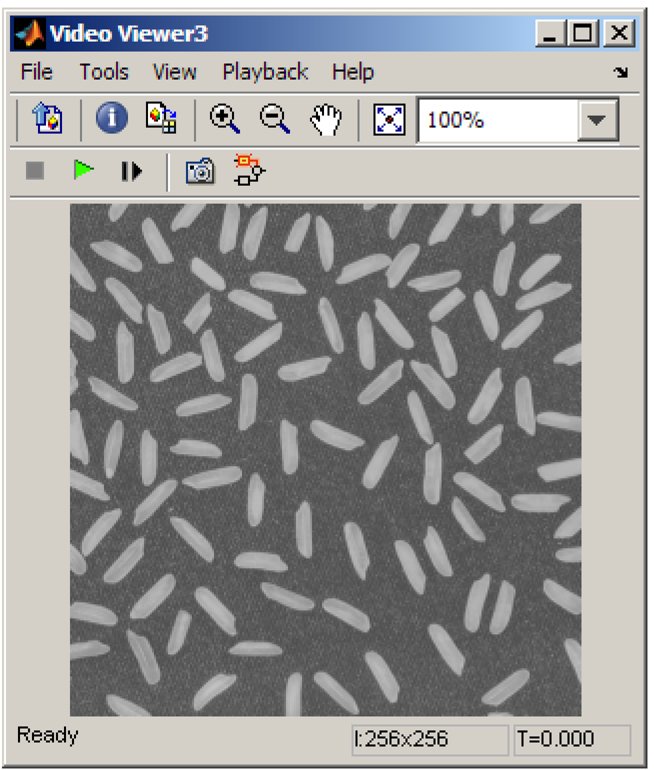


The image without the estimated background appears in the Video Viewer2 window.



The preceding image is too dark. The Constant block provides an offset value that you used to brighten the image.

The corrected image, which has even lighting, appears in the Video Viewer3 window. The following image is shown at its true size.



In this section, you have used the Opening block to remove irregular illumination from an image. For more information about this block, see the Opening reference page. For related information, see the Top-hat block reference page. For more information about STREL objects, see the `strel` function in the Image Processing Toolbox documentation.

Count Objects in an Image

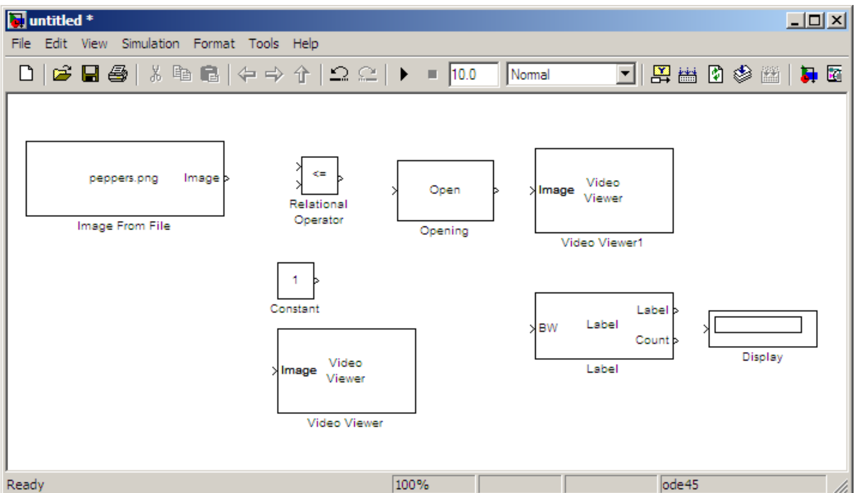
In this example, you import an intensity image of a wheel from the MATLAB workspace and convert it to binary. Then, using the Opening and Label blocks, you count the number of spokes in the wheel. You can use similar techniques to count objects in other intensity images. However, you might need to use additional morphological operators and different structuring elements.

Running this example requires a DSP System Toolbox license.

1. Create a new Simulink model, and add to it the blocks shown in the following table.

Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Opening	Computer Vision System Toolbox> Morphological Operations	1
Label	Computer Vision System Toolbox > Morphological Operations	1
Video Viewer	Computer Vision System Toolbox > Sinks	2
Constant	Simulink > Sources	1
Relational Operator	Simulink > Logic and Bit Operations	1
Display	DSP System Toolbox > Sinks	1

2. Position the blocks as shown in the following figure. The unconnected ports disappear when you set block parameters.



You are now ready to set your block parameters by double-clicking the blocks, modifying the block parameter values, and clicking **OK**.

3. Use the Image From File block to import your image. Set the **File name** parameter to `testpat1.png`. This is a 256-by-256 matrix image of 8-bit unsigned integers.

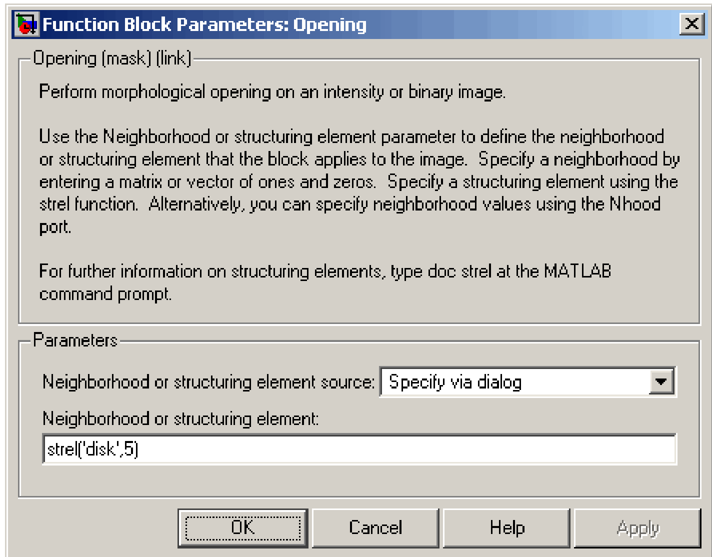
4. Use the Constant block to define a threshold value for the Relational Operator block. Set the **Constant value** parameter to `200`.

5. Use the Video Viewer block to view the original image. Accept the default parameters.

6. Use the Relational Operator block to perform a thresholding operation that converts your intensity image to a binary image. Set the **Relational Operator** parameter to <.

   If the input to the Relational Operator block is less than 200, its output is 1; otherwise, its output is 0. You must threshold your intensity image because the Label block expects binary input. Also, the objects it counts must be white.

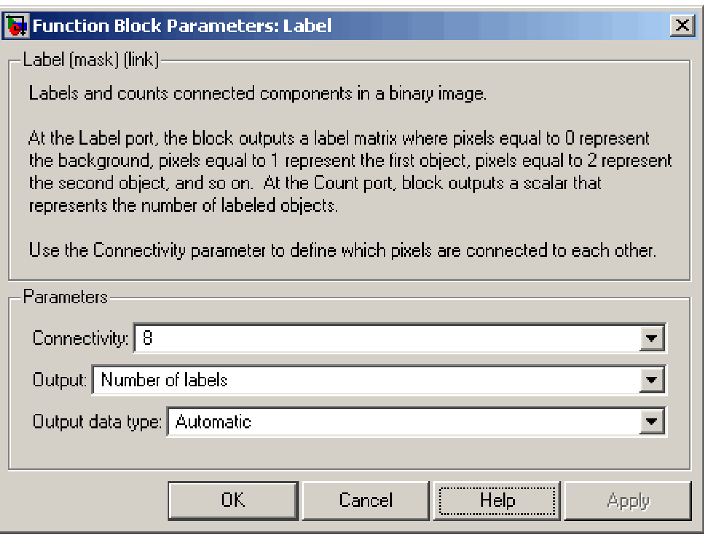
7. Use the Opening block to separate the spokes from the rim and from each other at the center of the wheel. Use the default parameters.



The `strel` function creates a circular STREL object with a radius of 5 pixels. When working with the Opening block, pick a STREL object that fits within the objects you want to keep. It often takes experimentation to find the neighborhood or STREL object that best suits your application.

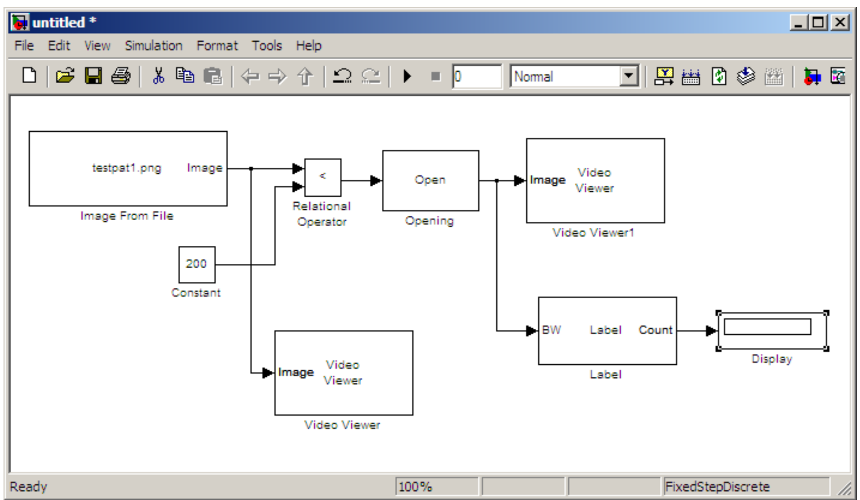
8. Use the Video Viewer1 block to view the opened image. Accept the default parameters.

9. Use the Label block to count the number of spokes in the input image. Set the **Output** parameter to `Number of labels`.



10 The Display block displays the number of spokes in the input image. Use the default parameters.

11 Connect the block as shown in the following figure.

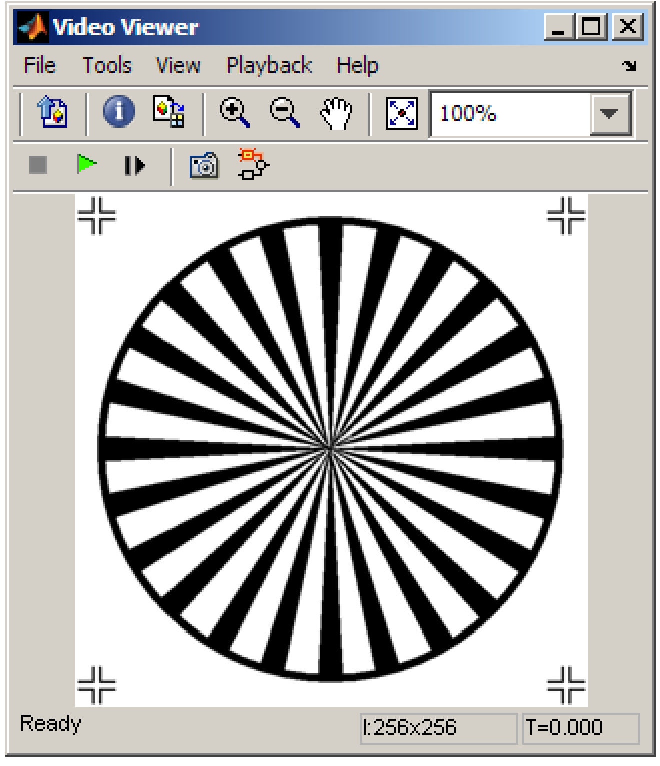


12 Set the configuration parameters. Open the Configuration dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

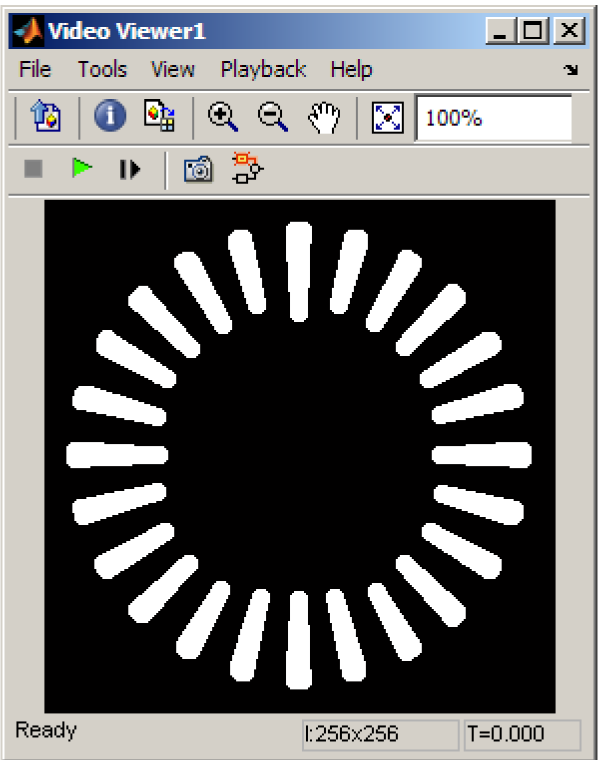
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `discrete (no continuous states)`

13 Run the model.

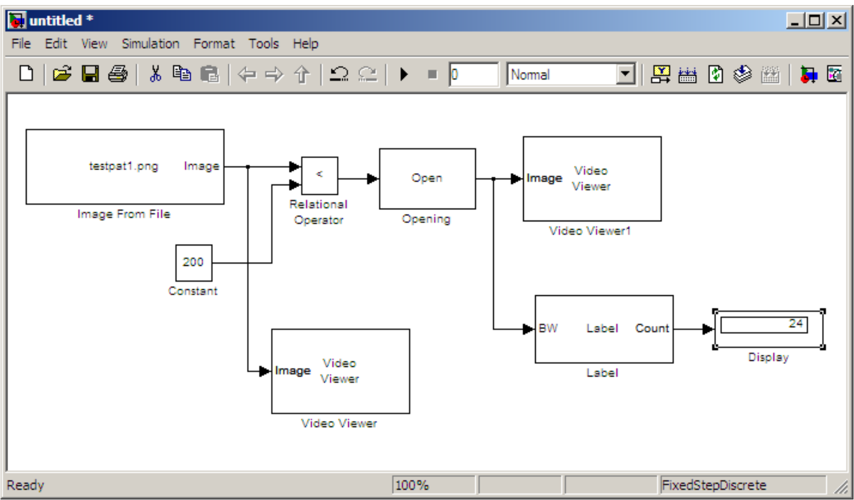
The original image appears in the Video Viewer1 window. To view the image at its true size, right-click the window and select **Set Display To True Size**.



The opened image appears in the Video Viewer window. The following image is shown at its true size.



As you can see in the preceding figure, the spokes are now separate white objects. In the model, the Display block correctly indicates that there are 24 distinct spokes.



You have used the Opening and Label blocks to count the number of spokes in an image. For more information about these blocks, see the Opening and Label block reference pages in the *Computer Vision System Toolbox Reference*. If you want to send the number of spokes to the MATLAB workspace, use the To Workspace block in Simulink or the Signal to Workspace block in DSP System Toolbox. For more information about STREL objects, see `strel` in the Image Processing Toolbox documentation.

8

Fixed-Point Design

- “Fixed-Point Signal Processing” on page 8-2
- “Fixed-Point Concepts and Terminology” on page 8-4
- “Arithmetic Operations” on page 8-10
- “Fixed-Point Support for MATLAB System Objects” on page 8-21
- “Specify Fixed-Point Attributes for Blocks” on page 8-25

Fixed-Point Signal Processing

In this section...

"Fixed-Point Features" on page 8-2

"Benefits of Fixed-Point Hardware" on page 8-2

"Benefits of Fixed-Point Design with System Toolboxes Software" on page 8-3

Note To take full advantage of fixed-point support in System Toolbox software, you must install Simulink Fixed Point™ software.

Fixed-Point Features

Many of the blocks in this product have fixed-point support, so you can design signal processing systems that use fixed-point arithmetic. Fixed-point support in DSP System Toolbox software includes

- Signed two's complement and unsigned fixed-point data types
- Word lengths from 2 to 128 bits in simulation
- Word lengths from 2 to the size of a `long` on the Simulink Coder C code-generation target
- Overflow handling and rounding methods
- C code generation for deployment on a fixed-point embedded processor, with Simulink Coder code generation software. The generated code uses all allowed data types supported by the embedded target, and automatically includes all necessary shift and scaling operations

Benefits of Fixed-Point Hardware

There are both benefits and trade-offs to using fixed-point hardware rather than floating-point hardware for signal processing development. Many signal processing applications require low-power and cost-effective circuitry, which makes fixed-point hardware a natural choice. Fixed-point hardware tends to be simpler and smaller. As a result, these units require less power and cost less to produce than floating-point circuitry.

Floating-point hardware is usually larger because it demands functionality and ease of development. Floating-point hardware can accurately represent real-world numbers, and its large dynamic range reduces the risk of overflow, quantization errors, and the need for scaling. In contrast, the smaller dynamic range of fixed-point hardware that allows for low-power, inexpensive units brings the possibility of these problems. Therefore, fixed-point development must minimize the negative effects of these factors, while exploiting the benefits of fixed-point hardware; cost- and size-effective units, less power and memory usage, and fast real-time processing.

Benefits of Fixed-Point Design with System Toolboxes Software

Simulating your fixed-point development choices before implementing them in hardware saves time and money. The built-in fixed-point operations provided by the System Toolboxes software save time in simulation and allow you to generate code automatically.

This software allows you to easily run multiple simulations with different word length, scaling, overflow handling, and rounding method choices to see the consequences of various fixed-point designs before committing to hardware. The traditional risks of fixed-point development, such as quantization errors and overflow, can be simulated and mitigated in software before going to hardware.

Fixed-point C code generation with System Toolbox software and Simulink Coder code generation software produces code ready for execution on a fixed-point processor. All the choices you make in simulation in terms of scaling, overflow handling, and rounding methods are automatically optimized in the generated code, without necessitating time-consuming and costly hand-optimized code. For more information on generating fixed-point code, see Code Generation.

Fixed-Point Concepts and Terminology

In this section...

"Fixed-Point Data Types" on page 8-4

"Scaling" on page 8-5

"Precision and Range" on page 8-6

Note The "Glossary" defines much of the vocabulary used in these sections. For more information on these subjects, see the Simulink Fixed Point and *Fixed-Point Toolbox™ User's Guide* documentation.

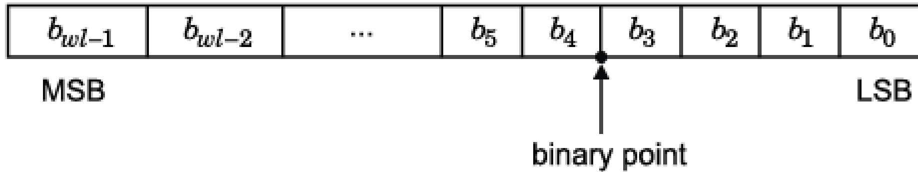
Fixed-Point Data Types

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

Binary numbers are represented as either fixed-point or floating-point data types. In this section, we discuss many terms and concepts relating to fixed-point numbers, data types, and mathematics.

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

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



where

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

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

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

Two's complement is the most common representation of signed fixed-point numbers and is used by System Toolbox software. See "Two's Complement" on page 8-11 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

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

where the slope can be expressed as

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

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

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

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

or

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

In System Toolbox software, you can define a fixed-point data type and scaling for the output or the parameters of many blocks by specifying the word length and fraction length of the quantity. The word length and fraction length define the whole of the data type and scaling information for binary-point only signals.

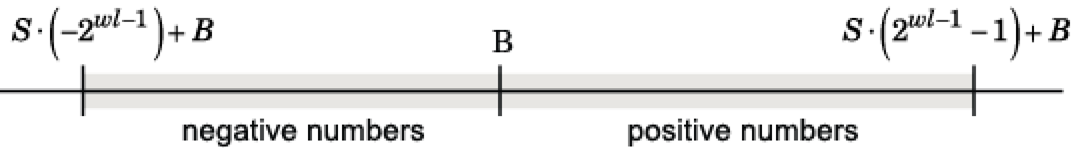
All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

Precision and Range

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

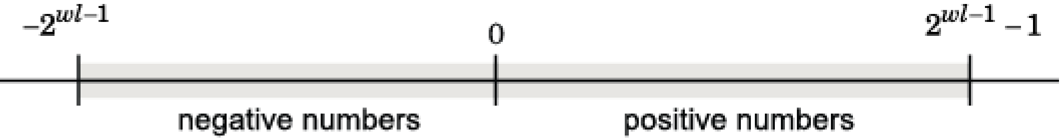
Range

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



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

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



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

System Toolbox software does not allow you to add guard bits to a data type on-the-fly in order to avoid overflows. Any guard bits must be allocated upon model initialization. However, the software does allow you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. See "Modulo Arithmetic" on page 8-10 for more information.

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value

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

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

Rounding Modes. When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and scaling, it is *rounded* to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the amount of bias that is introduced depends on the rounding mode itself. To provide you with greater flexibility in the trade-off between cost and bias, DSP System Toolbox software currently supports the following rounding modes:

- `Ceiling` rounds the result of a calculation to the closest representable number in the direction of positive infinity.
- `Convergent` rounds the result of a calculation to the closest representable number. In the case of a tie, `Convergent` rounds to the nearest even number. This is the least biased rounding mode provided by the toolbox.
- `Floor`, which is equivalent to truncation, rounds the result of a calculation to the closest representable number in the direction of negative infinity.
- `Nearest` rounds the result of a calculation to the closest representable number. In the case of a tie, `Nearest` rounds to the closest representable number in the direction of positive infinity.
- `Round` rounds the result of a calculation to the closest representable number. In the case of a tie, `Round` rounds positive numbers to the closest representable number in the direction of positive infinity, and rounds negative numbers to the closest representable number in the direction of negative infinity.
- `Simplest` rounds the result of a calculation using the rounding mode (`Floor` or `Zero`) that adds the least amount of extra rounding code to your

generated code. For more information, see “Rounding Mode: Simplest” in the Simulink Fixed Point documentation.

- `Zero` rounds the result of a calculation to the closest representable number in the direction of zero.

To learn more about each of these rounding modes, see “Rounding” in the Simulink Fixed Point documentation.

For a direct comparison of the rounding modes, see “Choosing a Rounding Method” in the Fixed-Point Toolbox documentation.

Arithmetic Operations

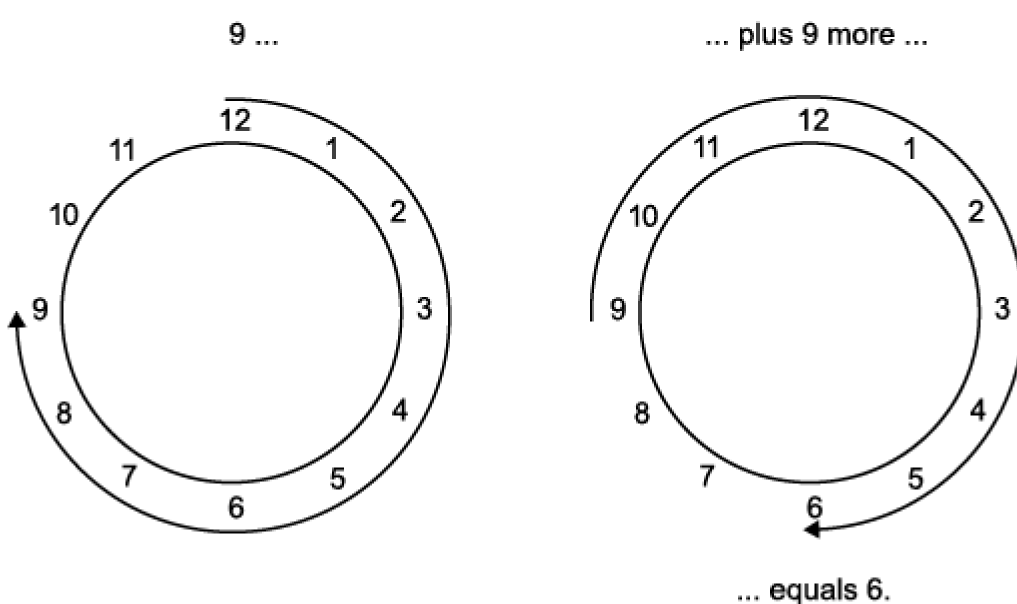
In this section...
"Modulo Arithmetic" on page 8-10
"Two's Complement" on page 8-11
"Addition and Subtraction" on page 8-12
"Multiplication" on page 8-13
"Casts" on page 8-16

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

Modulo Arithmetic

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

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



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

Two’s Complement

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

$$01 = (0 + 2^0) = 1$$
$$11 = ((-2^1) + (2^0)) = (-2 + 1) = -1$$

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

1. Take the one's complement, or "flip the bits."
2. Add a 1 using binary math.
3. Discard any bits carried beyond the original word length.

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

$$11010 \rightarrow 00101$$

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

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

Addition and Subtraction

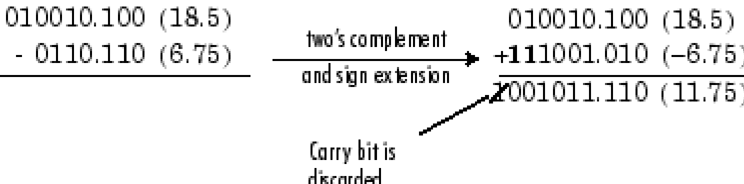
The addition of fixed-point numbers requires that the binary points of the addends be aligned. The addition is then performed using binary arithmetic so that no number other than 0 or 1 is used.

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

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

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends

must be sign extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

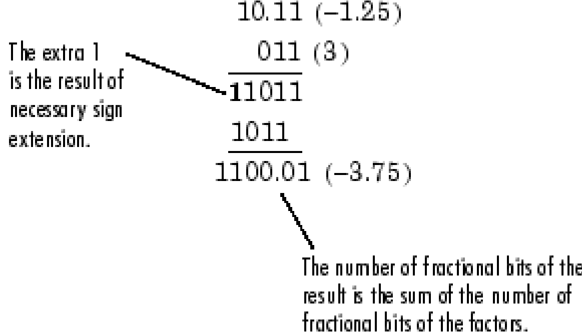


Most fixed-point DSP System Toolbox blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting is necessary during the addition to line up the binary points. See "Casts" on page 8-16 for more information.

Multiplication

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

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



Multiplication Data Types

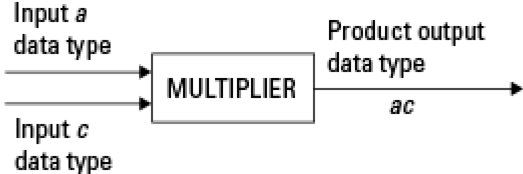
The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex

multiplication. See individual reference pages to determine whether a particular block accepts complex fixed-point inputs.

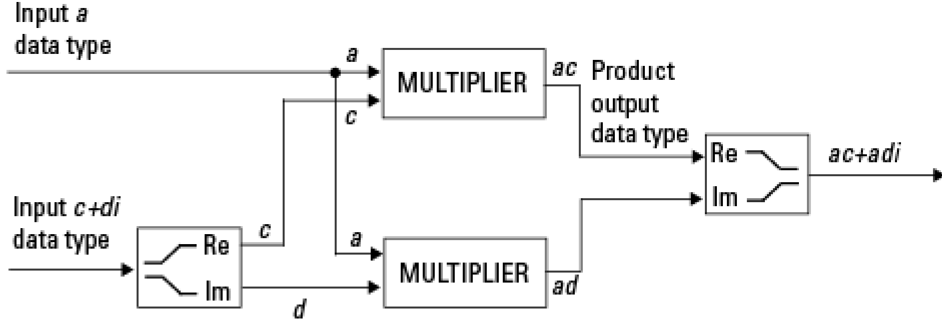
In most cases, you can set the data types used during multiplication in the block mask. See Accumulator Parameters, Intermediate Product Parameters, Product Output Parameters, and Output Parameters. These data types are defined in "Casts" on page 8-16.

Note The following diagrams show the use of fixed-point data types in multiplication in System Toolbox software. They do not represent actual subsystems used by the software to perform multiplication.

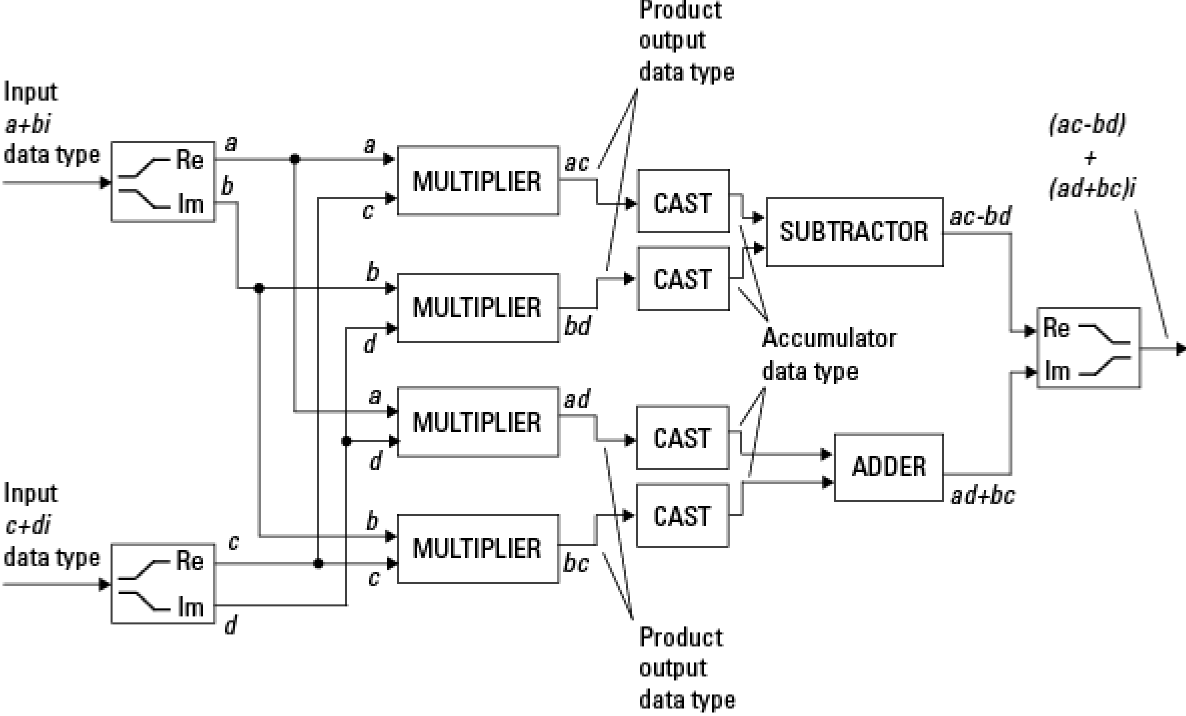
Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in System Toolbox software. The software returns the output of this operation in the product output data type, as the next figure shows.



Real-Complex Multiplication. The following diagram shows the data types used in the multiplication of a real and a complex fixed-point number in System Toolbox software. Real-complex and complex-real multiplication are equivalent. The software returns the output of this operation in the product output data type, as the next figure shows.



Complex-Complex Multiplication. The following diagram shows the multiplication of two complex fixed-point numbers in System Toolbox software. Note that the software returns the output of this operation in the accumulator output data type, as the next figure shows.



System Toolbox blocks cast to the accumulator data type before performing addition or subtraction operations. In the preceding diagram, this is equivalent to the C code

```
acc=ac;
acc-=bd;
```

for the subtractor, and

```
acc=ad;
acc+=bc;
```

for the adder, where *acc* is the accumulator.

Casts

Many fixed-point System Toolbox blocks that perform arithmetic operations allow you to specify the accumulator, intermediate product, and product output data types, as applicable, as well as the output data type of the block. This section gives an overview of the casts to these data types, so that you can tell if the data types you select will invoke sign extension, padding with zeros, rounding, and/or overflow.

Casts to the Accumulator Data Type

For most fixed-point System Toolbox blocks that perform addition or subtraction, the operands are first cast to an accumulator data type. Most of the time, you can specify the accumulator data type on the block mask. See Accumulator Parameters. Since the addends are both cast to the same accumulator data type before they are added together, no extra shift is necessary to insure that their binary points align. The result of the addition remains in the accumulator data type, with the possibility of overflow.

Casts to the Intermediate Product or Product Output Data Type

For System Toolbox blocks that perform multiplication, the output of the multiplier is placed into a product output data type. Blocks that then feed the product output back into the multiplier might first cast it to an intermediate product data type. Most of the time, you can specify these data types on the block mask. See Intermediate Product Parameters and Product Output Parameters.

Casts to the Output Data Type

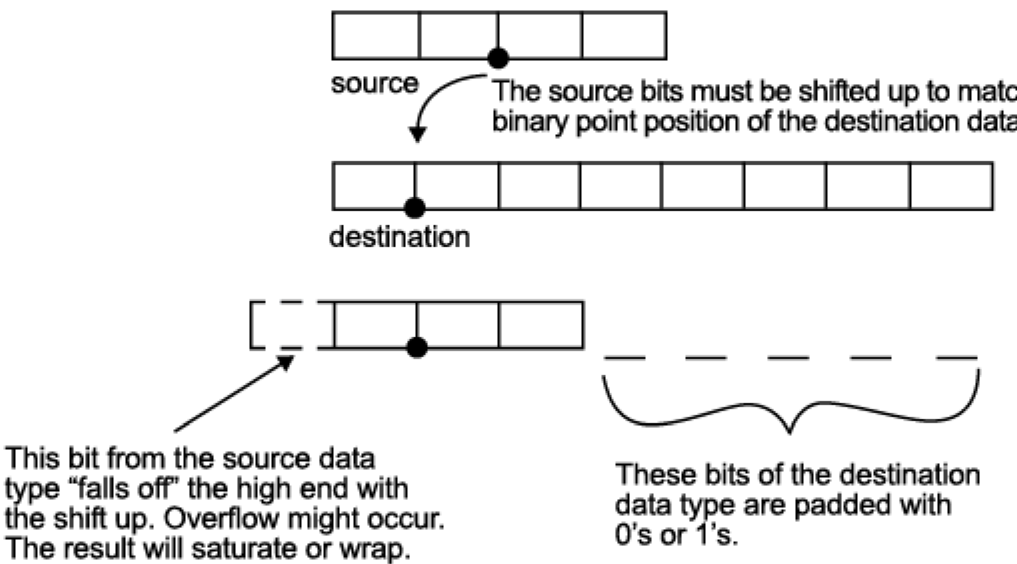
Many fixed-point System Toolbox blocks allow you to specify the data type and scaling of the block output on the mask. Remember that the software does not allow mixed types on the input and output ports of its blocks. Therefore, if you would like to specify a fixed-point output data type and scaling for a System Toolbox block that supports fixed-point data types, you must feed the input port of that block with a fixed-point signal. The final cast made by a fixed-point System Toolbox block is to the output data type of the block.

Note that although you can not mix fixed-point and floating-point signals on the input and output ports of blocks, you can have fixed-point signals with different word and fraction lengths on the ports of blocks that support fixed-point signals.

Casting Examples

It is important to keep in mind the ramifications of each cast when selecting these intermediate data types, as well as any other intermediate fixed-point data types that are allowed by a particular block. Depending upon the data types you select, overflow and/or rounding might occur. The following two examples demonstrate cases where overflow and rounding can occur.

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



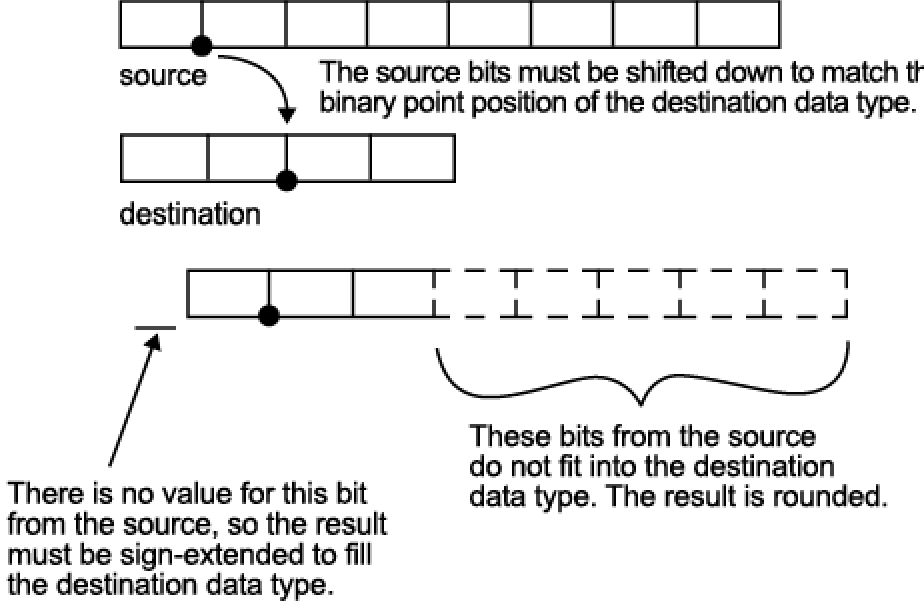
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either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

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

You can see that even with a cast from a shorter data type to a longer data type, overflow might still occur. This can happen when the integer length of the source data type (in this case two) is longer than the integer length of

the destination data type (in this case one). Similarly, rounding might be necessary even when casting from a shorter data type to a longer data type, if the destination data type and scaling has fewer fractional bits than the source.

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



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

In this case, even though the cast is from a longer data type to a shorter data type, all the integer bits are maintained. Conversely, full precision can be maintained even if you cast to a shorter data type, as long as the fraction length of the destination data type is the same length or longer than the

fraction length of the source data type. In that case, however, bits are lost from the high end of the result and overflow might occur.

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

Fixed-Point Support for MATLAB System Objects

In this section...

"Getting Information About Fixed-Point System Objects" on page 8-21

"Displaying Fixed-Point Properties" on page 8-22

"Setting System Object Fixed-Point Properties" on page 8-23

For information on working with Fixed-Point features, refer to Chapter 8, "Fixed-Point Design".

Getting Information About Fixed-Point System Objects

System objects that support fixed-point data processing have fixed-point properties, which you can display for a particular object by typing `vision.<ObjectName>.helpFixedPoint` at the command line.

See "Displaying Fixed-Point Properties" on page 8-22 to set the display of System object fixed-point properties.

The following Computer Vision System Toolbox objects support fixed-point data processing.

Fixed-Point Data Processing Support

```
vision.AlphaBlender
vision.Autocorrelator
vision.Autothresholder
vision.BlobAnalysis
vision.BlockMatcher
vision.ContrastAdjuster
vision.Convolver
vision.CornerDetector
vision.Crosscorrelator
vision.DCT
vision.Deinterlacer
vision.DemosaicInterpolator
```

```
vision.EdgeDetector
vision.FFT
vision.GeometricRotator
vision.GeometricScaler
vision.GeometricTranslator
vision.Histogram
vision.HoughLines
vision.HoughTransform
vision.IDCT
vision.IFFT
vision.ImageDataTypeConverter
vision.ImageFilter
vision.MarkerInserter
vision.Maximum
vision.Mean
vision.Median
vision.MedianFilter
vision.Minimum
vision.OpticalFlow
vision.PSNR
vision.Pyramid
vision.SAD
vision.ShapeInserter
vision.Variance
```

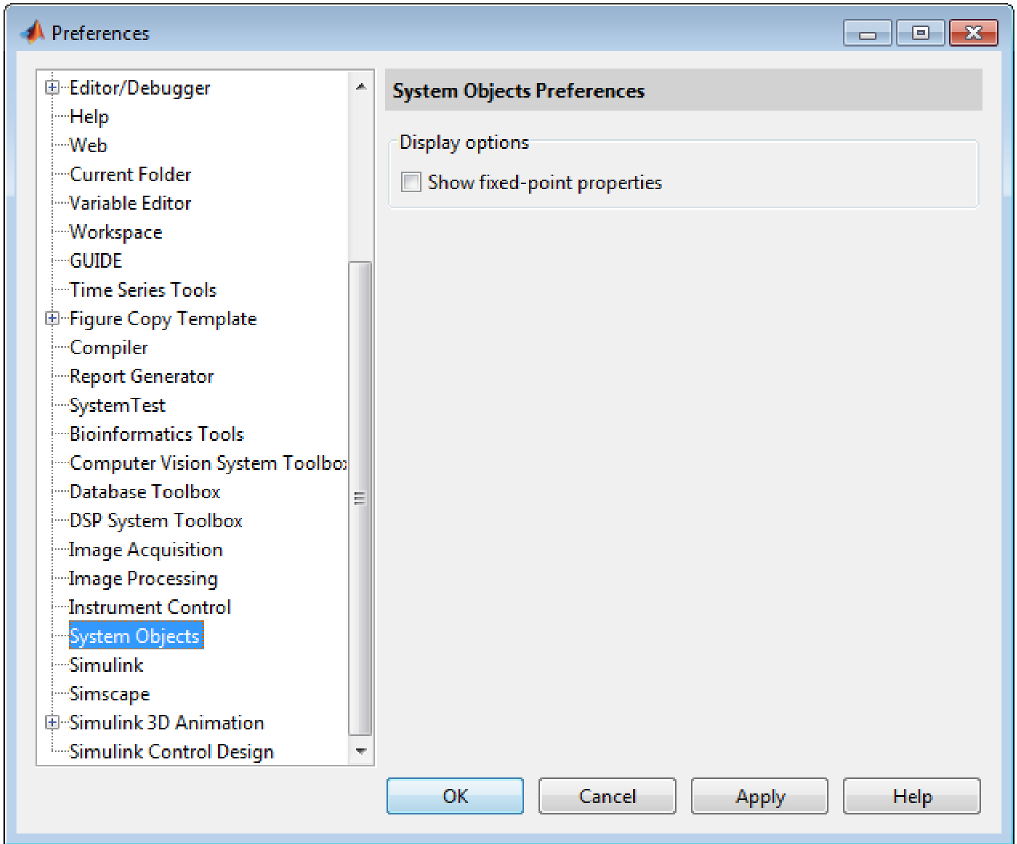
Displaying Fixed-Point Properties

You can control whether the software displays fixed-point properties with either of the following commands:

- `matlab.system.ShowFixedPointProperties`
- `matlab.system.HideFixedPointProperties`

at the MATLAB command line. These commands set the **Show fixed-point properties** display option. You can also set the display option directly via the MATLAB preferences dialog box. Select **File > Preferences** on the MATLAB

desktop, and then select **System Objects**. Finally, select or deselect **Show fixed-point properties**.



If an object supports fixed-point data processing, its fixed-point properties are active regardless of whether they are displayed or not.

Setting System Object Fixed-Point Properties

A number of properties affect the fixed-point data processing used by a System object. Objects perform fixed-point processing and use the current fixed-point property settings when they receive fixed-point input.

You change the values of fixed-point properties in the same way as you change any System object property value. See "Change a System Object Property". You also use the Fixed-Point Toolbox `numerictype` object to specify the desired data type as fixed-point, the signedness, and the word- and fraction-lengths.

In the same way as for blocks, the data type properties of many System objects can set the appropriate word lengths and scalings automatically by using full precision. System objects assume that the target specified on the Configuration Parameters Hardware Implementation target is `ASIC/FPGA`.

If you have not set the property that activates a dependent property and you attempt to change that dependent property, a warning message displays. For example, for the `vision.EdgeDetector` object, before you set `CustomProductDataType` to `numerictype(1,16,15)` you must set `ProductDataType` to `'Custom'`.

Note System objects do not support fixed-point word lengths greater than 128 bits.

The fimath settings for any fimath attached to a fi input or a fi property are ignored. Outputs from a System object never have an attached fimath.

Specify Fixed-Point Attributes for Blocks

In this section...

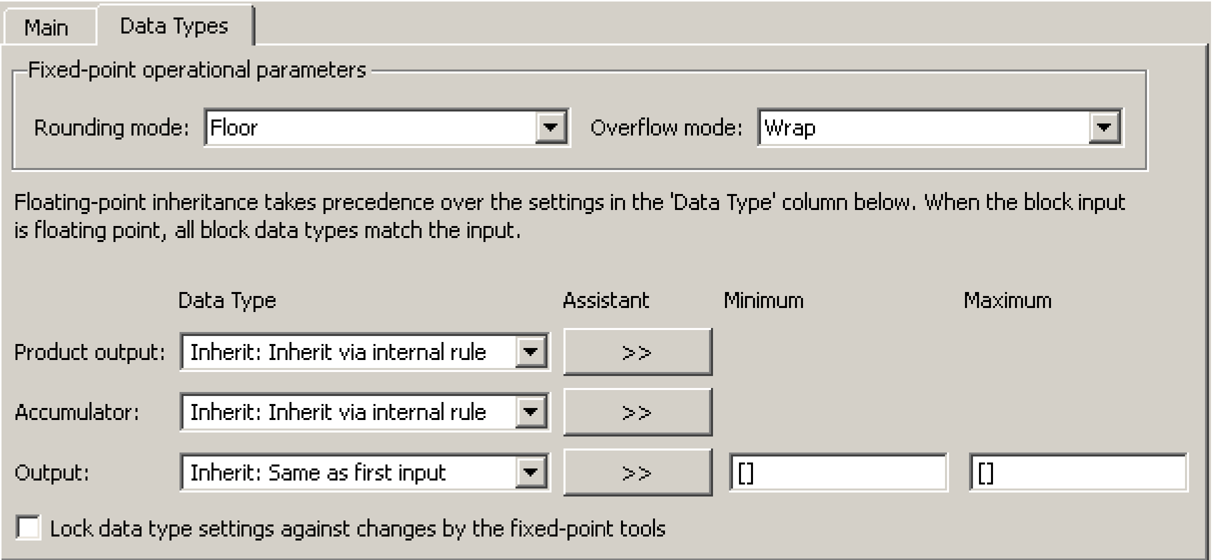
"Fixed-Point Block Parameters" on page 8-25
"Specify System-Level Settings" on page 8-28
"Inherit via Internal Rule" on page 8-29
"Select and Specify Data Types for Fixed-Point Blocks" on page 8-40

Fixed-Point Block Parameters

System Toolbox blocks that have fixed-point support usually allow you to specify fixed-point characteristics through block parameters. By specifying data type and scaling information for these fixed-point parameters, you can simulate your target hardware more closely.

Note Floating-point inheritance takes precedence over the settings discussed in this section. When the block has floating-point input, all block data types match the input.

You can find most fixed-point parameters on the **Data Types** pane of System Toolbox blocks. The following figure shows a typical **Data Types** pane.



All System Toolbox blocks with fixed-point capabilities share a set of common parameters, but each block can have a different subset of these fixed-point parameters. The following table provides an overview of the most common fixed-point block parameters.

Fixed-Point Data Type Parameter	Description
Rounding Mode	Specifies the rounding mode for the block to use when the specified data type and scaling cannot exactly represent the result of a fixed-point calculation. See "Rounding Modes" on page 8-8 for more information on the available options.
Overflow Mode	Specifies the overflow mode to use when the result of a fixed-point calculation does not fit into the representable range of the specified data type. See "Overflow Handling" on page 8-7 for more information on the available options.

Fixed-Point Data Type Parameter	Description
Intermediate Product	Specifies the data type and scaling of the intermediate product for fixed-point blocks. Blocks that feed multiplication results back to the input of the multiplier use the intermediate product data type. See the reference page of a specific block to learn about the intermediate product data type for that block.
Product Output	Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results. See the reference page of a specific block to learn about the product output data type for that block. For or complex-complex multiplication, the multiplication result is in the accumulator data type. See "Multiplication Data Types" on page 8-13 for more information on complex fixed-point multiplication in System toolbox software.
Accumulator	Specifies the data type and scaling of the accumulator (sum) for fixed-point blocks that must hold summation results for further calculation. Most such blocks cast to the accumulator data type before performing the add operations (summation). See the reference page of a specific block for details on the accumulator data type of that block.
Output	Specifies the output data type and scaling for blocks.

Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool available on the **Data Types** pane of some fixed-point System Toolbox blocks.

To learn more about using the **Data Type Assistant** to help you specify block data type parameters, see the following section of the Simulink documentation:

"Using the Data Type Assistant"

Checking Signal Ranges

Some fixed-point System Toolbox blocks have **Minimum** and **Maximum** parameters on the **Data Types** pane. When a fixed-point data type has these parameters, you can use them to specify appropriate minimum and maximum values for range checking purposes.

To learn how to specify signal ranges and enable signal range checking, see "Signal Ranges" in the Simulink documentation.

Specify System-Level Settings

You can monitor and control fixed-point settings for System Toolbox blocks at a system or subsystem level with the Fixed-Point Tool. For additional information on these subjects, see

- The `fxptdlg` reference page — A reference page on the Fixed-Point Tool in the Simulink documentation
- "Fixed-Point Tool" — A tutorial that highlights the use of the Fixed-Point Tool in the Simulink Fixed Point software documentation

Logging

The Fixed-Point Tool logs overflows, saturations, and simulation minimums and maximums for fixed-point System Toolbox blocks. The Fixed-Point Tool does not log overflows and saturations when the `Data overflow` line in the **Diagnostics > Data Integrity** pane of the Configuration Parameters dialog box is set to `None`.

Autoscaling

You can use the Fixed-Point Tool autoscaling feature to set the scaling for System Toolbox fixed-point data types.

Data type override

System Toolbox blocks obey the `Use local settings`, `Double`, `Single`, and `Off` modes of the **Data type override** parameter in the Fixed-Point Tool. The `Scaled double` mode is also supported for System Toolboxes source and byte-shuffling blocks, and for some arithmetic blocks such as Difference and Normalization.

Inherit via Internal Rule

Selecting appropriate word lengths and scalings for the fixed-point parameters in your model can be challenging. To aid you, an `Inherit via internal rule` choice is often available for fixed-point block data type parameters, such as the **Accumulator** and **Product output** signals. The following sections describe how the word and fraction lengths are selected for you when you choose `Inherit via internal rule` for a fixed-point block data type parameter in System Toolbox software:

- "Internal Rule for Accumulator Data Types" on page 8-29
- "Internal Rule for Product Data Types" on page 8-30
- "Internal Rule for Output Data Types" on page 8-30
- "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 8-31
- "Internal Rule Examples" on page 8-32

Note In the equations in the following sections, WL = word length and FL = fraction length.

Internal Rule for Accumulator Data Types

The internal rule for accumulator data types first calculates the ideal, full-precision result. Where N is the number of addends:

$$WL_{ideal\ accumulator} = WL_{input\ to\ accumulator} + \text{floor}(\log_2(N-1)) + 1$$

$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$

For example, consider summing all the elements of a vector of length 6 and data type sfix10_En8. The ideal, full-precision result has a word length of 13 and a fraction length of 8.

The accumulator can be real or complex. The preceding equations are used for both the real and imaginary parts of the accumulator. For any calculation, after the full-precision result is calculated, the final word and fraction lengths set by the internal rule are affected by your particular hardware. See "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 8-31 for more information.

Internal Rule for Product Data Types

The internal rule for product data types first calculates the ideal, full-precision result:

$$WL_{ideal\ product} = WL_{input\ 1} + WL_{input\ 2}$$

$$FL_{ideal\ product} = FL_{input\ 1} + FL_{input\ 2}$$

For example, multiplying together the elements of a real vector of length 2 and data type sfix10_En8. The ideal, full-precision result has a word length of 20 and a fraction length of 16.

For real-complex multiplication, the ideal word length and fraction length is used for both the complex and real portion of the result. For complex-complex multiplication, the ideal word length and fraction length is used for the partial products, and the internal rule for accumulator data types described above is used for the final sums. For any calculation, after the full-precision result is calculated, the final word and fraction lengths set by the internal rule are affected by your particular hardware. See "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 8-31 for more information.

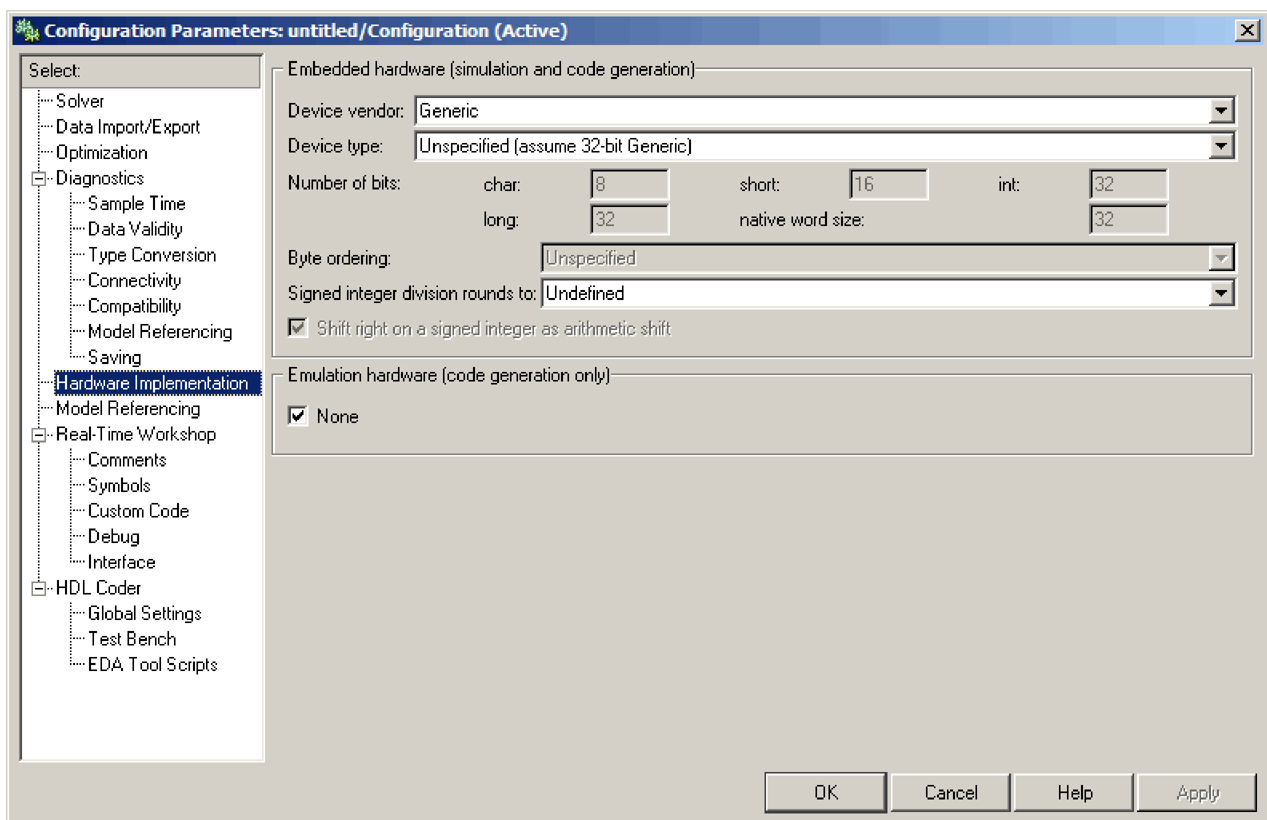
Internal Rule for Output Data Types

A few System Toolbox blocks have an `Inherit via internal rule` choice available for the block output. The internal rule used in these cases is block-specific, and the equations are listed in the block reference page.

As with accumulator and product data types, the final output word and fraction lengths set by the internal rule are affected by your particular hardware, as described in "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 8-31.

The Effect of the Hardware Implementation Pane on the Internal Rule

The internal rule selects word lengths and fraction lengths that are appropriate for your hardware. To get the best results using the internal rule, you must specify the type of hardware you are using on the **Hardware Implementation** pane of the Configuration Parameters dialog box. You can open this dialog box from the **Simulation** menu in your model.



ASIC/FPGA. On an ASIC/FPGA target, the ideal, full-precision word length and fraction length calculated by the internal rule are used. If the calculated ideal word length is larger than the largest allowed word length, you receive an error. The largest word length allowed for Simulink and System Toolbox software is 128 bits.

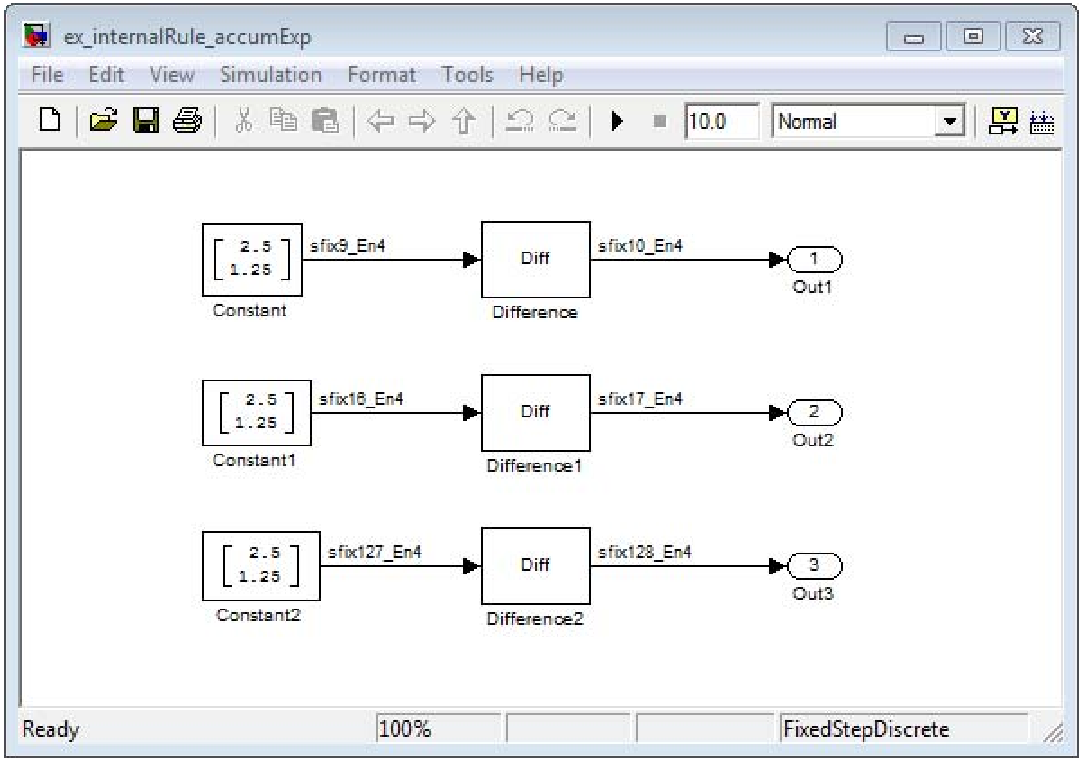
Other targets. For all targets other than ASIC/FPGA, the ideal, full-precision word length calculated by the internal rule is rounded up to the next available word length of the target. The calculated ideal fraction length is used, keeping the least-significant bits.

If the calculated ideal word length for a product data type is larger than the largest word length on the target, you receive an error. If the calculated ideal word length for an accumulator or output data type is larger than the largest word length on the target, the largest target word length is used.

Internal Rule Examples

The following sections show examples of how the internal rule interacts with the **Hardware Implementation** pane to calculate accumulator data types and product data types.

Accumulator Data Types. Consider the following model ex_internalRule_accumExp.



In the Difference blocks, the **Accumulator** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit: Same as accumulator`. Therefore, you can see the accumulator data type calculated by the internal rule on the output signal in the model.

In the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to `ASIC/FPGA`. Therefore, the accumulator data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Difference blocks in the model:

$$WL_{ideal\ accumulator} = WL_{input\ to\ accumulator} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator} = 9 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator} = 9 + 0 + 1 = 10$$

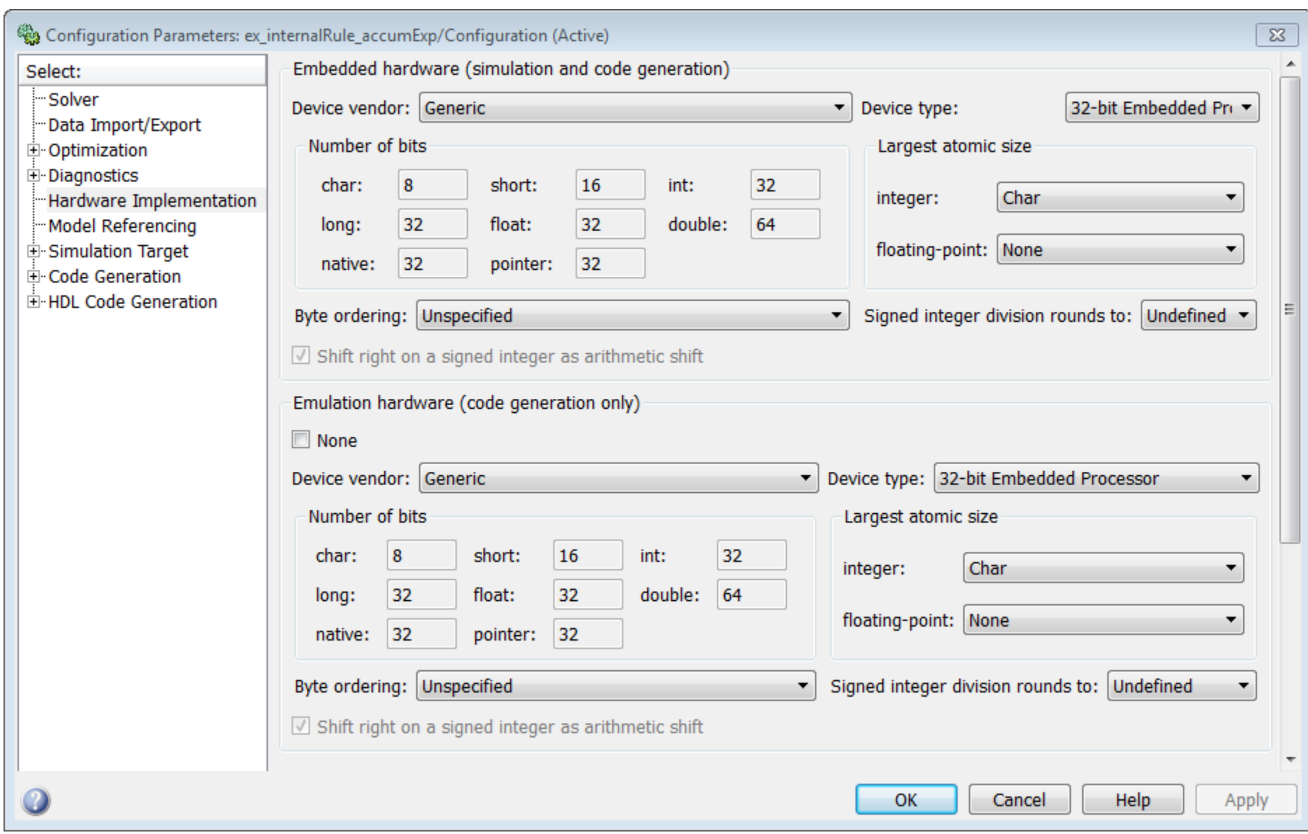
$$WL_{ideal\ accumulator1} = WL_{input\ to\ accumulator1} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator1} = 16 + 0 + 1 = 17$$

$$WL_{ideal\ accumulator2} = WL_{input\ to\ accumulator2} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator2} = 127 + 0 + 1 = 128$$

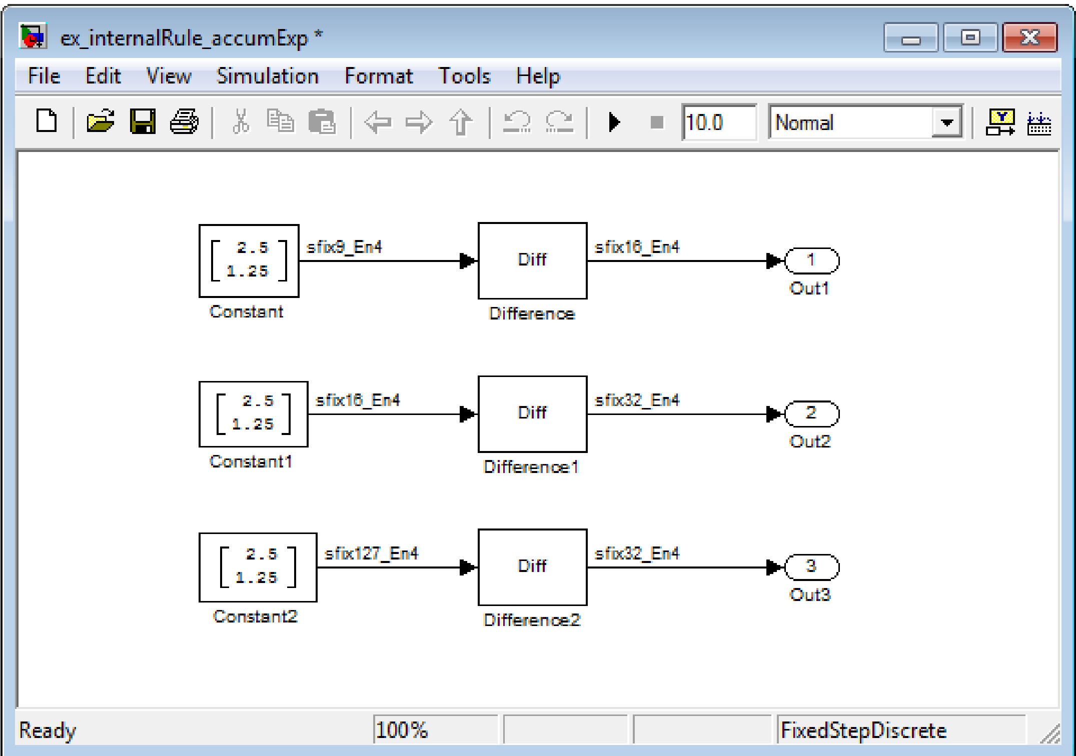
Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$
$$FL_{ideal\ accumulator} = 4$$

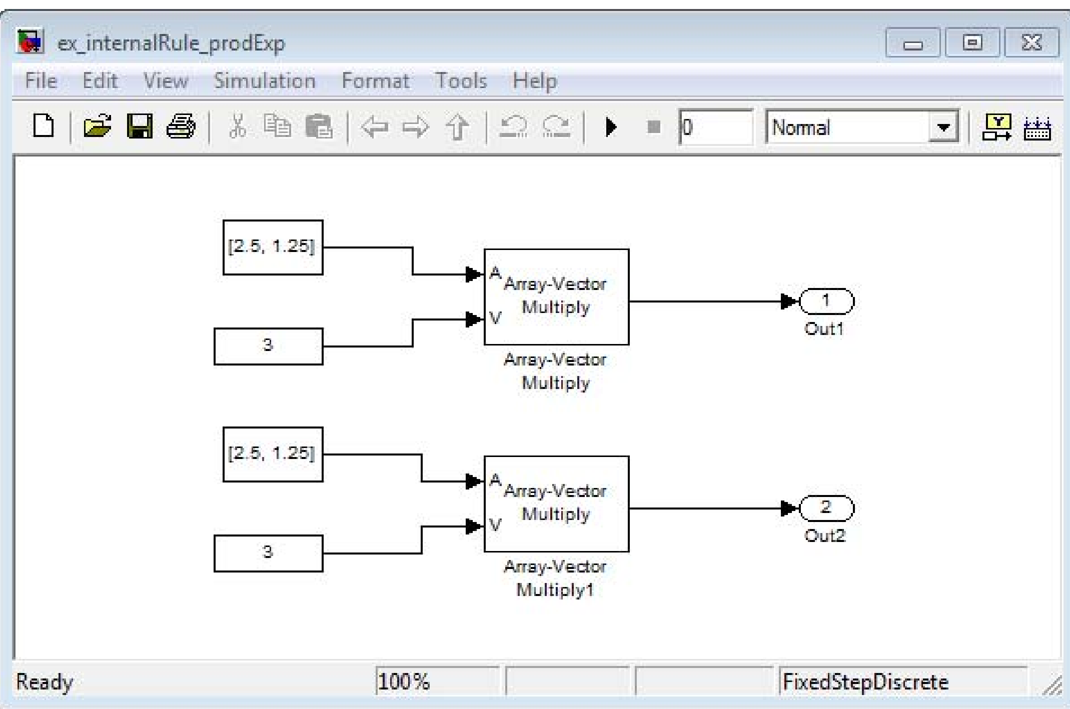
Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to `32 bit Embedded Processor`, by changing the parameters as shown in the following figure.



As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 10, 17, and 128 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case You can see this if you rerun the model, as shown in the following figure.


ex_internalRule_accumExp *
File Edit View Simulation Format Tools Help
10.0
Normal
2.5
1.25
Constant
sfix9_En4
Diff
Difference
sfix16_En4
1
Out1
2.5
1.25
Constant1
sfix16_En4
Diff
Difference1
sfix32_En4
2
Out2
2.5
1.25
Constant2
sfix127_En4
Diff
Difference2
sfix32_En4
3
Out3
Ready
100%
FixedStepDiscrete

Product Data Types. Consider the following model ex_internalRule_prodExp.



In the Array-Vector Multiply blocks, the **Product Output** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit: Same as product output`. Therefore, you can see the product output data type calculated by the internal rule on the output signal in the model. The setting of the **Accumulator** parameter does not matter because this example uses real values.

For the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to `ASIC/FPGA`. Therefore, the product data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Array-Vector Multiply blocks in the model:

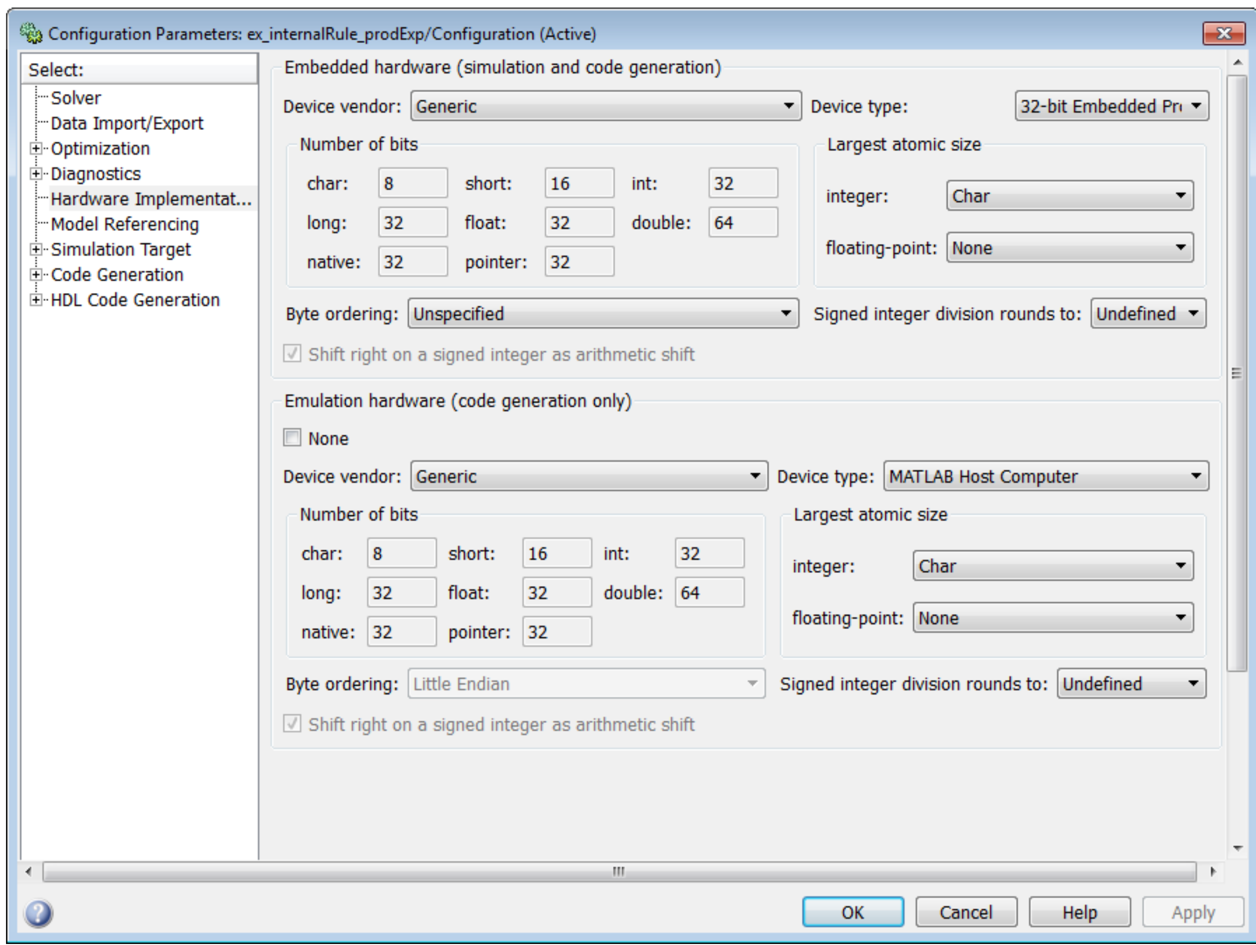
$$WL_{ideal\ product} = WL_{input\ a} + WL_{input\ b}$$
$$WL_{ideal\ product} = 7 + 5 = 12$$

$$WL_{ideal\ product1} = WL_{input\ a} + WL_{input\ b}$$
$$WL_{ideal\ product1} = 16 + 15 = 31$$

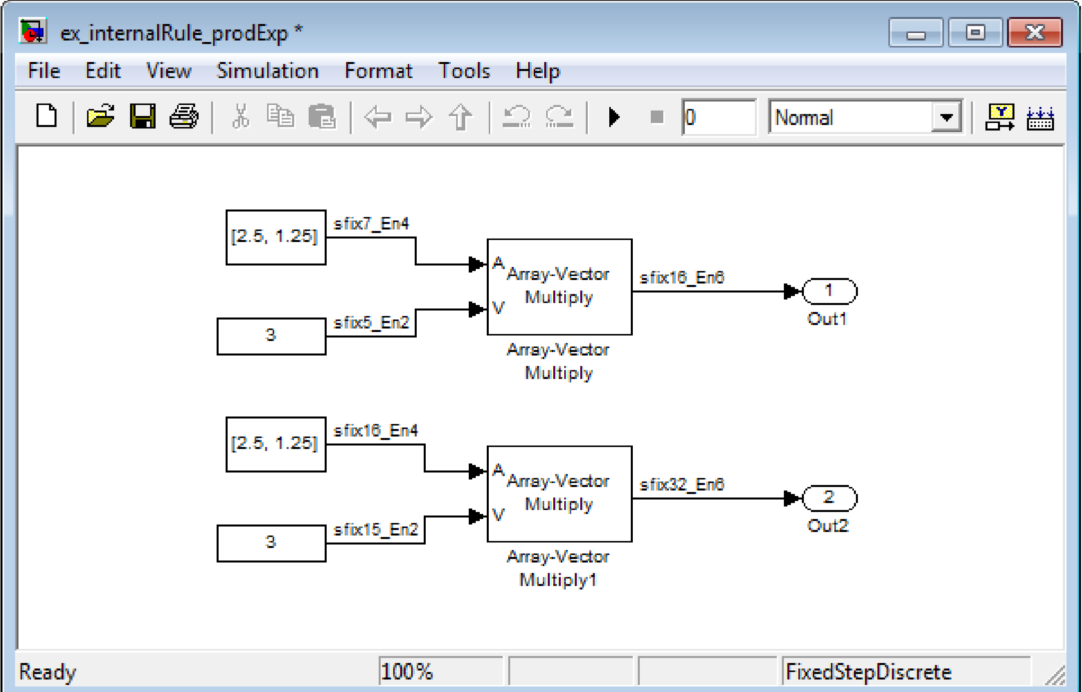
Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$
$$FL_{ideal\ accumulator} = 4$$

Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to `32 bit Embedded Processor`, as shown in the following figure.



As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 12 and 31 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case. You can see this if you rerun the model, as shown in the following figure.



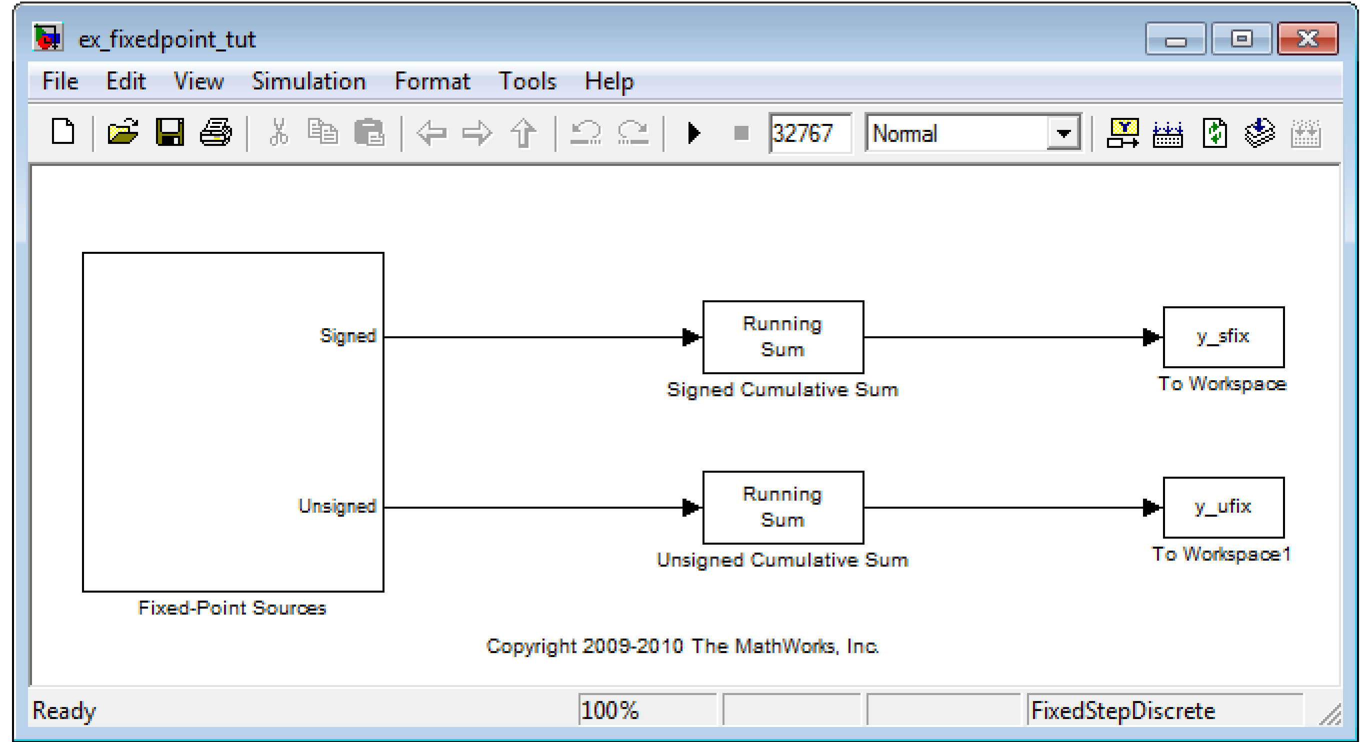
Select and Specify Data Types for Fixed-Point Blocks

The following sections show you how to use the Fixed-Point Tool to select appropriate data types for fixed-point blocks in the `ex_fixedpoint_tut` model:

- "Prepare the Model" on page 8-40
- "Use Data Type Override to Find a Floating-Point Benchmark" on page 8-45
- "Use the Fixed-Point Tool to Propose Fraction Lengths" on page 8-46
- "Examine the Results and Accept the Proposed Scaling" on page 8-46

Prepare the Model

1 Open the model by typing ex_fixedpoint_tut at the MATLAB command line.



This model uses the Cumulative Sum block to sum the input coming from the Fixed-Point Sources subsystem. The Fixed-Point Sources subsystem outputs two signals with different data types:

- The Signed source has a word length of 16 bits and a fraction length of 15 bits.
- The Unsigned source has a word length of 16 bits and a fraction length of 16 bits.

2 Run the model to check for overflow. MATLAB displays the following warnings at the command line:

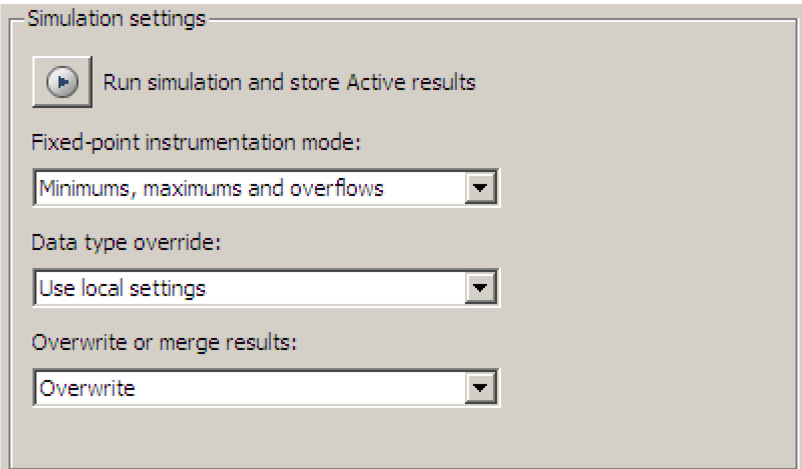
```
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Signed Cumulative Sum'.
Warning: Overflow occurred. This originated from
```

`'ex_fixedpoint_tut/Unsigned Cumulative Sum'.`

According to these warnings, overflow occurs in both Cumulative Sum blocks. You can control the display of these warnings using the "Configuration Parameters Dialog Box".

3 To investigate the overflows in this model, use the Fixed-Point Tool. You can open the Fixed-Point Tool by selecting **Tools > Fixed-Point > Fixed-Point Tool** from the model menu. Turn on logging for all blocks in your model by setting the **Fixed-point instrumentation mode** parameter to `Minimums, maximums and overflows`.

4 Now that you have turned on logging, rerun the model by clicking the **Run simulation and store active results** button in the **Simulation settings pane**.



5 The results of the simulation appear in a table in the central **Contents** pane of the Fixed-Point Tool. Review the following columns:

- **Name** — Provides the name of each signal in the following format: `Subsystem Name/Block Name:  Signal Name`.
- **SimDT** — The simulation data type of each logged signal.
- **SpecifiedDT** — The data type specified on the block dialog for each signal.

- **SimMin** — The smallest representable value achieved during simulation for each logged signal.
- **SimMax** — The largest representable value achieved during simulation for each logged signal.
- **OverflowWraps** — The number of overflows that wrap during simulation.

For more information on each of the columns in this table, see the "Contents Pane" section of the Simulink `fxptdlg` function reference page.

You can also see that the **SimMin** and **SimMax** values for the Accumulator data types range from `0` to `.9997`. The logged results indicate that 8,192 overflows wrapped during simulation in the Accumulator data type of the Signed Cumulative Sum block. Similarly, the Accumulator data type of the Unsigned Cumulative Sum block had 16,383 overflows wrap during simulation.

To get more information about each of these data types, highlight them in the **Contents** pane, and click the **Show details for selected result** button ( )

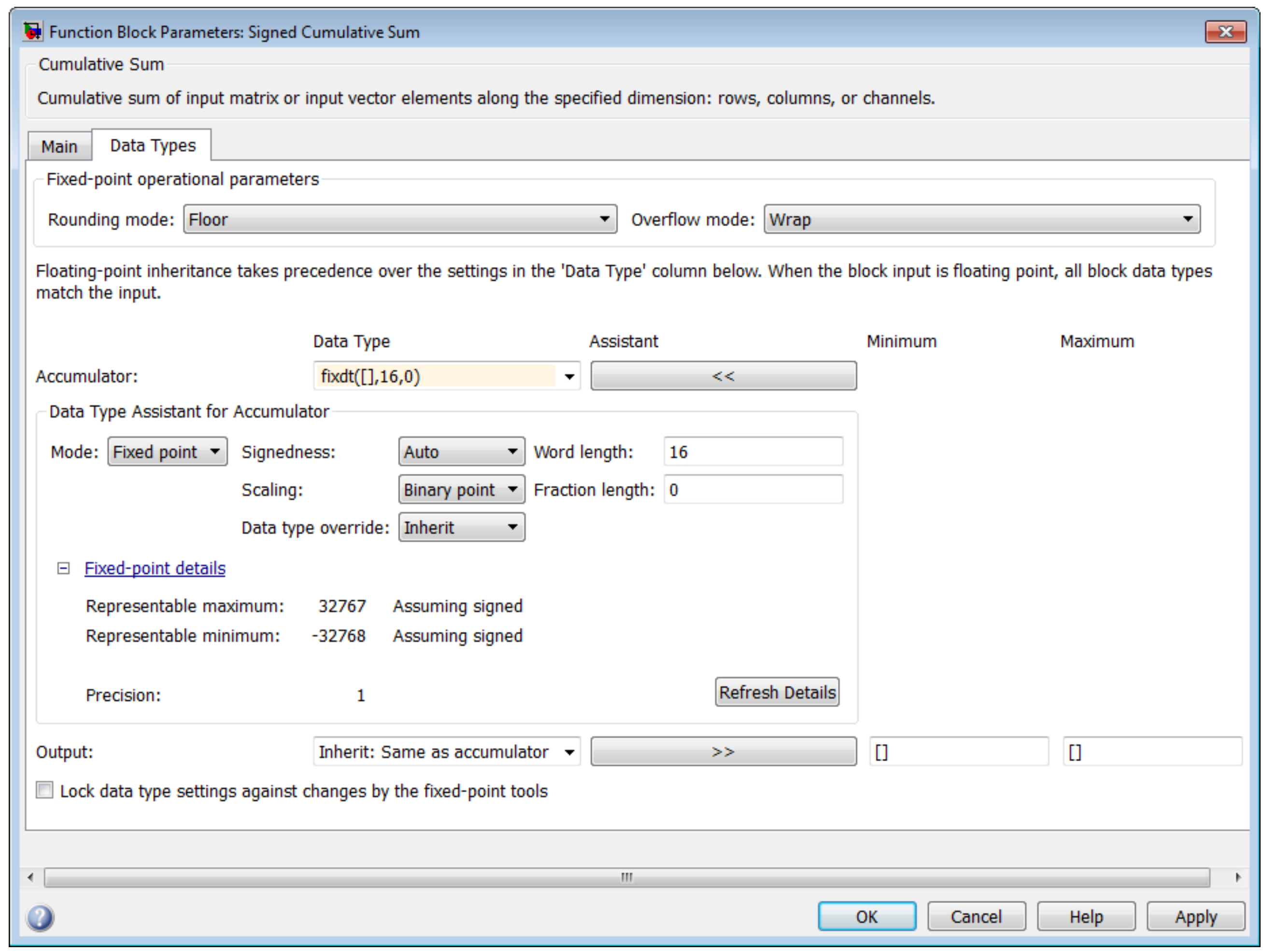
6 Assume a target hardware that supports 32-bit integers, and set the Accumulator word length in both Cumulative Sum blocks to `32`. To do so, perform the following steps:

   a Right-click the `Signed Cumulative Sum:  Accumulator` row in the Fixed-Point Tool pane, and select **Highlight Block In Model**.

   b Double-click the block in the model, and select the **Data Types** pane of the dialog box.

   c Open the **Data Type Assistant for Accumulator** by clicking the Assistant button (>>) in the Accumulator data type row.

   d Set the **Mode** to `Fixed Point`. To see the representable range of the current specified data type, click the **Fixed-point details** link. The tool displays the representable maximum and representable minimum values for the current data type.



e Change the **Word length** to 32, and click the **Refresh details** button in the **Fixed-point details** section to see the updated representable range. When you change the value of the **Word length** parameter, the data type string in the **Data Type** edit box automatically updates.

f Click **OK** on the block dialog box to save your changes and close the window.

g Set the word length of the Accumulator data type of the Unsigned Cumulative Sum block to 32 bits. You can do so in one of two ways:

- Type the data type string `fixdt([],32,0)` directly into **Data Type** edit box for the Accumulator data type parameter.
- Perform the same steps you used to set the word length of the Accumulator data type of the Signed Cumulative Sum block to `32` bits.

7. To verify your changes in word length and check for overflow, rerun your model. To do so, click the **Simulate** button in the Fixed-Point Tool.

   The **Contents** pane of the Fixed-Point Tool updates, and you can see that no overflows occurred in the most recent simulation. However, you can also see that the **SimMin** and **SimMax** values range from `0` to `0`. This underflow happens because the fraction length of the Accumulator data type is too small. The **SpecifiedDT** cannot represent the precision of the data values. The following sections discuss how to find a floating-point benchmark and use the Fixed-Point Tool to propose fraction lengths.

Use Data Type Override to Find a Floating-Point Benchmark

The **Data type override** feature of the Fixed-Point tool allows you to override the data types specified in your model with floating-point types. Running your model in `Double` override mode gives you a reference range to help you select appropriate fraction lengths for your fixed-point data types. To do so, perform the following steps:

1. Open the Fixed-Point Tool and set **Data type override** to `Double`.
2. Run your model by clicking the **Run simulation and store active results** button.
3. Examine the results in the **Contents** pane of the Fixed-Point Tool. Because you ran the model in `Double` override mode, you get an accurate, idealized representation of the simulation minimums and maximums. These values appear in the **SimMin** and **SimMax** parameters.
4. Now that you have an accurate reference representation of the simulation minimum and maximum values, you can more easily choose appropriate fraction lengths. Before making these choices, save your active results to reference so you can use them as your floating-point benchmark. To do so, select **Results > Move Active Results To Reference** from the Fixed-Point

Tool menu. The status displayed in the **Run** column changes from `Active` to `Reference` for all signals in your model.

Use the Fixed-Point Tool to Propose Fraction Lengths

Now that you have your `Double` override results saved as a floating-point reference, you are ready to propose fraction lengths.

1. To propose fraction lengths for your data types, you must have a set of `Active` results available in the Fixed-Point Tool. To produce an active set of results, simply rerun your model. The tool now displays both the `Active` results and the `Reference` results for each signal.

2. Select the **Use simulation min/max if design min/max is not available** check box. You did not specify any design minimums or maximums for the data types in this model. Thus, the tool uses the logged information to compute and propose fraction lengths. For information on specifying design minimums and maximums, see "Signal Ranges" in the Simulink documentation.

3. Click the **Propose fraction lengths** button (). The tool populates the proposed data types in the **ProposedDT** column of the **Contents** pane. The corresponding proposed minimums and maximums are displayed in the **ProposedMin** and **ProposedMax** columns.

Examine the Results and Accept the Proposed Scaling

Before accepting the fraction lengths proposed by the Fixed-Point Tool, it is important to look at the details of that data type. Doing so allows you to see how much of your data the suggested data type can represent. To examine the suggested data types and accept the proposed scaling, perform the following steps:

1. In the **Contents** pane of the Fixed-Point Tool, you can see the proposed fraction lengths for the data types in your model.

   - The proposed fraction length for the Accumulator data type of both the Signed and Unsigned Cumulative Sum blocks is 17 bits.

   - To get more details about the proposed scaling for a particular data type, highlight the data type in the **Contents** pane of the Fixed-Point Tool.

- Open the Autoscale Information window for the highlighted data type by clicking the **Show autoscale information for the selected result** button ( ).

2 When the Autoscale Information window opens, check the **Value** and **Percent Proposed Representable** columns for the **Simulation Minimum** and **Simulation Maximum** parameters. You can see that the proposed data type can represent 100% of the range of simulation data.

3 To accept the proposed data types, select the check box in the **Accept** column for each data type whose proposed scaling you want to keep. Then, click the **Apply accepted fraction lengths** button ( ). The tool updates the specified data types on the block dialog boxes and the **SpecifiedDT** column in the **Contents** pane.

4 To verify the newly accepted scaling, set the **Data type override** parameter back to **Use local settings**, and run the model. Looking at **Contents** pane of the Fixed-Point Tool, you can see the following details:

   - The **SimMin** and **SimMax** values of the `Active` run match the **SimMin** and **SimMax** values from the floating-point `Reference` run.
   - There are no longer any overflows.
   - The **SimDT** does not match the **SpecifiedDT** for the Accumulator data type of either Cumulative Sum block. This difference occurs because the Cumulative Sum block always inherits its **Signedness** from the input signal and only allows you to specify a **Signedness** of `Auto`. Therefore, the **SpecifiedDT** for both Accumulator data types is `fixdt([],32,17)`. However, because the Signed Cumulative Sum block has a signed input signal, the **SimDT** for the Accumulator parameter of that block is also signed (`fixdt(1,32,17)`). Similarly, the **SimDT** for the Accumulator parameter of the Unsigned Cumulative Sum block inherits its **Signedness** from its input signal and thus is unsigned (`fixdt(0,32,17)`).

9

Code Generation

- “Code Generation with System Objects” on page 9-2
- “Functions that Generate Code” on page 9-7
- “Shared Library Dependencies” on page 9-8
- “Accelerating Simulink Models” on page 9-9

Code Generation with System Objects

The following signal processing System objects support code generation in MATLAB via the `codegen` function. To use the `codegen` function, you must have a MATLAB Coder license. See "Use System Objects for Code Generation from MATLAB" for more information.

Supported Computer Vision System Toolbox System Objects

Object	Description
Analysis & Enhancement	
`vision.BoundaryTracer`	Trace object boundaries in binary images
`vision.ContrastAdjuster`	Adjust image contrast by linear scaling
`vision.Deinterlacer`	Remove motion artifacts by deinterlacing input video signal
`vision.EdgeDetector`	Find edges of objects in images
`vision.ForegroundDetector`	Detect foreground using Gaussian Mixture Models. This object supports tunable properties in code generation.
`vision.HistogramEqualizer`	Enhance contrast of images using histogram equalization
`vision.TemplateMatcher`	Perform template matching by shifting template over image
Conversions	
`vision.Autothresholder`	Convert intensity image to binary image
`vision.ChromaResampler`	Downsample or upsample chrominance components of images
`vision.ColorSpaceConverter`	Convert color information between color spaces
`vision.DemosaicInterpolator`	Demosaic Bayer's format images
`vision.GammaCorrector`	Apply or remove gamma correction from images or video streams

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
`vision.ImageComplementer`	Compute complement of pixel values in binary, intensity, or RGB images
`vision.ImageDataTypeConverter`	Convert and scale input image to specified output data type
Feature Detection, Extraction, and Matching	
`vision.CornerDetector`	Corner metric matrix and corner detector. This object supports tunable properties in code generation.
Filtering	
`vision.Convolver`	Compute 2-D discrete convolution of two input matrices
`vision.ImageFilter`	Perform 2-D FIR filtering of input matrix
`vision.MedianFilter`	2D median filtering
Geometric Transformations	
`vision.GeometricRotator`	Rotate image by specified angle
`vision.GeometricScaler`	Enlarge or shrink image size
`vision.GeometricShearer`	Shift rows or columns of image by linearly varying offset
`vision.GeometricTransformer`	Apply projective or affine transformation to an image
`vision.GeometricTransformEstimator`	Estimate geometric transformation from matching point pairs
`vision.GeometricTranslator`	Translate image in two-dimensional plane using displacement vector
Morphological Operations	

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
`vision.ConnectedComponentLabeler`	Label and count the connected regions in a binary image
`vision.MorphologicalClose`	Perform morphological closing on image
`vision.MorphologicalDilate`	Perform morphological dilation on an image
`vision.MorphologicalErode`	Perform morphological erosion on an image
`vision.MorphologicalOpen`	Perform morphological opening on an image
Object Detection	
`vision.HistogramBasedTracker`	Track object in video based on histogram. This object supports tunable properties in code generation
Sinks	
`vision.DeployableVideoPlayer`	Send video data to computer screen
`vision.VideoFileWriter`	Write video frames and audio samples to multimedia file
Sources	
`vision.VideoFileReader`	Read video frames and audio samples from compressed multimedia file
Statistics	
`vision.Autocorrelator`	Compute 2-D autocorrelation of input matrix
`vision.BlobAnalysis`	Compute statistics for connected regions in a binary image
`vision.Crosscorrelator`	Compute 2-D cross-correlation of two input matrices
`vision.Histogram`	Generate histogram of each input matrix

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
`vision.LocalMaximaFinder`	Find local maxima in matrices
`vision.Maximum`	Find maximum values in input or sequence of inputs
`vision.Mean`	Find mean value of input or sequence of inputs
`vision.Median`	Find median values in an input
`vision.Minimum`	Find minimum values in input or sequence of inputs
`vision.PSNR`	Compute peak signal-to-noise ratio (PSNR) between images
`vision.StandardDeviation`	Find standard deviation of input or sequence of inputs
`vision.Variance`	Find variance values in an input or sequence of inputs
Text & Graphics	
`vision.AlphaBlender`	Combine images, overlay images, or highlight selected pixels
`vision.MarkerInserter`	Draw markers on output image
`vision.ShapeInserter`	Draw rectangles, lines, polygons, or circles on images
`vision.TextInserter`	Draw text on image or video stream
Transforms	
`vision.DCT`	Compute 2-D discrete cosine transform
`vision.FFT`	Two-dimensional discrete Fourier transform
`vision.HoughLines`	Find Cartesian coordinates of lines that are described by rho and theta pairs
`vision.HoughTransform`	Find lines in images via Hough transform
`vision.IDCT`	Compute 2-D inverse discrete cosine transform

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
`vision.IFFT`	Two–dimensional inverse discrete Fourier transform
`vision.Pyramid`	Perform Gaussian pyramid decomposition
Utilities	
`vision.ImagePadder`	Pad or crop input image along its rows, columns, or both

Functions that Generate Code

The following Computer Vision System Toolbox functions support code generation in MATLAB. See“About MATLAB Coder” for more information.

Function	Description
`epipolarLine`	Compute epipolar lines for stereo images
`estimateFundamentalMatrix`	Estimate fundamental matrix from corresponding points in stereo image
`estimateUncalibratedRectification`	Uncalibrated stereo rectification
`extractFeatures`	Extract interest point descriptors
`isEpipoleInImage`	Determine whether image contains epipole
`lineToBorderPoints`	Intersection points of lines in image and image border
`matchFeatures`	Find matching image features

Shared Library Dependencies

In general, the code you generate from Computer Vision System Toolbox blocks is portable ANSI® C code. After you generate the code, you can deploy it on another machine. For more information on how to do so, see “Relocate Code to Another Development Environment” in the Simulink Coder documentation.

There are a few Computer Vision System Toolbox blocks that generate code with limited portability. These blocks use precompiled shared libraries, such as DLLs, to support I/O for specific types of devices and file formats. To find out which blocks use precompiled shared libraries, open the Computer Vision System Toolbox Block Support Table. You can identify blocks that use precompiled shared libraries by checking the footnotes listed in the **Code Generation Support** column of the table. All blocks that use shared libraries have the following footnote:

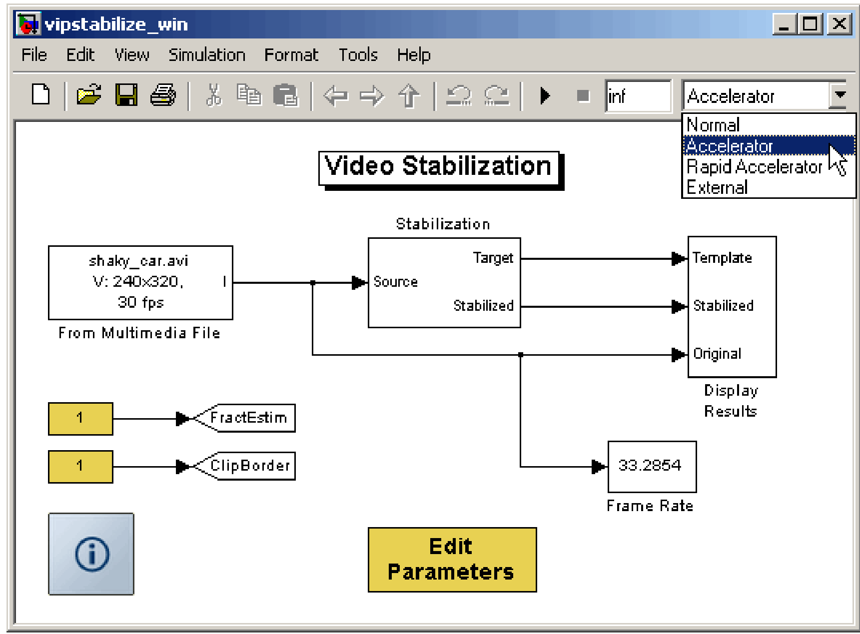
```
Host computer only.  Excludes Real-Time Windows (RTWIN) target.
```

Simulink Coder provides functions to help you set up and manage the build information for your models. For example, one of the “Build Information ” functions that Simulink Coder provides is `getNonBuildFiles`. This function allows you to identify the shared libraries required by blocks in your model. If your model contains any blocks that use precompiled shared libraries, you can install those libraries on the target system. The folder that you install the shared libraries in must be on the system path. The target system does not need to have MATLAB installed, but it does need to be supported by MATLAB.

Accelerating Simulink Models

The Simulink software offer `Accelerator` and `Rapid Accelerator` simulation modes that remove much of the computational overhead required by Simulink models. These modes compile target code of your model. Through this method, the Simulink environment can achieve substantial performance improvements for larger models. The performance gains are tied to the size and complexity of your model. Therefore, large models that contain Computer Vision System Toolbox blocks run faster in `Rapid Accelerator` or `Accelerator` mode.

To change between `Rapid Accelerator`, `Accelerator`, and `Normal` mode, use the drop-down list at the top of the model window.



For more information on the accelerator modes in Simulink, see “Accelerating Models” in the Simulink User’s Guide.

Define New System Objects

- “Define Basic System Objects” on page 10-2
- “Change Number of Step Method Inputs or Outputs” on page 10-4
- “Validate Property and Input Values” on page 10-7
- “Initialize Properties and Setup One-Time Calculations” on page 10-10
- “Set Property Values at Construction from Name-Value Pairs” on page 10-13
- “Reset Algorithm State” on page 10-16
- “Define Property Attributes” on page 10-18
- “Hide Inactive Properties” on page 10-21
- “Limit Property Values to a Finite Set of Strings” on page 10-23
- “Process Tuned Properties” on page 10-26
- “Release System Object Resources” on page 10-28
- “Define Composite System Objects” on page 10-30
- “Define Finite Source Objects” on page 10-34
- “Methods Timing” on page 10-36

Define Basic System Objects

This example shows the structure of a basic System object that increments a number by one.

The class definition file contains the minimum elements required to define a System object.

Create the Class Definition File

1 Create a MATLAB file named `AddOne.m` to contain the definition of your System object.

```
edit AddOne.m
```

2 Subclass your object from `matlab.System`. Insert this line as the first line of your file.

```
classdef AddOne < matlab.System
```

3 Add the `stepImpl` method, which contains the algorithm that runs when users call the `step` method on your object. You always set the `stepImpl` method access to `protected` because it is an internal method that users do not directly call or run.

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle.

In this example, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables from occurring.

By default, the number of inputs and outputs are both one. To change the number of inputs or outputs, use the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively.

```
methods (Access=protected)
   function y = stepImpl(~, x)
      y = x + 1;
   end
end
```

Note Instead of manually creating your class definition file, you can use **File > New > System Object** to open a sample System object file in the editor. You then can edit that file, using it as guideline, to create your own System object.

Complete Class Definition File for Basic System Object

```
classdef AddOne < matlab.System
%ADDONE Compute an output value one greater than the input value

  % All methods occur inside a methods declaration.
  % The stepImpl method has protected access
  methods (Access=protected)

    function y = stepImpl(~,x)
      y = x + 1;
    end
  end
end
```

See Also

`stepImpl` | `getNumInputsImpl` | `getNumOutputsImpl` | `matlab.System` |

Related Examples

- “Change Number of Step Method Inputs or Outputs” on page 10-4

More About

- “Process Data using System Objects”
- *Object-Oriented Programming*
- “Class Definition—Syntax Reference”
- “Methods — Defining Class Operations”

Change Number of Step Method Inputs or Outputs

This example shows how to specify two inputs and two outputs for the `step` method.

If you do not specify the `getNumInputsImpl` and `getNumOutputsImpl` methods, the object uses the default values of 1 input and 1 output. In this case, the user must provide an input to the `step` method.

To specify no inputs, you must explicitly set the number of inputs to 0 using the `getNumInputsImpl` method. To specify no outputs, you must explicitly return 0 in the `getNumOutputsImpl` method.

You always set the `getNumInputsImpl` and `getNumOutputsImpl` methods access to `protected` because they are internal methods that users do not directly call or run.

All methods, except static methods, expect the System object handle as the first input argument. You can use any name for your System object handle. In this example, instead of passing in the object handle, ~ is used to indicate that the object handle is not used in the function. Using ~ instead of an object handle prevents warnings about unused variables from occurring.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to accept a second input and provide a second output.

```
methods (Access=protected)
   function [y1,y2] = stepImpl(~,x1,x2)
      y1 = x1 + 1
      y2 = x2 + 1;
   end
end
```

Update the Associated Methods

Use `getNumInputsImpl` and `getNumOutputsImpl` to specify two inputs and two outputs, respectively.

```
methods (Access=protected)
   function numIn = getNumInputsImpl(~)
      numIn = 2;
   end

   function numOut = getNumOutputsImpl(~)
      numOut = 2;
   end
end
```

Complete Class Definition File with Multiple Inputs and Outputs

```
classdef AddOne < matlab.System
%ADDONE Compute output values two greater than the input values

  % All methods occur inside a methods declaration.
  % The stepImpl method has protected access
  methods(Access=protected)

    function [y1 y2] = stepImpl(~,x1,x2)
      y1 = x1 + 1;
      y2 = x2 + 1;
    end

    % getNumInputsImpl method calculates number of inputs
    function num = getNumInputsImpl(~)
      num = 2;
    end

    % getNumOutputsImpl method calculates number of outputs
    function num = getNumOutputsImpl(~)
      num = 2;
    end
  end
end
```

See Also

`getNumInputsImpl` | `getNumOutputsImpl` |

Related Examples

- “Validate Property and Input Values” on page 10-7
- “Define Basic System Objects” on page 10-2

More About

- “Class Definition—Syntax Reference”
- “Methods — Defining Class Operations”

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

You use the `validateInputsImpl` and `validatePropertiesImpl` methods to perform this validation

Note All inputs default to variable-size inputs. See "Change System Object Input Complexity or Dimensions" for more information.

Validate Properties

This example shows how to validate the value of a single property using `set.`*`PropertyName`* syntax. In this case, the *`PropertyName`* is `Increment`.

```
methods
   % Validate the properties of the object
   function set.Increment(obj,val)
       if val >= 10
         error('The increment value must be less than 10');
       end
       obj.Increment = val;
   end
 end
```

This example shows how to validate the value of two interdependent properties using the `validatePropertiesImpl` method. In this case, the `UseIncrement` property value must be `true` and the `WrapValue` property value must be less than the `Increment` property value.

```
methods (Access=protected)
   function validatePropertiesImpl(obj)
       if obj.UseIncrement && obj.WrapValue < obj.Increment
         error('Wrap value must be less than increment value');
       end
   end
```

```
 end
```

Validate Inputs

This example shows how to validate that the first input is a numeric value.

```
methods (Access=protected)
   function validateInputsImpl(~,x)
      if ~isnumeric(x)
         error('Input must be numeric');
      end
   end
end
```

Complete Class Definition File with Property and Input Validation

```
classdef AddOne < matlab.System
%ADDONE Compute an output value by incrementing the input value

  % All properties occur inside a properties declaration.
  % These properties have public access (the default)
  properties (Logical)
    UseIncrement = true
  end

  properties (PositiveInteger)
    Increment = 1
    WrapValue = 10
  end

  methods
    % Validate the properties of the object
    function set.Increment(obj, val)
        if val >= 10
          error('The increment value must be less than 10');
        end
        obj.Increment = val;
    end
  end

  methods (Access=protected)
```

```
    function validatePropertiesImpl(obj)
        if obj.UseIncrement && obj.WrapValue < obj.Increment
          error('Wrap value must be less than increment value');
        end
    end

    % Validate the inputs to the object
    function validateInputsImpl(~,x)
        if ~isnumeric(x)
          error('Input must be numeric');
        end
    end

    function out = stepImpl(obj,in)
      if obj.UseIncrement
        y = x + obj.Increment;
      else
        y = x + 1;
      end
    end
  end
end
```

See Also

`validateInputsImpl` | `validatePropertiesImpl` |

Related Examples

- "Define Basic System Objects" on page 10-2
- "Validate Property and Input Values" on page 10-7

More About

- "Methods Timing" on page 10-36
- "Property Set Methods"

Initialize Properties and Setup One-Time Calculations

This example shows how to write code to initialize and set up a System object.

In this example, you allocate file resources by opening the file so the System object can write to that file. You do these initialization tasks one time during setup, rather than every time you call the step method.

Define Properties to Initialize

In this example, you define the public `Filename` property and specify the value of that property as the nontunable string, `default.bin`. Users cannot change *nontunable* properties after the `setup` method has been called. Refer to the Methods Timing section for more information.

```
properties (Nontunable)
  Filename ='default.bin'
end
```

Users cannot access *private* properties directly, but only through methods of the System object. In this example, you define the `pFileID` property as a private property. You also define this property as *hidden* to indicate it is an internal property that never displays to the user.

```
properties (Hidden,Access=private)
  pFileID;
end
```

Define Setup

You use the `setupImpl` method to perform setup and initialization tasks. You should include code in the `setupImpl` method that you want to execute one time only. The `setupImpl` method is called once during the first call to the `step` method. In this example, you allocate file resources by opening the file for writing binary data.

```
methods
  function setupImpl(obj, data)
    obj.pFileID = fopen(obj.Filename, 'wb');
    if obj.pFileID < 0
       error('Opening the file failed');
```

```
    end
  end
end
```

Although not part of setup, you should close files when your code is done using them. You use the `releaseImpl` method to release resources.

Complete Class Definition File with Initialization and Setup

```
classdef MyFile < matlab.System
%MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step  method has been called.
    properties (Nontunable)
        Filename ='default.bin' % the name of the file to create
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access=private)
        pFileID; % The identifier of the file to open
    end

    methods (Access=protected)
        % In setup allocate any resources, which in this case
        % means opening the file.
        function setupImpl(obj,data)
            obj.pFileID = fopen(obj.Filename, 'wb');
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end

        % This System object writes the input to the file.
        function stepImpl(obj,data)
            fwrite(obj.pFileID, data);
        end

        % Use release to close the file to prevent the
```

```
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end

        % You indicate that no outputs are provided by returning
        % zero from getNumOutputsImpl
        function numOutputs = getNumOutputsImpl(~)
            numOutputs = 0;
        end
    end
end
```

See Also

`setupImpl` | `releaseImpl` | `stepImpl` |

Related Examples

- “Release System Object Resources” on page 10-28
- “Define Property Attributes” on page 10-18

More About

- “Methods Timing” on page 10-36

Set Property Values at Construction from Name-Value Pairs

This example shows how to define a System object constructor and allow it to accept name-value property pairs as input.

Set Properties to Use Name-Value Pair Input

Define the System object constructor, which is a method that has the same name as the class (`MyFile` in this example). Within that method, you use the `setProperties` method to make all public properties available for input when the user constructs the object. `nargin` is a MATLAB function that determines the number of input arguments. `varargin` indicates all of the object's public properties.

```
methods
   function obj = MyFile(varargin)
      setProperties(obj,nargin,varargin{:});
   end
end
```

Complete Class Definition File with Constructor Setup

```
classdef MyFile < matlab.System
%MyFile write numbers to a file

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        Filename ='default.bin' % the name of the file to create
        Access = 'wb' % The file access string (write, binary)
    end

    % These properties are private. Customers can only access
    % these properties through methods on this object
    properties (Hidden,Access=private)
        pFileID; % The identifier of the file to open
    end

    methods
```

```
        % You call setProperties in the constructor to let
        % a user specify public properties of object as
        % name-value pairs.
        function obj = MyFile(varargin)
          setProperties(obj,nargin,varargin{:});
        end
    end

    methods (Access=protected)
        % In setup allocate any resources, which in this case is
        % opening the file.
        function setupImpl(obj, ~)
            obj.pFileID = fopen(obj.Filename,obj.Access);
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        end

        % This System object writes the input to the file.
        function stepImpl(obj, data)
            fwrite(obj.pFileID, data);
        end

        % Use release to close the file to prevent the
        % file handle from being left open.
        function releaseImpl(obj)
            fclose(obj.pFileID);
        end
        % You indicate that no outputs are provided by returning
        % zero from getNumOutputsImpl
        function numOutputs = getNumOutputsImpl(~)
            numOutputs = 0;
        end
    end
end
```

See Also

`narginsetProperties` |

Related Examples

- "Define Property Attributes" on page 10-18
- "Release System Object Resources" on page 10-28

Reset Algorithm State

This example shows how to reset an object state.

Reset Counter to Zero

`pCount` is an internal counter property of the System object `obj`. The user calls the `reset` method, which calls the `resetImpl` method. In this example , `pCount` resets to 0. See "Methods Timing" on page 10-36 for more information.

Note When resetting an object's state, make sure you reset the size, complexity, and data type correctly.

```
methods (Access=protected)
   function resetImpl(obj)
      obj.pCount = 0;
   end
end
```

Complete Class Definition File with State Reset

```
classdef Counter < matlab.System
%Counter System object that increments a counter

   properties(Access = private)
      pCount
   end

   methods (Access=protected)
      % In step, increment the counter and return
      % its value as an output
      function c = stepImpl(obj)
          obj.pCount = obj.pCount + 1;
          c = obj.pCount;
      end

      % Reset the counter to zero.
      function resetImpl(obj)
```

```
            obj.pCount = 0;
         end

         % The step method takes no inputs
         function numIn = getNumInputsImpl(~)
           numIn = 0;
         end
      end
   end
   end
```

See Also `resetImpl` |

More About

- “Methods Timing” on page 10-36

Define Property Attributes

This example shows how to specify property attributes.

Property attributes, which add details to a property, provide a layer of control to your properties. In addition to the MATLAB property attributes, System objects can use these three additional attributes—`nontunable`, `logical`, and `positiveInteger`. To specify multiple attributes, separate them with commas.

Specify Property as Nontunable

System object users cannot change *nontunable* properties after the `setup` or `step` method has been called. In this example, you define the `InitialValue` property, and set its value to 0.

```
properties (Nontunable)
   InitialValue = 0;
end
```

Specify Property as Logical

Logical properties have the value, `true` or `false`. System object users can enter `1` or `0`, but the value displays as `true` or `false`, respectively. You can use sparse logical values, but they must be scalar values. In this example, the `Increment` property indicates whether to increase the counter. By default, `Increment` is tunable property.

```
properties (Logical)
   Increment = true
end
```

Specify Property as Positive Integer

In this example, the private property `pCount` is constrained to accept only real, positive integers. You cannot use sparse values.

```
properties (PositiveInteger)
   Count
end
```

Specify Property as DiscreteState

If your algorithm uses properties that hold state, you can assign those properties the `DiscreteState` attribute . Properties with this attribute display their state values when users call `getDiscreteStateImpl` via the `getDiscreteState` method. The following restrictions apply to a property with the `DiscreteState` attribute,

- Numeric, logical, or fi value
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`, or `Transient`.
- No default value
- Not publicly settable
- `GetAccess=Public` by default
- Value set only using the `setupImpl` method or when the System object is locked during `resetImpl` or `stepImpl`

In this example, you define the `Count` property.

```
properties (DiscreteState)
   Count;
end
```

Complete Class Definition File with Property Attributes

```
classdef Counter < matlab.System
%Counter Increment a counter starting at an initial value

  % These properties are nontunable. They cannot be changed
  % after the setup or step method has been called.
  properties (Nontunable)
      % The inital value of the counter
      InitialValue = 0
  end

  properties (Logical)
      % Whether to increment the counter
      Increment = true
```

```
  end

  % Count state variable
  properties (DiscreteState, PositiveInteger)
      Count
  end

  methods (Access=protected)
      % In step, increment the counter and return its value
      % as an output
      function c = stepImpl(obj)
          if obj.Increment
              obj.Count = obj.Count + 1;
          end
          c = obj.Count;
      end
      % Setup the Count state variable
      function setupImpl(obj)
          obj.Count = 0;
      end
      % Reset the counter to zero.
      function resetImpl(obj)
          obj.Count = obj.InitialValue;
      end
      % The step method takes no inputs
      function numIn = getNumInputsImpl(~)
          numIn = 0;
      end
  end
end
```

More About

- "Class Attributes"
- "What are System Object Locking and Property Tunability?"
- "Methods Timing" on page 10-36

Hide Inactive Properties

This example shows how to hide the display of a property that is not active for a particular object configuration.

Hide an inactive property

You use the `isInactivePropertyImpl` method to hide a property from displaying. If the `isInactiveProperty` method returns `true` to the property you pass in, then that property does not display.

```
methods (Access=protected)
  function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
      flag = obj.UseRandomInitialValue;
    else
      flag = false;
    end
  end
end
```

Complete Class Definition File with Hidden Inactive Property

```
classdef Counter < matlab.System
  %Counter Increment a counter

  % These properties are nontunable. They cannot be changed
  % after the setup or step method has been called.
  properties (Nontunable)
    % Allow the user to set the initial value
    UseRandomInitialValue = true
    InitialValue = 0
  end

  % The private count variable, which is tunable by default
  properties (Access=private)
    pCount
  end

  methods (Access=protected)
```

```
      % In step, increment the counter and return its value
      % as an output
      function c = stepImpl(obj)
        obj.pCount = obj.pCount + 1;
        c = obj.pCount;
      end

      %Reset the counter to either a random value or the initial
      % value.
      function resetImpl(obj)
        if obj.UseRandomInitialValue
          obj.pCount = rand();
        else
          obj.pCount = obj.InitialValue;
        end
      end

      % The step method takes no inputs
      function numIn = getNumInputsImpl(~)
        numIn = 0;
      end

      % This method controls visibility of the object's properties
      function flag = isInactivePropertyImpl(obj,propertyName)
        if strcmp(propertyName,'InitialValue')
          flag = obj.UseRandomInitialValue;
        else
          flag = false;
        end
      end
    end
  end
```

See Also

`isInactivePropertyImpl` |

Limit Property Values to a Finite Set of Strings

This example shows how to limit a property to accept only a finite set of string values.

Specify a Set of Valid String Values

Sting sets use two related properties. You first specify the user-visible property name and default string value. Then, you specify the associated hidden property by appending “Set” to the property name. You must use a capital “S” in “Set.”

In the “Set” property, you specify the valid string values as a cell array of the `matlab.system.Stringset` class. This example uses `Color` and `ColorSet` as the associated properties.

```
properties
   Color = 'blue'
end

properties (Hidden,Transient)
   ColorSet = matlab.system.StringSet({'red', 'blue', 'green'});
end
```

Complete Class Definition File with String Set

```
classdef Whiteboard < matlab.System
%Whiteboard Draw lines on a figure window
%
% This System object illustrates the use of StringSets

   properties
     Color = 'blue'
   end

   properties (Hidden,Transient)
     % Let them choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end
```

```
    methods(Access = protected)
      function stepImpl(obj)
          h = Whiteboard.getWhiteboard();
          plot(h, ...
            randn([2,1]), randn([2,1]), ...
            'Color', obj.Color(1));
      end
      function releaseImpl(obj)
          cla(Whiteboard.getWhiteboard());
          hold('on');
      end
      function n = getNumInputsImpl(~)
           n = 0;
      end
      function n = getNumOutputsImpl(~)
          n = 0;
      end
    end

    methods (Static)
      function a = getWhiteboard()
          h = findobj('tag','whiteboard');
          if isempty(h)
            h = figure('tag','whiteboard');
            hold('on');
          end
          a = gca;
      end
    end
end
```

String Set System Object Example

```
%%
% Each call to step draws lines on a whiteboard

%% Construct the System object
hGreenInk = Whiteboard;
hBlueInk  = Whiteboard;
```

```
% Change the color
% Note: Press tab after typing the first single quote to
% display all enumerated values.
hGreenInk.Color = 'green';
hBlueInk.Color  = 'blue';

% Take a few steps
for i=1:3
  hGreenInk.step();
  hBlueInk.step();
end

%% Clear the whiteboard
hBlueInk.release();

%% Display System object used in this example
type('Whiteboard.m');
```

See Also

`matlab.system.StringSet` |

More About

- "Enumerations"

Process Tuned Properties

This example shows how to specify the action to take when a tunable property value changes during simulation.

The `processTunedPropertiesImpl` method is useful for managing actions to prevent duplication. In many cases, changing one of multiple interdependent properties causes an action. With the `processTunedPropertiesImpl` method, you can control when that action is taken so it is not repeated unnecessarily.

Control When a Lookup Table Is Generated

This example of `processTunedPropertiesImpl` causes the `pLookupTable` to be regenerated when either the `NumNotes` or `MiddleC` property changes.

```
methods  (Access = protected)
   function processTunedPropertiesImpl(obj)
      obj.pLookupTable = obj.MiddleC * ...
        (1+log(1:obj.NumNotes)/log(12));
   end
end
```

Complete Class Definition File with Tuned Property Processing

```
classdef TuningFork < matlab.System
  %TuningFork Illustrate the processing of tuned parameters
  %

  properties
    MiddleC = 440
    NumNotes = 12
  end

  properties (Access=private)
    pLookupTable
  end

  methods(Access = protected)
    function resetImpl(obj)
      obj.MiddleC = 440;
```

```
      obj.pLookupTable = obj.MiddleC * ...
         (1+log(1:obj.NumNotes)/log(12));
    end

    function hz = stepImpl(obj, noteShift)
      % A noteShift value of 1 corresponds to obj.MiddleC
      hz = obj.pLookupTable(noteShift);
    end

    function processTunedPropertiesImpl(obj)
      % Generate a lookup table of note frequencies
      obj.pLookupTable = obj.MiddleC * ...
         (1+log(1:obj.NumNotes)/log(12));
    end
  end
end
```

See Also

`processTunedPropertiesImpl` |

Release System Object Resources

This example shows how to release resources allocated and used by the System object. These resources include allocated memory, files used for reading or writing, etc.

Release Memory by Clearing the Object

This method allows you to clear the axes on the Whiteboard figure window while keeping the figure open.

```
methods
   function releaseImpl(obj)
      cla(Whiteboard.getWhiteboard());
      hold('on');
   end
end
```

Complete Class Definition File with Released Resources

```
classdef Whiteboard < matlab.System
%Whiteboard Draw lines on a figure window
%
% This System object illustrates the use of StringSets
%
   properties
     Color = 'blue'
   end

   properties (Hidden)
     % Let them choose a color
     ColorSet = matlab.system.StringSet({'red','blue','green'});
   end

   methods(Access = protected)
     function stepImpl(obj)
         h = Whiteboard.getWhiteboard();
         plot(h, ...
           randn([2,1]), randn([2,1]), ...
           'Color', obj.Color(1));
```

```
      end

      function releaseImpl(obj)
          cla(Whiteboard.getWhiteboard());
          hold('on');
      end

      function n = getNumInputsImpl(~)
          n = 0;
      end
      function n = getNumOutputsImpl(~)
          n = 0;
      end
    end

    methods (Static)
      function a = getWhiteboard()
          h = findobj('tag','whiteboard');
          if isempty(h)
            h = figure('tag','whiteboard');
            hold('on');
          end
          a = gca;
      end
    end
end
```

See Also

`isInactivePropertyImpl` |

Related Examples

- "Initialize Properties and Setup One-Time Calculations" on page 10-10

Define Composite System Objects

This example shows how to define System objects that include other System objects.

This example defines a filter System object from an FIR System object and an IIR System object.

Store System Objects in Properties

To define a System object from other System objects, store those objects in your class definition file as properties. In this example, FIR and IIR are separate System objects defined in their own class-definition files. You use those two objects to calculate the `pFir` and `pIir` property values.

```
properties (Nontunable, Access = private)
   pFir % store the FIR filter
   pIir % store the IIR filter
end

methods
   function obj = Filter(varargin)
      setProperties(obj, nargin, varargin{:});
      obj.pFir = FIR(obj.zero);
      obj.pIir = IIR(obj.pole);
   end
end
```

Complete Class Definition File of Composite System Object

```
classdef Filter < matlab.System
%Filter System object with a single pole and a single zero
%
%   This System object illustrates composition by
%   composing an instance of itself.
%

    properties (Nontunable)
      zero = 0.01
      pole = 0.5
```

```
    end

    properties (Nontunable,Access=private)
      pZero % store the FIR filter
      pPole % store the IIR filter
    end

    methods
        function obj = Filter(varargin)
            setProperties(obj,nargin, varargin{:});
            % Create instances of FIR and IIR as
            % private properties
            obj.pZero = Zero(obj.zero);
            obj.pPole = Pole(obj.pole);
        end
    end

    methods (Access=protected)
        function setupImpl(obj,x)
            setup(obj.pZero,x);
            setup(obj.pPole,x);
        end

        function resetImpl(obj)
            reset(obj.pZero);
            reset(obj.pPole);
        end

        function y = stepImpl(obj,x)
            y = step(obj.pZero,x) + step(obj.pPole,x);
        end
        function releaseImpl(obj)
            release(obj.pZero);
            release(obj.pPole);
        end
    end
end
```

Class Definition File for FIR Component of Filter

```
classdef Pole < matlab.System

  properties
    Den = 1
  end

  properties (Access=private)
    tap = 0
  end

  methods
    function obj = Pole(varargin)
      setProperties(obj,nargin,varargin{:},'Den');
    end
  end

  methods (Access=protected)
    function y = stepImpl(obj,x)
      y = x + obj.tap * obj.Den;
      obj.tap = y;
    end
  end

end
```

Class Definition File for IIR Component of Filter

```
classdef Zero < matlab.System

  properties
    Num = 1
  end

  properties (Access=private)
    tap = 0
  end

  methods
    function obj = Zero(varargin)
      setProperties(obj, nargin,varargin{:},'Num');
```

```
    end
  end

  methods (Access=protected)
    function y = stepImpl(obj,x)
      y = x + obj.tap * obj.Num;
      obj.tap = x;
    end
  end

end
```

See Also

nargin

Define Finite Source Objects

This example shows how to define a System object that performs a specific number of steps or specific number of reads from a file.

Use the FiniteSource Class and Specify End of the Source

1 Subclass from finite source class.

```
classdef RunTwice < matlab.System & ...
   matlab.system.mixin.FiniteSource
```

2 Specify the end of the source with the `isDoneImpl` method. In this example, the source has two iterations.

```
methods (Access = protected)
  function bDone = isDoneImpl(obj)
     bDone = obj.NumSteps==2
  end
```

Complete Class Definition File with Finite Source

```
classdef RunTwice < matlab.System & ...
    matlab.system.mixin.FiniteSource
  %RunTwice System object that runs exactly two times
  %
  properties (Access=private)
    NumSteps
  end

  methods (Access=protected)
    function resetImpl(obj)
      obj.NumSteps = 0;
    end

    function y = stepImpl(obj)
      if ~obj.isDone()
        obj.NumSteps = obj.NumSteps + 1;
        y = obj.NumSteps;
      else
```

```
        out = 0;
      end
    end

    function bDone = isDoneImpl(obj)
      bDone = obj.NumSteps==2;
    end
  end

  methods (Access=protected)
    function n = getNumInputsImpl(~)
      n = 0;
    end
    function n = getNumOutputsImpl(~)
      n = 1;
    end
  end

end
```

See Also

`matlab.system.mixin.FiniteSource` |

More About

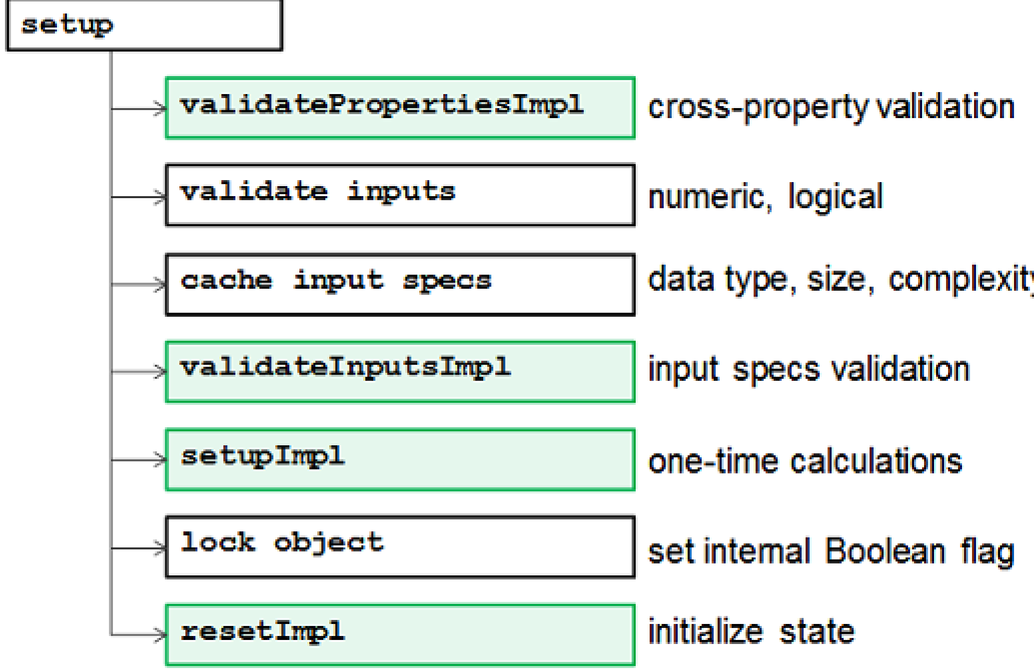
- "Subclassing Multiple Classes"

Methods Timing

In this section...
“Setup Method Call Sequence” on page 10-36
“Step Method Call Sequence” on page 10-37
“Reset Method Call Sequence” on page 10-37
“Release Method Call Sequence” on page 10-38

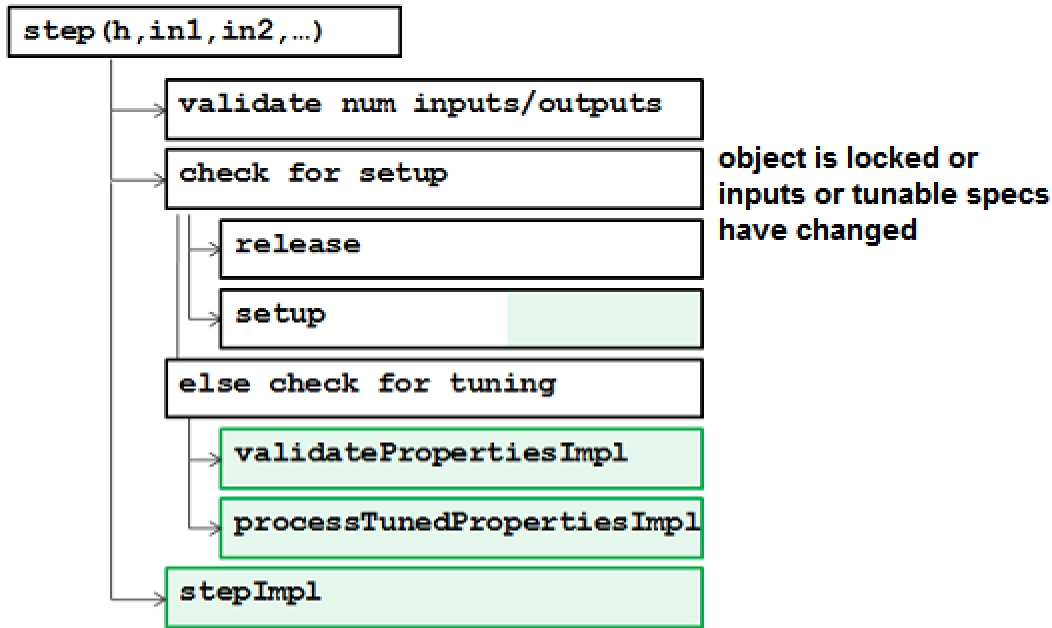
Setup Method Call Sequence

This hierarchy shows the actions performed when you call the `setup` method.



Step Method Call Sequence

This hierarchy shows the actions performed when you call the `step` method.

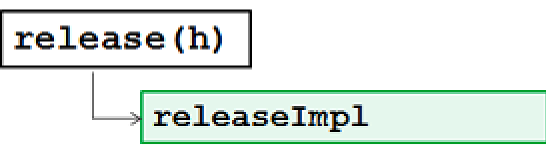


Reset Method Call Sequence

This hierarchy shows the actions performed when you call the `reset` method.

Release Method Call Sequence

This hierarchy shows the actions performed when you call the `release` method.



See Also

`setupImpl` | `stepImpl` | `releaseImpl` | `resetImpl` |

Related Examples

- "Release System Object Resources" on page 10-28
- "Reset Algorithm State" on page 10-16
- "Set Property Values at Construction from Name-Value Pairs" on page 10-13
- "Define Basic System Objects" on page 10-2

More About

- "What are System Object Methods?"
- "The Step Method"
- "Common Methods"

Index

A

`Accelerator` mode 9-9
adding periodic noise to a signal 6-14
adjusting
  intensity image contrast 6-2
  RGB image contrast 6-8
algorithms
  bicubic interpolation 5-24
  bilinear interpolation 5-23
  nearest neighbor interpolation 5-22
angles
  rotation 5-2
annotating
  AVI files 2-30
arithmetic operations
  fixed-point 8-10
artifacts
  in an image 6-14
audio
  exporting to video file 1-15
Autothreshold block
  to perform thresholding 1-31
AVI files
  annotating 2-30
  cropping 1-11
  saving to multiple files 1-11
  splitting 1-11

B

background
  estimation 7-9
  pixels 3-2
batch processing 1-6
bicubic interpolation 5-24
bilinear interpolation 5-23
binary
  conversion from intensity 1-25
  images 1-51
block parameters
  fixed-point 8-25
blurring images 6-30
Boolean matrices 1-51
boundaries
  of objects 3-2
boundary artifacts 6-14
brightening images 7-9

C

casts
  fixed-point 8-16
changing
  image size 5-10
  intensity image contrast 6-2
  RGB image contrast 6-8
chroma components
  of images 1-21
chroma resampling 1-21
chrominance resampling 1-21
code generation
  computer vision objects 9-2
  fixed-point 8-3
color
  definition of 1-49
colormaps 1-51
column-major format 1-50
complex multiplication
  fixed-point 8-13
continuous rotation 5-2
contrast
  increasing 1-5
conventions
  column-major format 1-50
conversion
  intensity to binary 1-25
  R'G'B' to intensity 1-37
correction
  of uneven lighting 7-9
correlation

used in object tracking 4-4
counting objects 7-17
cropping
AVI files 1-11
images 5-16

D

data types 1-49
definition of
intensity and color 1-49
demos
Periodic noise reduction 6-14
detection of
edges 3-2
lines 3-8
downsampling
chroma components 1-21
dynamic range 1-49

E

edge
pixels 3-2
thinning 3-2
edge detection 3-2
electrical interference 6-14
estimation
of image background 7-9
exporting
video files 1-5

F

feature extraction
finding angles between lines 3-14
finding edges 3-2
finding lines 3-8
filtering
median 6-23
finding
angles between lines 3-14
edges of objects 3-2
histograms of images 7-2
lines in images 3-8
fixed point
System object preferences 8-22
fixed point properties
System objects 8-23
fixed-point attributes, specification
at the block level 8-25
at the system level 8-28
fixed-point block parameters
setting 8-25
fixed-point code generation 8-3
fixed-point data types 8-4
addition 8-12
arithmetic operations 8-10
attributes 8-25
casts 8-16
complex multiplication 8-13
concepts 8-4
logging 8-28
modular arithmetic 8-10
multiplication 8-13
overflow handling 8-6
precision 8-6
range 8-6
rounding 8-7
saturation 8-6
scaling 8-5
subtraction 8-12
terminology 8-4
two's complement 8-11
wrapping 8-6
fixed-point development 8-2
Fixed-Point Tool 8-28
frequency distribution
of elements in an image 7-2
`fspecial` function 6-30

G

gradient components
- of images 3-2

H

histograms
- of images 7-2

I

image data
- storage of 1-50

image rotation 5-2
image sequence processing 1-6
image types 1-50
images
- binary 1-51
- boundary artifacts 6-14
- brightening 7-9
- correcting for uneven lighting 7-9
- counting objects in 7-17
- cropping 5-16
- finding angles between lines 3-14
- finding edges in 3-2
- finding histograms of 7-2
- finding lines in 3-8
- gradient components 3-2
- intensity 1-51
- intensity to binary conversion 1-25
- labeling objects in 7-17
- lightening 7-9
- noisy 6-23
- periodic noise removal 6-14
- removing salt and pepper noise 6-23
- resizing of 5-10
- RGB 1-51
- rotation of 5-2
- sharpening and blurring 6-30
- true-color 1-51
- types of 1-50

importing
- multimedia files 1-2

improvement
- of performance 9-9

increasing video contrast 1-5
inherit via internal rule 8-29
intensity
- conversion from R'G'B' 1-37
- conversion to binary 1-25
- definition of 1-49
- images 1-51

intensity images
- adjusting the contrast of 6-2

interference
- electrical 6-14

interpolation
- bicubic 5-24
- bilinear 5-23
- examples 5-22
- nearest neighbor 5-22
- overview 5-22

irregular illumination 7-9

L

labeling objects 7-17
lightening images 7-9
location of
- lines 3-8
- object edges 3-2
- objects in an image 4-4

logging
- fixed-point data types 8-28

luma components
- applying highpass filter 6-30
- applying lowpass filter 6-30
- of images 1-21

luminance 1-21

M

median filtering 6-23
methods
    interpolation 5-22
    thresholding 7-9
modes
    `Normal` and `Accelerator` 9-9
modular arithmetic 8-10
morphology
    opening 7-17
    STREL object 7-17
multimedia files
    exporting 1-5
    importing 1-2
    viewing 1-2
multiplication
    fixed-point 8-13

N

nearest neighbor interpolation 5-22
noise
    adding to a signal 6-14
noise removal
    periodic 6-14
    salt and pepper 6-23
nonuniform illumination
    correcting for 7-9
`Normal` mode 9-9

O

object boundaries 3-2
object tracking
    using correlation 4-4
objects
    delineating 7-9
    location of 4-4
opening 7-17
operations
    thresholding 1-26
overflow handling 8-6
overview of
    interpolation 5-22

P

padding 6-14
performance
    improving 9-9
periodic noise
    removal 6-14
precision
    fixed-point data types 8-6
preferences 8-22

R

R’B’G’
    conversion to intensity 1-37
range
    fixed-point data types 8-6
reception
    of an RGB image 1-21
reduction
    of image size 5-10
region of interest
    cropping to 5-16
    visualizing 4-4
relational operators
    to perform thresholding 1-26
removal of
    periodic noise 6-14
    salt and pepper noise 6-23
resampling
    chroma 1-21
resizing
    images 5-10
RGB images 1-51
    adjusting the contrast of 6-8

rotation
continual 5-2
of an image 5-2
rounding
fixed-point data types 8-7

S

salt and pepper noise removal 6-23
saturation 8-6
saving
to multiple AVI files 1-11
scaling 1-49 8-5
sectioning
AVI files 1-11
sequence
of images 1-6
sharpening images 6-30
shrinking
image size 5-10
Sobel kernel 3-2
splitting
AVI files 1-11
storage of image data 1-50
STREL object 7-17
System object
fixed point
Computer Vision System Toolbox™
objects 8-21
preferences 8-22
system-level settings
fixed-point 8-28

T

techniques
thresholding 7-9
thresholding operation 1-26
with uneven lighting 1-31
thresholding techniques 7-9
tracking
of an object 4-4
transmission
of an RGB image 1-21
trimming
images 5-16
true-color images 1-51
two's complement 8-11
types of images 1-50

U

uneven lighting
correcting for 7-9

V

video
annotating AVI files with video frame
numbers 2-30
exporting from video file 1-5
importing from multimedia file 1-2
increasing the contrast of 1-5
interpretation of 1-49
video files
exporting audio and video 1-15
viewing
multimedia files 1-2
video files 1-2

W

wrapping
fixed-point data types 8-6