

Computer Vision System Toolbox™

User's Guide

R2012b

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)
September 2012	Online only	Revised for Version 5.1 (Release R2012b)

Input, Output, and Conversions

1

Export to Video Files	1-2
Setting Block Parameters for this Example	1-3
Configuration Parameters	1-3
Import from Video Files	1-4
Setting Block Parameters for this Example	1-5
Configuration Parameters	1-6
Batch Process Image Files	1-7
Configuration Parameters	1-8
Display a Sequence of Images	1-9
Pre-loading Code	1-10
Configuration Parameters	1-11
Partition Video Frames to Multiple Image Files	1-12
Setting Block Parameters for this Example	1-12
Using the Enabled Subsystem Block	1-15
Configuration Parameters	1-17
Combine Video and Audio Streams	1-18
Setting Up the Video Input Block	1-18
Setting Up the Audio Input Block	1-18
Setting Up the Output Block	1-19
Configuration Parameters	1-19
Import MATLAB Workspace Variables	1-20
Transmit Audio and Video Content Over Network	1-22
Resample Image Chroma	1-24
Setting Block Parameters for This Example	1-26

Configuration Parameters	1-28
Convert Intensity to Binary Images	1-29
Thresholding Intensity Images Using Relational Operators	1-29
Thresholding Intensity Images Using the Autothreshold Block	1-34
Convert R'G'B' to Intensity Images	1-41
Process Multidimensional Color Video Signals	1-46
Data Formats	1-51
Video Formats	1-51
Video Data Stored in Column-Major Format	1-52
Image Formats	1-52

Display and Graphics

2

Display	2-2
View Streaming Video in MATLAB	2-2
Preview Video in MATLAB using MPlay Function	2-2
View Video with Simulink Blocks	2-3
View Video with MPlay	2-4
MPlay	2-7
Graphics	2-25
Abandoned Object Detection	2-25
Abandoned Object Detection	2-29
Annotate Video Files with Frame Numbers	2-35
Draw Shapes and Lines	2-37

Registration and Stereo Vision

3

Computer Vision Workflow	3-2
Feature Detection, Extraction, and Matching	3-3
Detect Edges in Images	3-3
Detect Lines in Images	3-11
Video Stabilization Using Point Feature Matching	3-15
Video Stabilization	3-28
Measure Angle Between Lines	3-33
Stereo Image Rectification	3-42
Stereo Vision	3-54
Find Fundamental Matrix Describing Epipolar Geometry	3-71

Motion Estimation and Tracking

4

Tracking Cars Using Gaussian Mixture Models	4-2
Video Mosaicking	4-7
Video Mosaicking	4-13
Track an Object Using Correlation	4-19
Panorama Creation	4-25

Geometric Transformations

5

Rotate an Image	5-2
Resize an Image	5-8
Crop an Image	5-12
Interpolation Methods	5-17
Nearest Neighbor Interpolation	5-17
Bilinear Interpolation	5-18
Bicubic Interpolation	5-19
Video Stabilization	5-21
Video Stabilization	5-26

Filters, Transforms, and Enhancements

6

Adjust the Contrast of Intensity Images	6-2
Adjust the Contrast of Color Images	6-7
Remove Salt and Pepper Noise from Images	6-12
Sharpen an Image	6-18

Statistics and Morphological Operations

7

Find the Histogram of an Image	7-2
Correct Nonuniform Illumination	7-8
Count Objects in an Image	7-15

Fixed-Point Design

8

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
Specify Data Types for Fixed-Point Blocks	8-40

Code Generation

9

Code Generation with System Objects	9-2
Functions and Classes that Generate Code	9-7
Shared Library Dependencies	9-9
Accelerating Simulink Models	9-10

Define New System Objects

10

Define Basic System Objects	10-2
Change Number of Step Inputs or Outputs	10-4
Validate Property and Input Values	10-6
Initialize Properties and Setup One-Time Calculations	10-9
Set Property Values at Construction Time	10-12
Reset Algorithm State	10-15
Define Property Attributes	10-17

Hide Inactive Properties	10-21
Limit Property Values to Finite String Set	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
Save System Object	10-36
Load System Object	10-39
Methods Timing	10-42
Setup Method Call Sequence	10-42
Step Method Call Sequence	10-43
Reset Method Call Sequence	10-43
Release Method Call Sequence	10-44

Index



Input, Output, and Conversions

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

- “Export to Video Files” on page 1-2
- “Import from Video Files” on page 1-4
- “Batch Process Image Files” on page 1-7
- “Display a Sequence of Images” on page 1-9
- “Partition Video Frames to Multiple Image Files” on page 1-12
- “Combine Video and Audio Streams” on page 1-18
- “Import MATLAB Workspace Variables” on page 1-20
- “Transmit Audio and Video Content Over Network” on page 1-22
- “Resample Image Chroma” on page 1-24
- “Convert Intensity to Binary Images” on page 1-29
- “Convert R’G’B’ to Intensity Images” on page 1-41
- “Process Multidimensional Color Video Signals” on page 1-46
- “Data Formats” on page 1-51

Export to Video Files

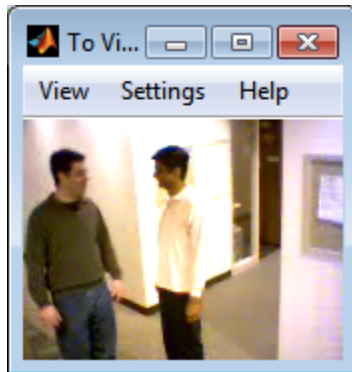
The Computer Vision System Toolbox™ blocks enable you to export video data from your Simulink® model. In this example, you use the To Multimedia File block to export a multimedia file from your model. This example also uses Gain blocks from the **Math Operations** Simulink library.

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	<p>The Gain blocks are used to increase the red, green, and blue values of the video stream. This increases the contrast of the video:</p> <ul style="list-style-type: none"> • Main pane, Gain = 1.2 • Signal Attributes pane, Output data type = Inherit: Same as input
To Multimedia File	<p>The To Multimedia File block exports the video to a multimedia file:</p> <ul style="list-style-type: none"> • Output file name = my_output.avi • Write = Video only • Image signal = Separate color signals

Configuration Parameters

You can locate the **Model Configuration Parameters** by selecting **Model 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)

Import from Video Files

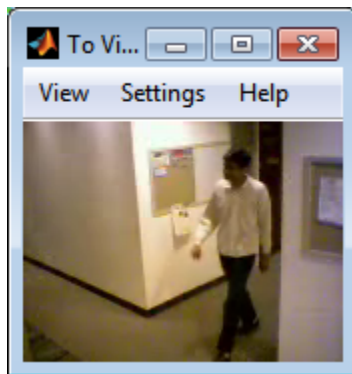
In this example, you use the From Multimedia File source block to import a video stream into a Simulink model and the To Video Display sink block to view it. This procedure assumes you are working on a Windows platform.

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 1-2 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.

Setting Block Parameters for this Example

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

Block	Parameter
From Multimedia File	<p>Use the From Multimedia File block to import the multimedia file into the model:</p> <ul style="list-style-type: none"> • If you do not have your own multimedia file, use the default vipmen.avi 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 Separate color signals. <p>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.</p>
To Video Display	<p>Use the To Video Display block to view the multimedia file.</p> <ul style="list-style-type: none"> • Image signal: Separate color signals <p>Set this parameter from the Settings menu of the display viewer.</p>

Configuration Parameters

You can locate the **Model Configuration Parameters** by selecting **Model 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
```

For additional information about batch processing, see the Batch Processing Image Files Using Distributed Computing example in Image Processing Toolbox. You can run the example by typing `ipexbatch` at the MATLAB command prompt.

Configuration Parameters

You can locate the **Model Configuration Parameters** by selecting **Model 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

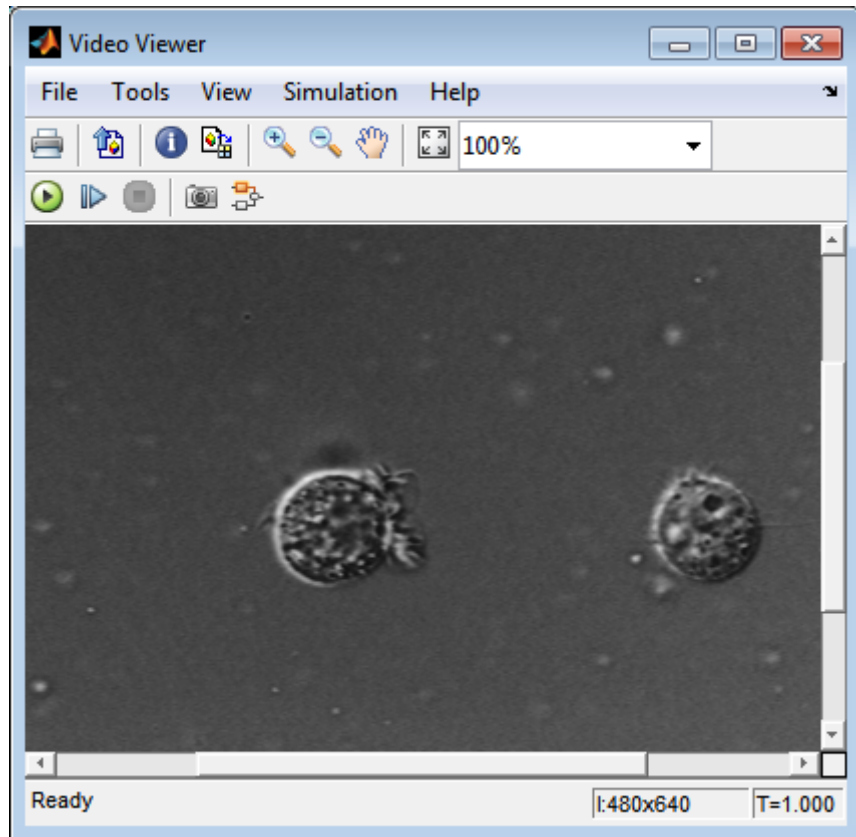
This example displays a sequence of images, which were saved in a folder, and then stored in a variable in the MATLAB workspace. At load time, this model executes the code from the “Batch Process Image Files” on page 1-7 example, which stores images in a workspace variable.

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 **File > Model Properties > Model Properties**. Then select the **Callbacks** tab. For more details on how to set-up callbacks, see "Callback Functions".

Configuration Parameters

You can locate the **Model Configuration Parameters** by selecting **Model 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	<p>The From Multimedia File block imports an AVI file into the model.</p> <ul style="list-style-type: none"> • Cleared Inherit sample time from file checkbox.
Insert Text	<p>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.</p> <ul style="list-style-type: none"> • Text: 'Frame %d' • Color: [0 1 0] • Location: [10 10]
To Multimedia File	<p>The To Multimedia File blocks send the video stream to three separate AVI files. These block parameters were modified as follows:</p> <ul style="list-style-type: none"> • Output file name: output1.avi, output2.avi, and output3.avi, respectively • Write: Video only
Counter	<p>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:</p> <ul style="list-style-type: none"> • Number of bits: 8 • Sample time: 1/30

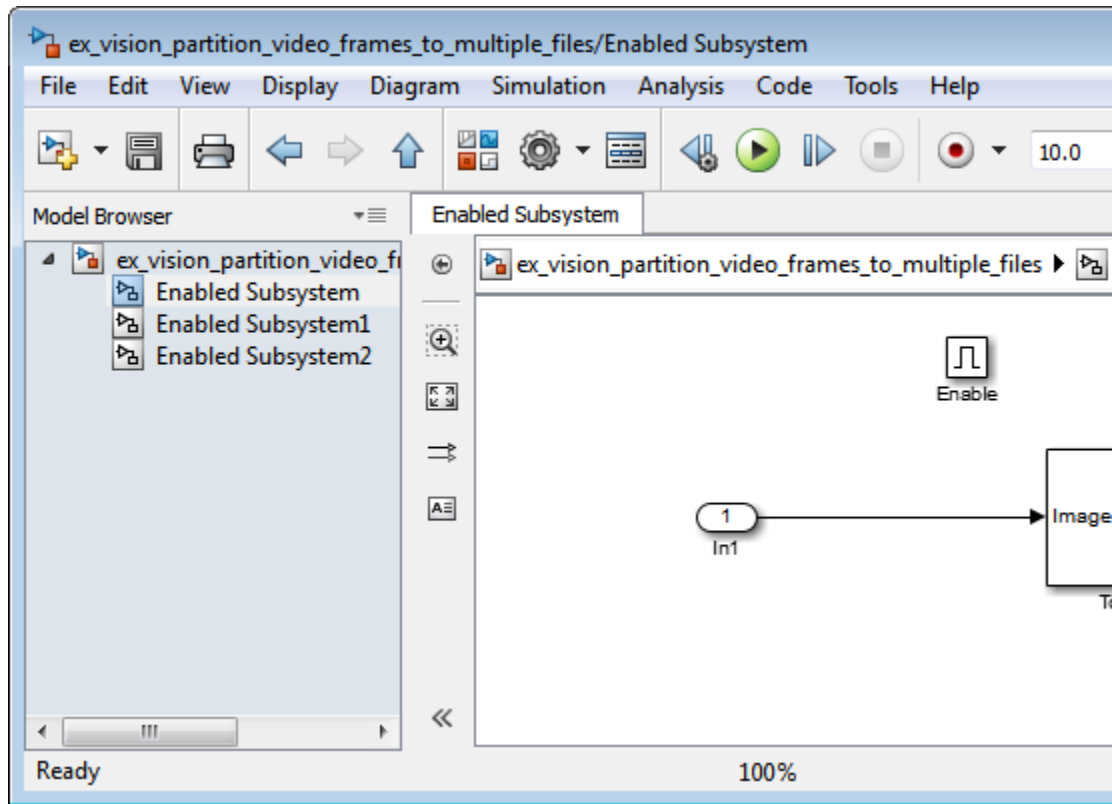
Block	Parameter
Bias	<p>The bias block adds a bias to the input. The block parameters are modified as follows:</p> <ul style="list-style-type: none"> • Bias: 1
Compare to Constant	<p>The Compare to Constant block sends frames 1 to 9 to the first AVI file. The block parameters are modified as follows:</p> <ul style="list-style-type: none"> • Operator: < • Constant value: 10
Compare to Constant1 Compare to Constant2	<p>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:</p> <ul style="list-style-type: none"> • Operator: >= • Constant value: 10 <p>The Compare to Constant2 block parameters are modified as follows:</p> <ul style="list-style-type: none"> • Operator: < • Constant value: 20

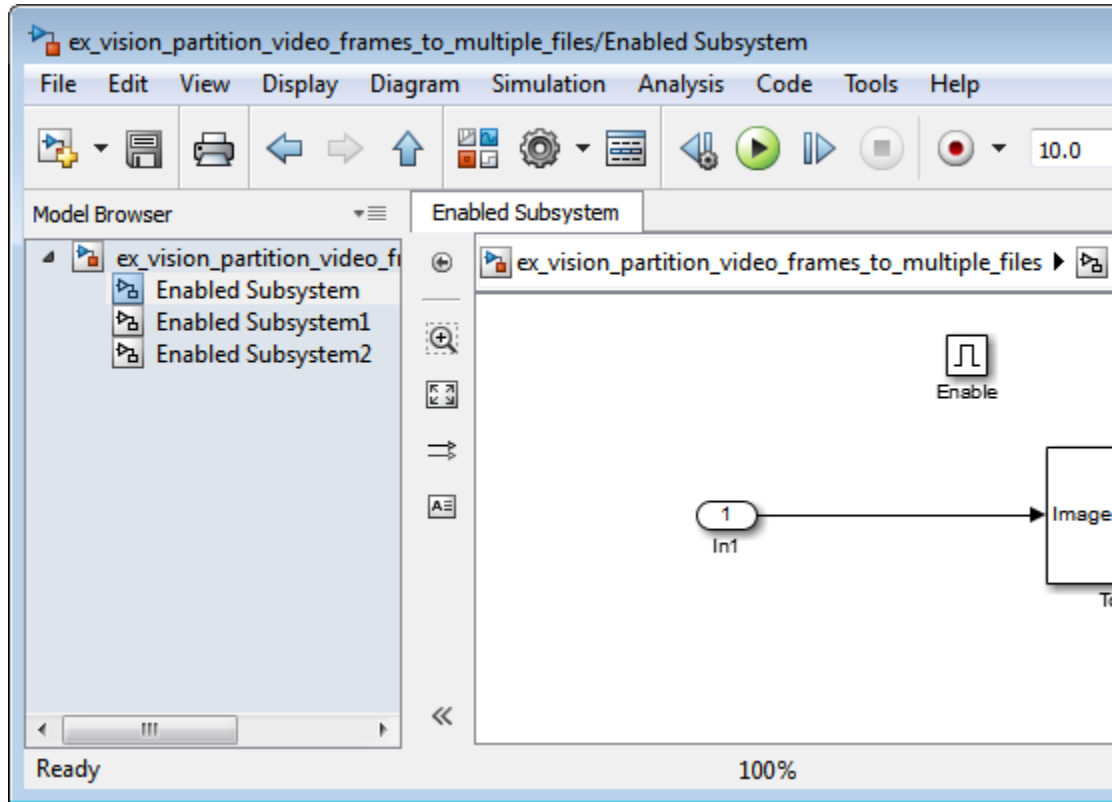
Block	Parameter
Compare to Constant3	<p>The Compare to Constant3 block send frames 20 to 30 to the third AVI file. The block parameters are modified as follows:</p> <ul style="list-style-type: none"> • Operator: >= • Constant value: 20
Compare to Constant4	<p>The Compare to Constant4 block stopa the simulation when the video reaches frame 30. The block parameters are modified as follows:</p> <ul style="list-style-type: none"> • 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 **Model Configuration Parameters** by selecting **Model 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

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 $1/(\text{frame rate})$. 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.

$$\text{frame period} = (\text{frame size})/(\text{frequency})$$

$$\text{frame period} = (735 \text{ samples per frame})/(22050 \text{ samples per second})$$

$$\text{frame period} = 0.0333 \text{ seconds per frame}$$

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.

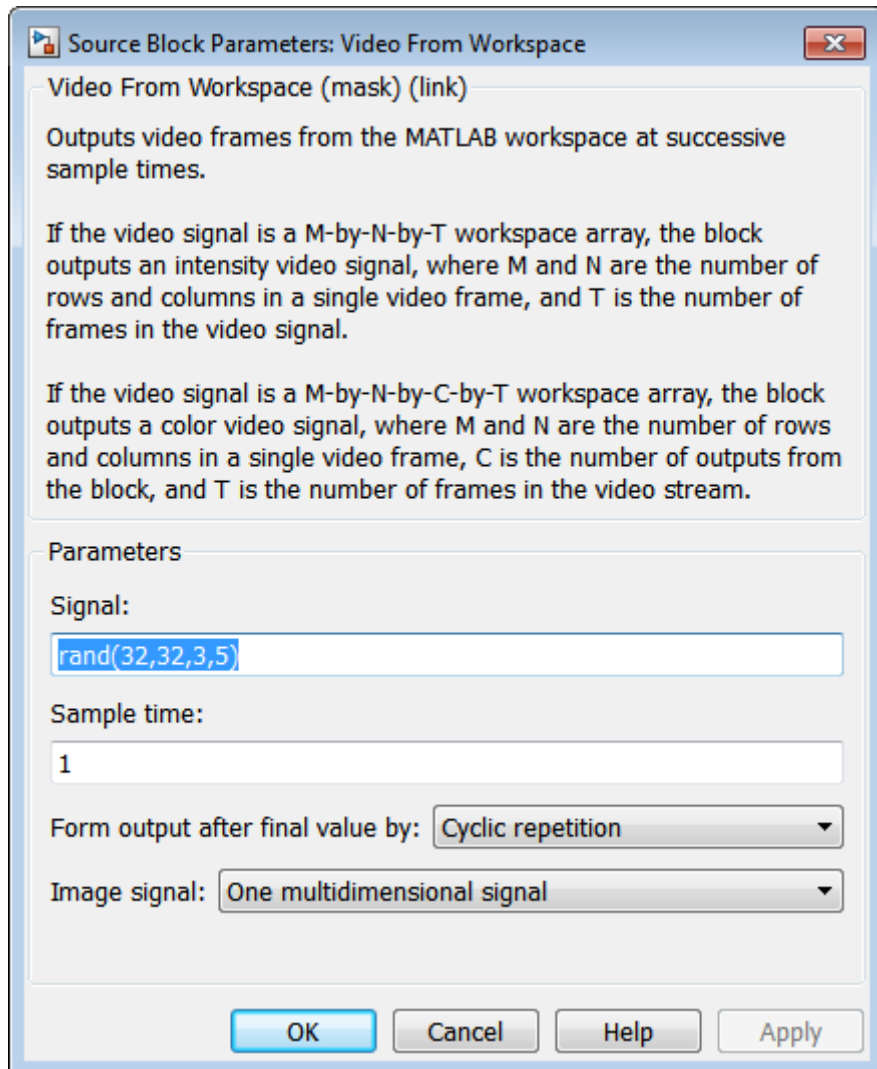
Configuration Parameters

You can locate the Configuration Parameters by selecting **Model Configuration Parameters** from the **Simulation** menu. 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](#).

Transmit Audio and Video Content Over 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™.

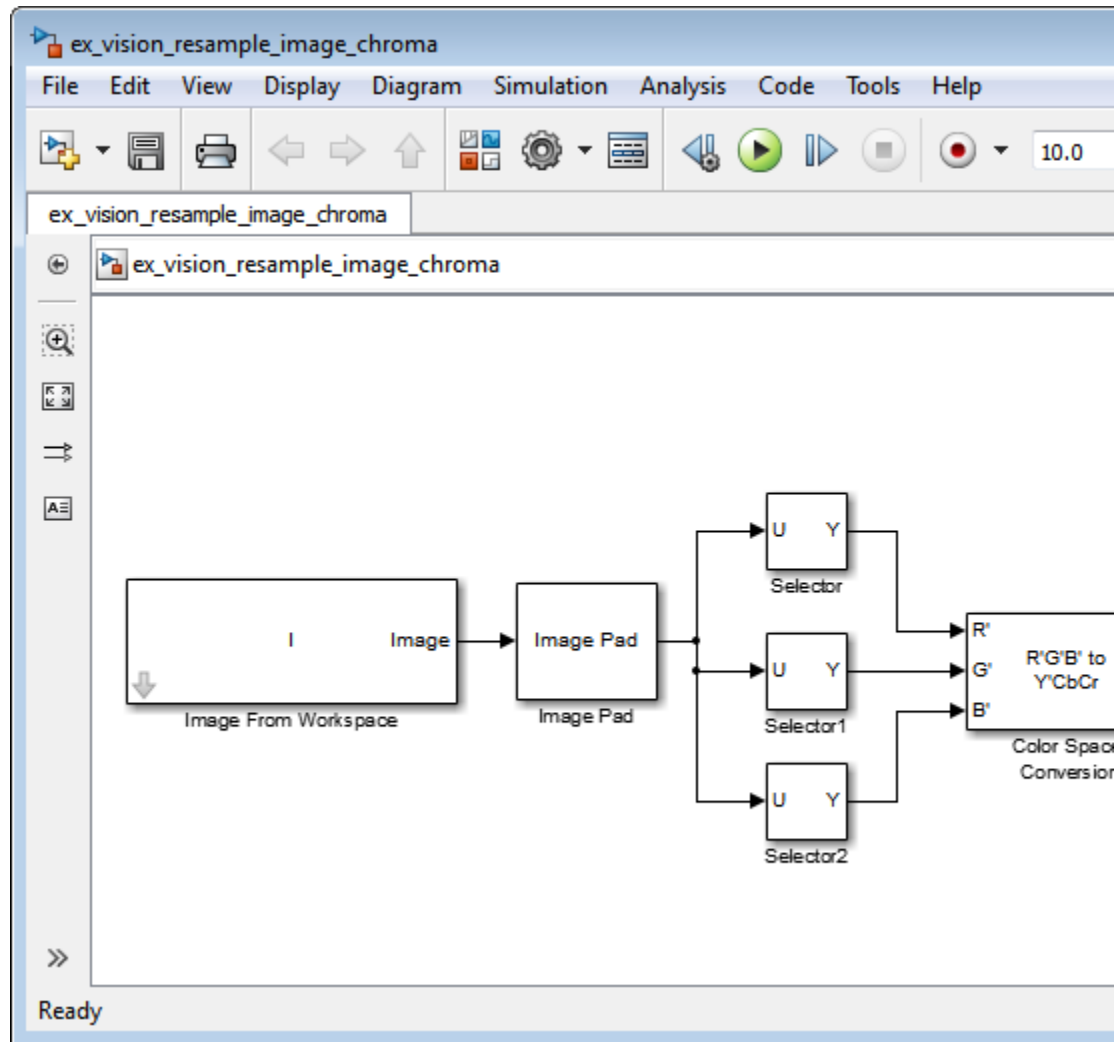
Resample Image Chroma

In this example, you use the Chroma Resampling block to downsample the Cb and Cr components of an image. The YCbCr 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:

Block	Parameter
Image from Workspace	Import your image from the MATLAB workspace. Set the Value parameter to <code>I</code> .
Image Pad	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: <ul style="list-style-type: none"> • Method = Symmetric • Add columns to = Right • Number of added columns = 1

Block	Parameter
	<ul style="list-style-type: none"> • Add row to = No padding <p>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.</p> <hr/> <p>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.</p> <hr/>
Selector, Selector1, Selector2	<p>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:</p> <p>Selector1</p> <ul style="list-style-type: none"> • Number of input dimensions = 3 • Index 1 = Select all • Index 2 = Select all • Index 3 = Index vector (dialog) and Index = 1 <p>Selector2</p> <ul style="list-style-type: none"> • Number of input dimensions = 3 • Index 1 = Select all • Index 2 = Select all • Index 3 = Index vector (dialog) and Index = 2 <p>Selector2</p> <ul style="list-style-type: none"> • Number of input dimensions = 3

Block	Parameter
	<ul style="list-style-type: none"> • 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.
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: <ul style="list-style-type: none"> • 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 **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)

Convert Intensity to Binary Images

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” on page 1-29 “Thresholding Intensity Images Using the Autothreshold Block” on page 1-34

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.

You can open the example model by typing

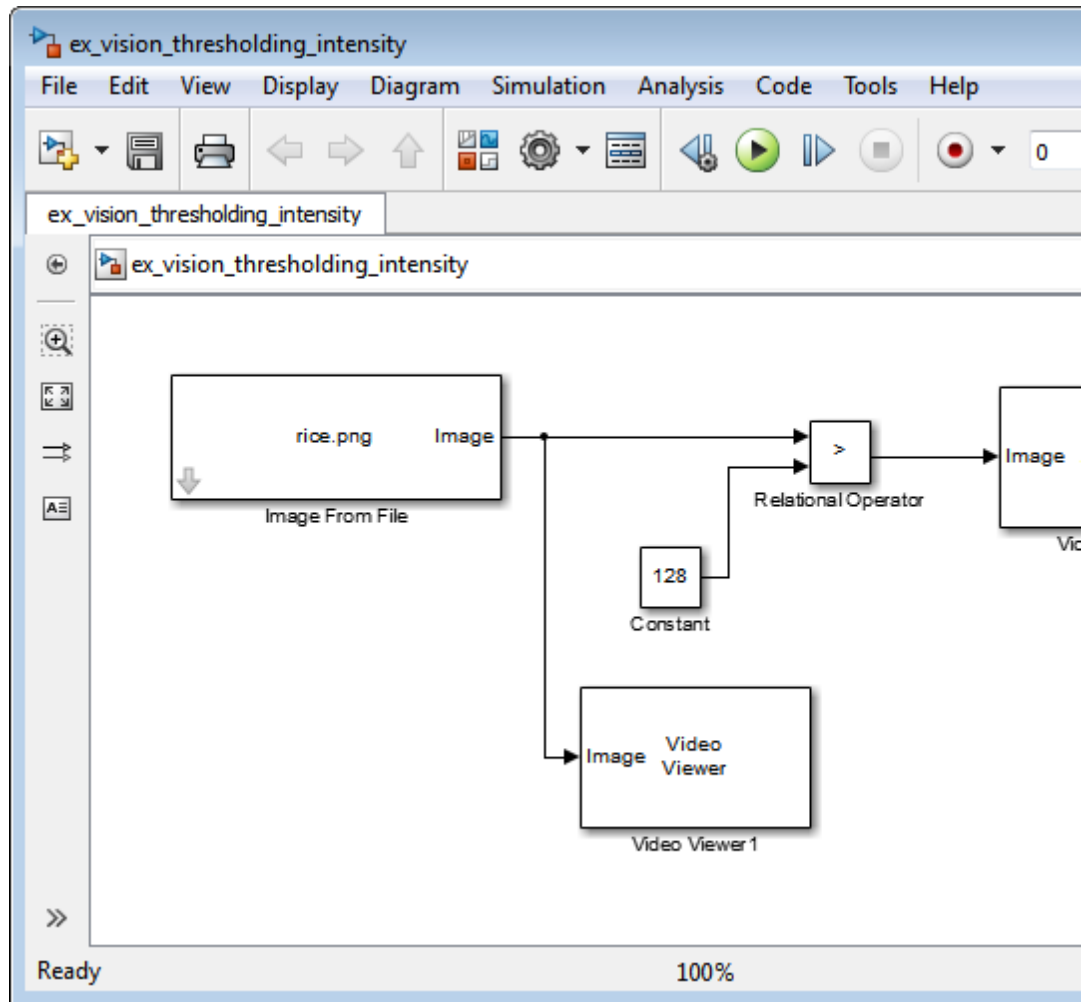
```
ex_vision_thresholding_intensity
```

on the MATLAB command line.

- 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 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 Use the Video Viewer1 block to view the original intensity image. Accept the default parameters.
- 4 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.
- 5 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.
- 6 Use the Video Viewer block to view the binary image. Accept the default parameters.
- 7 Connect the blocks as shown in the following figure.

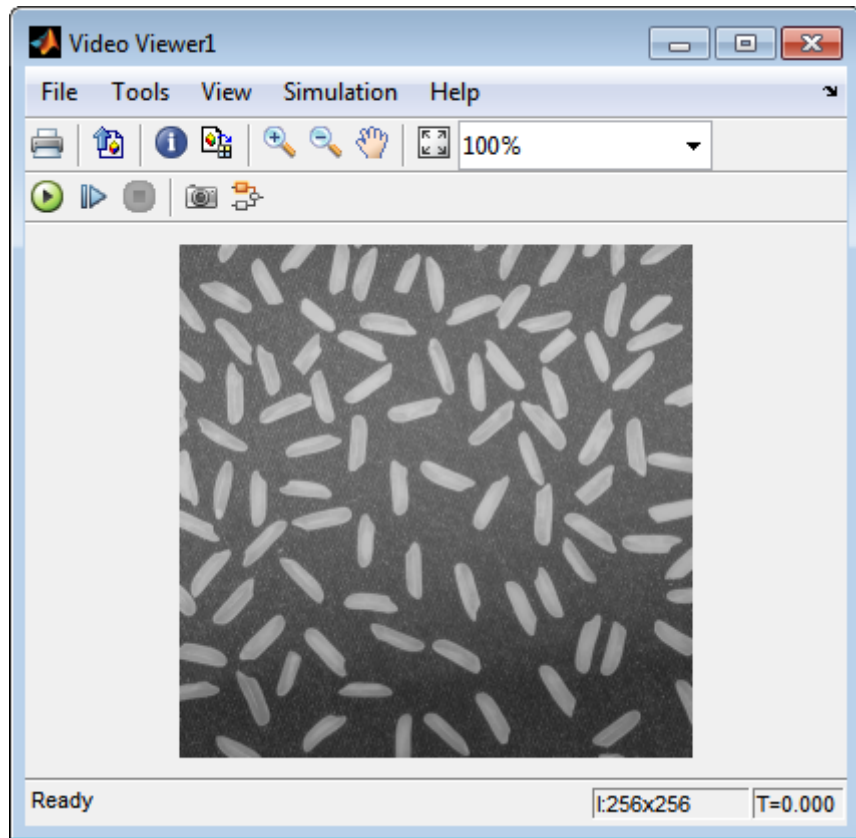


8 Set the configuration parameters. Open the Configuration dialog box by selecting **Simulation > Model Configuration Parameters** menu. Set the parameters as follows:

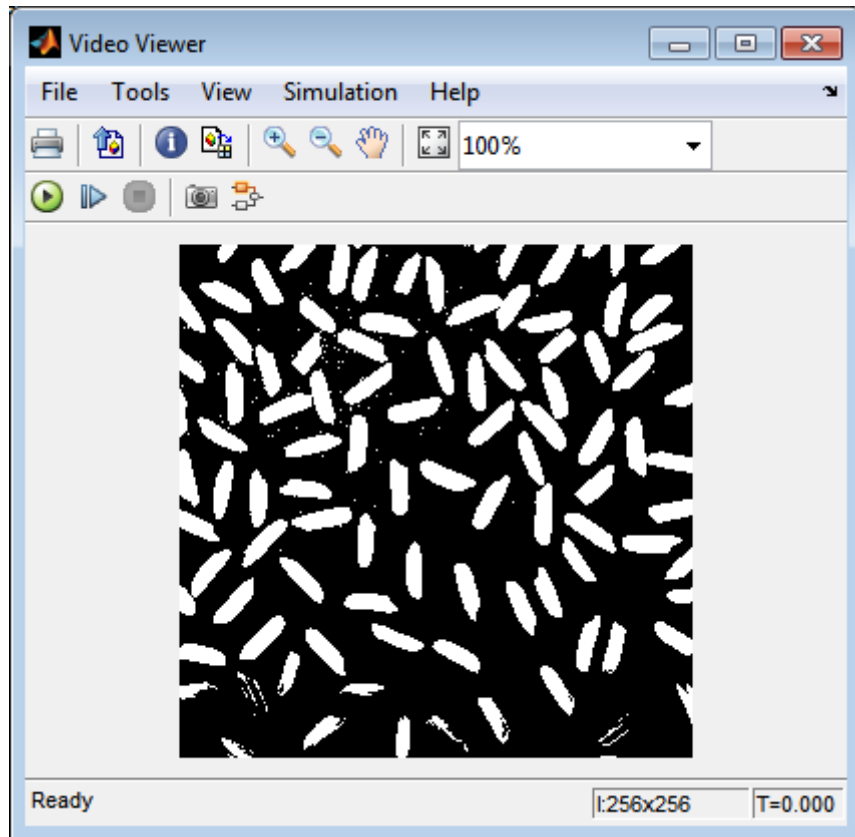
- **Solver** pane, **Stop time** = 0
- **Solver** pane, **Type** = Fixed-step
- **Solver** pane, **Solver** = Discrete (no continuous states)

9 Run 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 7-8.

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.

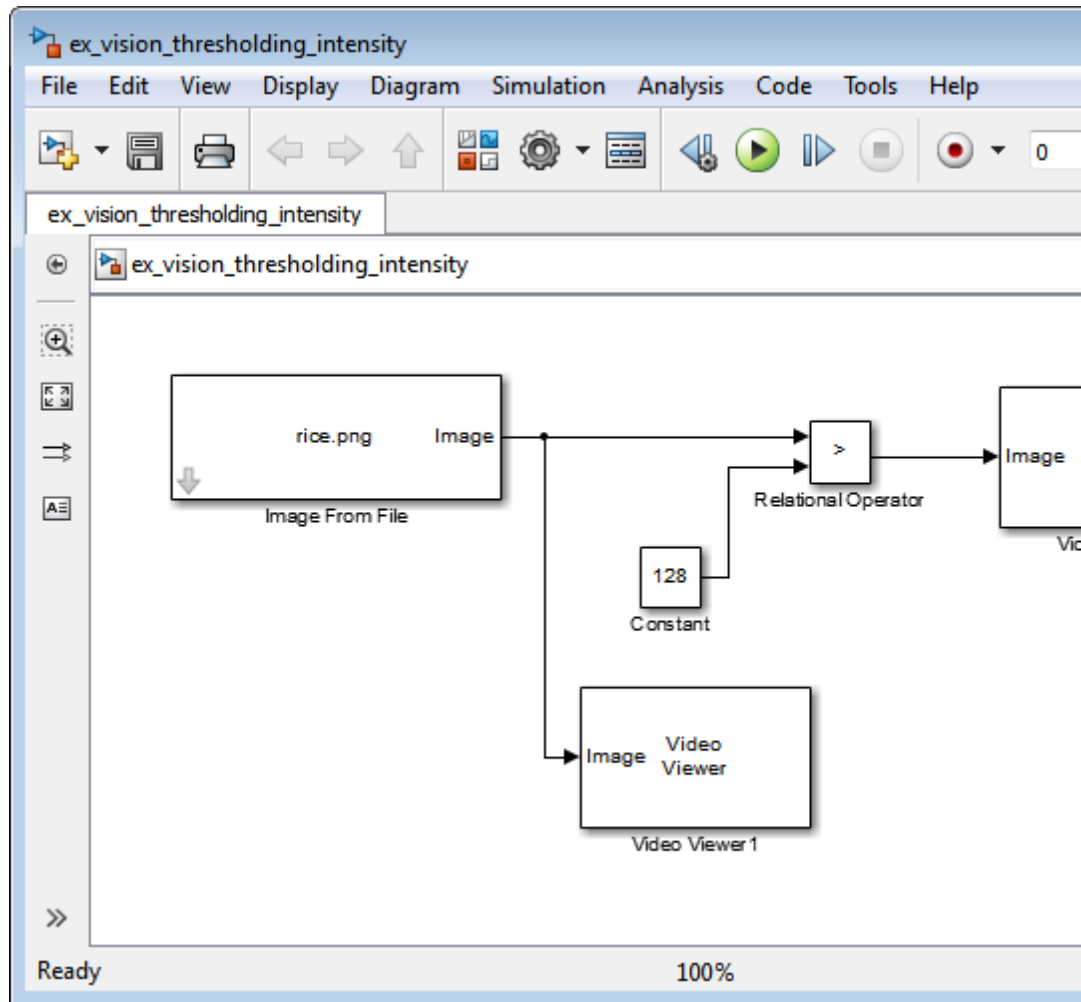
Note Running this example requires a DSP System Toolbox™ license.

`ex_vision_autothreshold`

- 1** If the model you created in “Thresholding Intensity Images Using Relational Operators” on page 1-29 is not open on your desktop, you can open the model by typing

`ex_vision_thresholding_intensity`

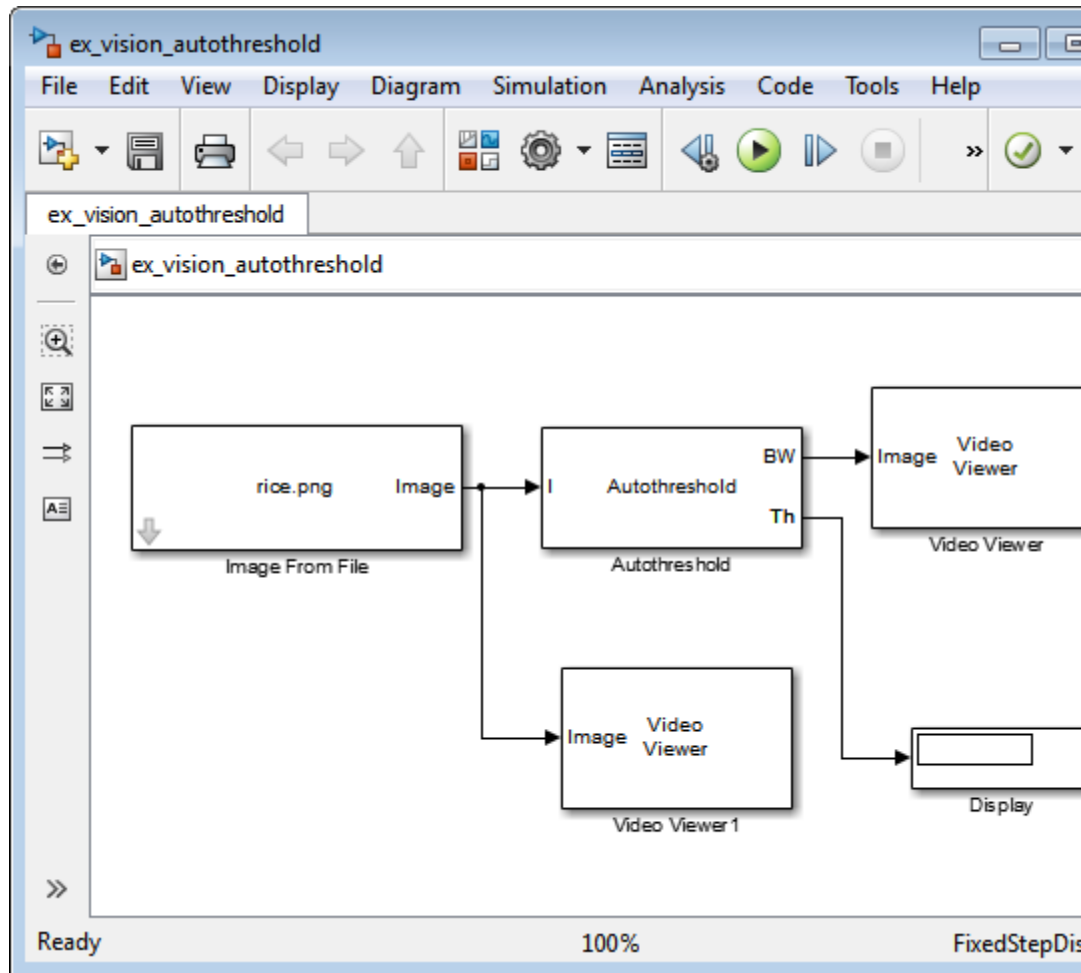
at the MATLAB command prompt.



- 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 Use the Autothreshold block to perform a thresholding operation that converts your intensity image to a binary image. Select the **Output threshold** check box. This block outputs the calculated threshold value at the **Th** port.
- 6 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 Autothreshold block.

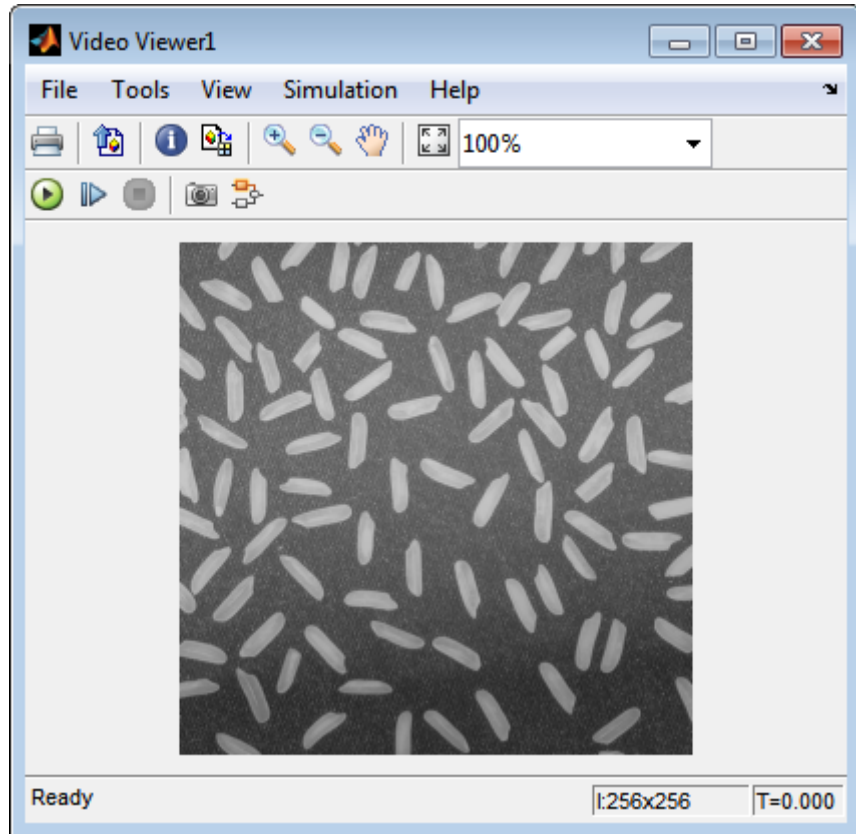
Your model should look similar to the following figure:



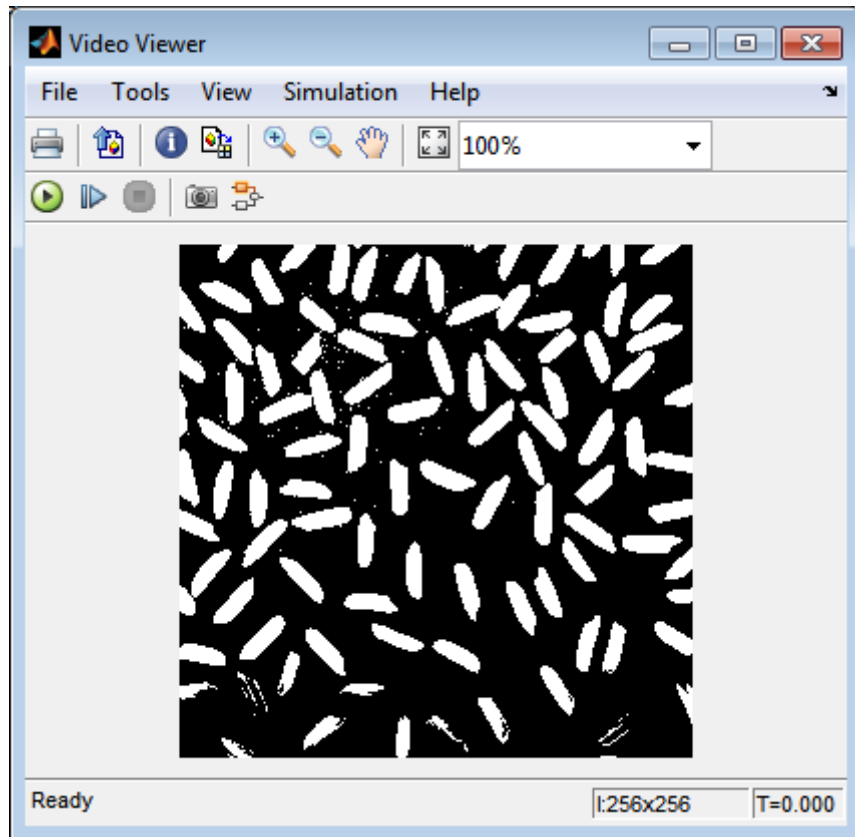
- 7 Double-click the Image From File block. On the **Data Types** pane, set the **Output data type** parameter to double.
- 8 If you have not already done so, set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:
 - **Solver** pane, **Stop time** = 0
 - **Solver** pane, **Type** = Fixed-step

- Solver pane, Solver = Discrete (no continuous states)
- 9 Run the model.

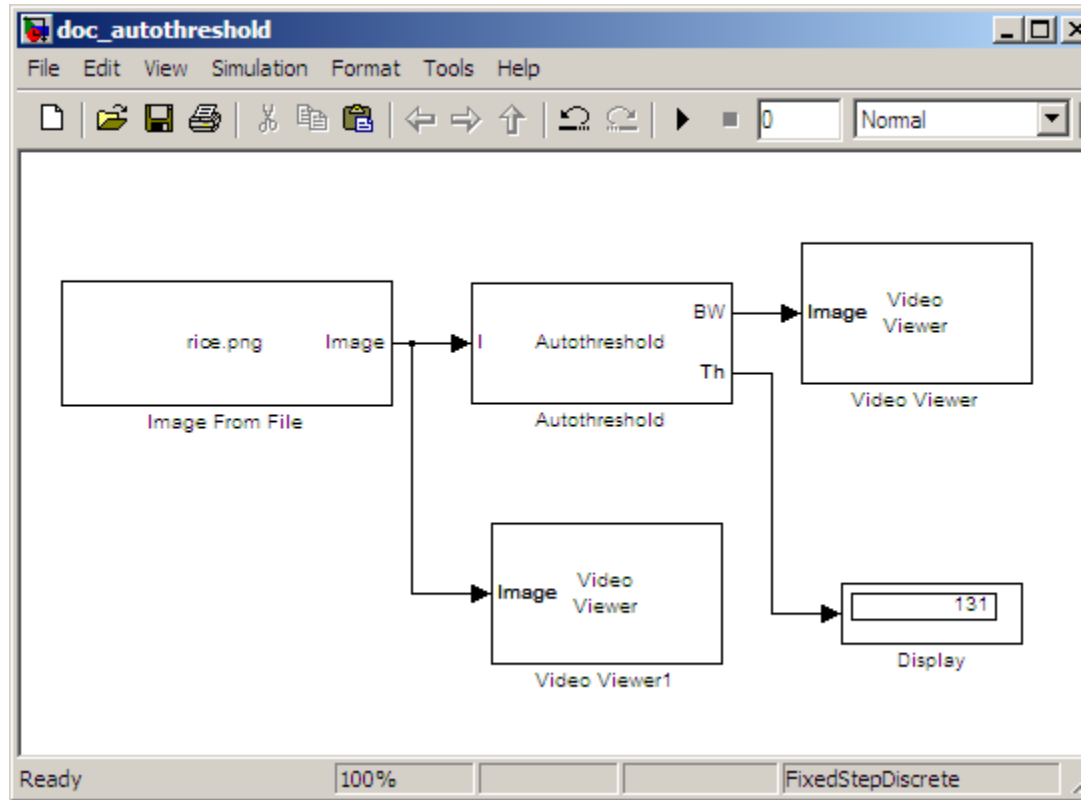
The original intensity image appears in the Video Viewer1 window.



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 an example 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:

```
ex_vision_convert_rgb
```

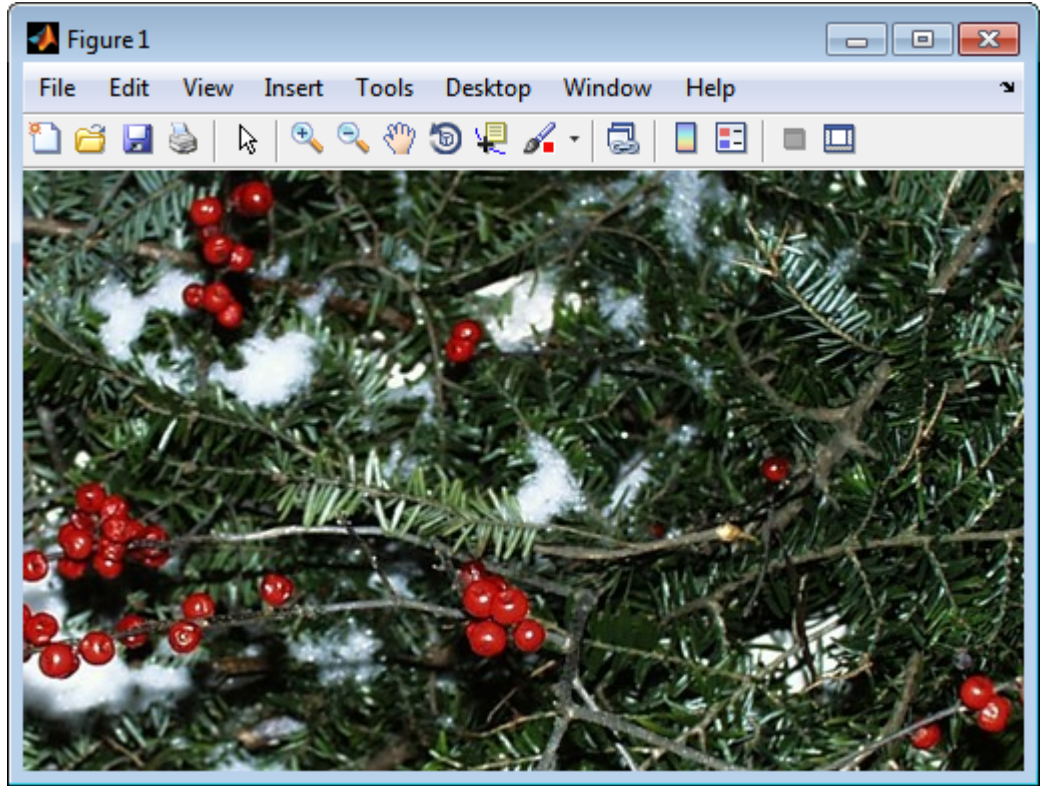
- 1** Define an R'G'B' image in the MATLAB workspace. To read in an R'G'B' image from a JPG file, at the MATLAB command prompt, type

```
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)
```

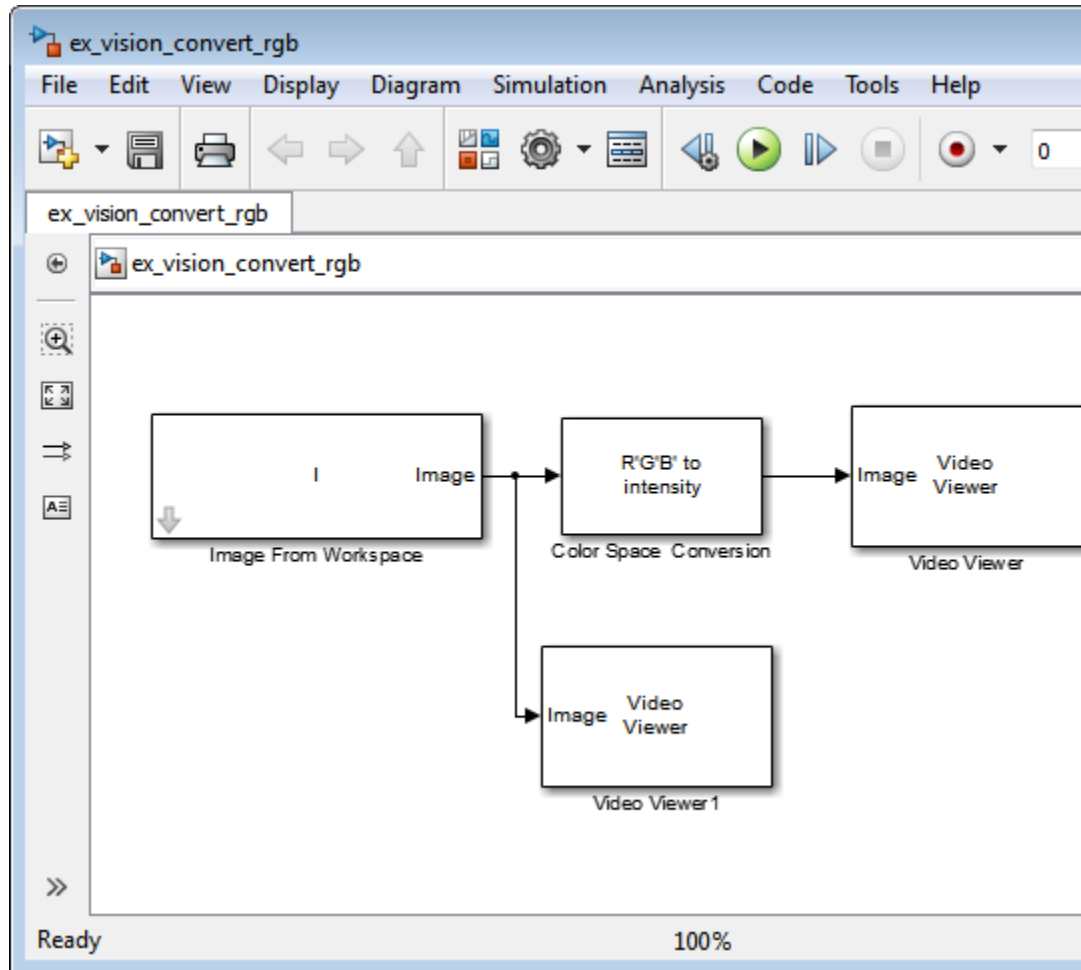


- 3 Create a new Simulink model, and add to it the blocks shown in the following table.

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	2

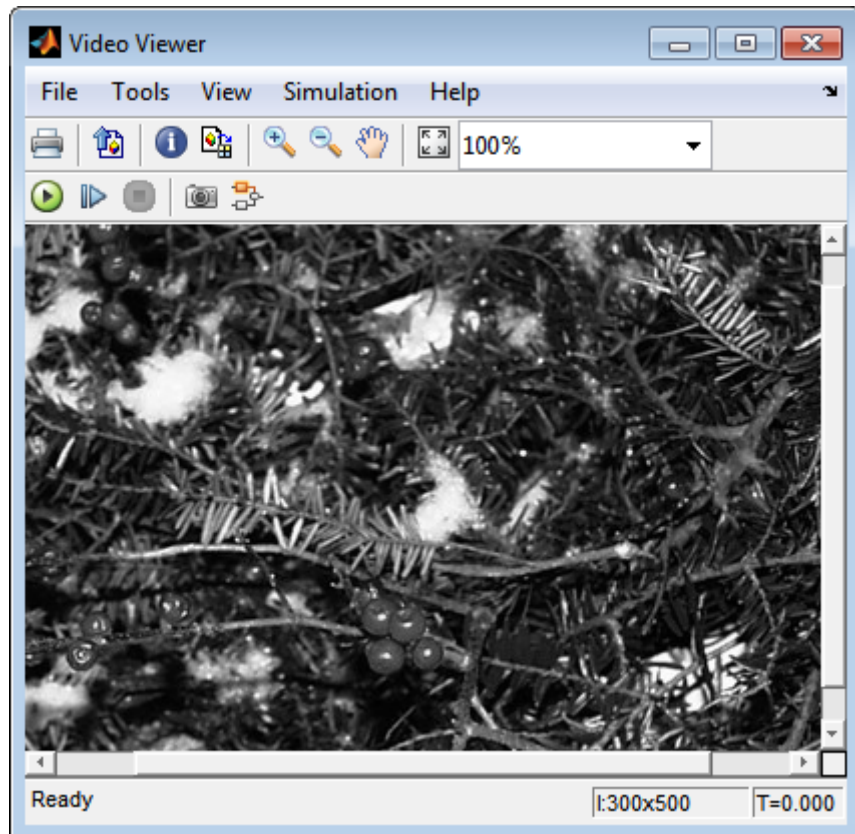
- 4 Use the Image from Workspace block to import your image from the MATLAB workspace. Set the **Value** parameter to I.

- 5 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.
- 6 View the modified image using the Video Viewer block. View the original image using the Video Viewer1 block. Accept the default parameters.
- 7 Connect the blocks so that your model is similar to the following figure.



- 8 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)
- 9 Run your model.

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

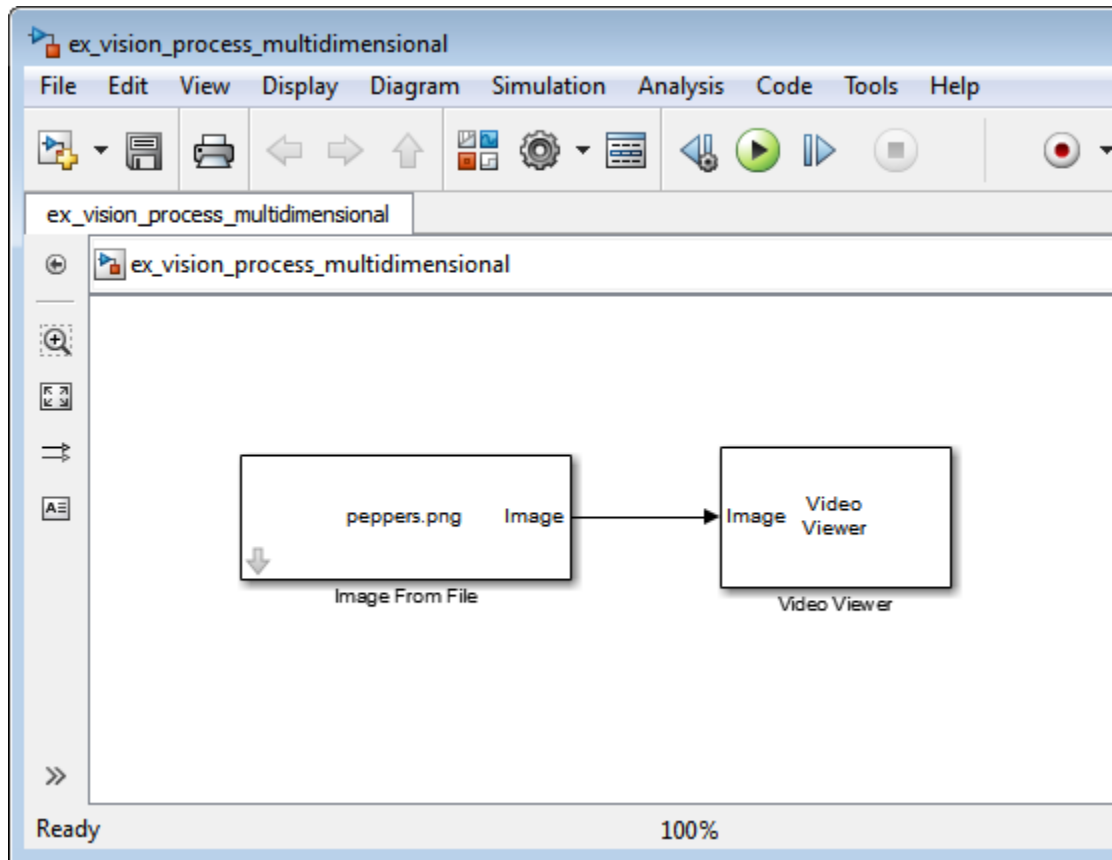


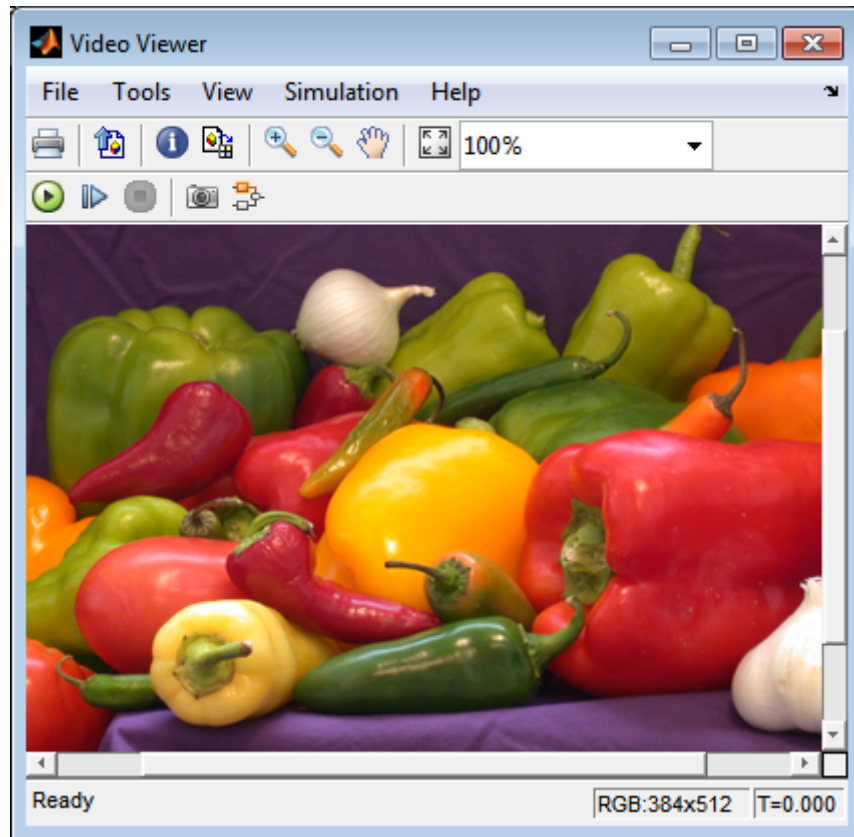
In this topic, you used the Color Space Conversion block to convert color information from the R'G'B' color space to intensity. For more information on this block, see the Color Space Conversion block reference page.

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.

```
ex_vision_process_multidimensional
```





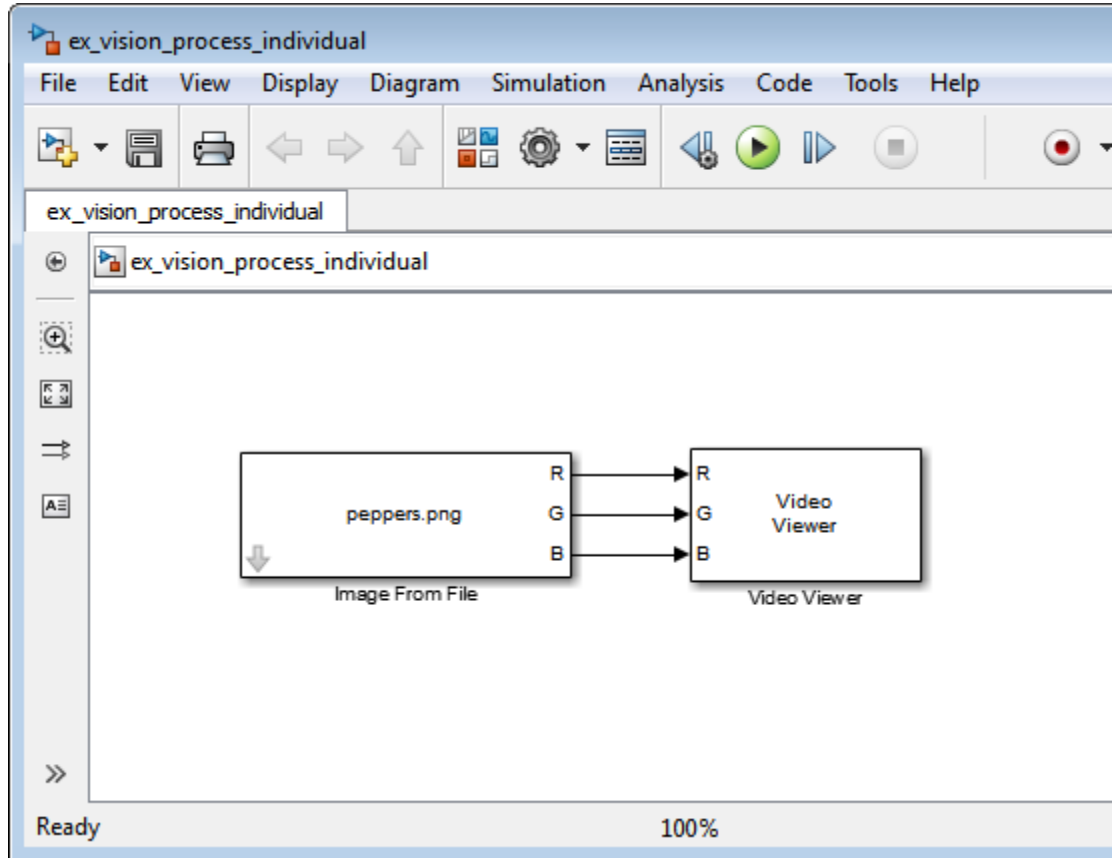
You can choose to process the image as a multidimensional array by setting the **Image signal** parameter to One multidimensional signal in the Image From File block dialog box.

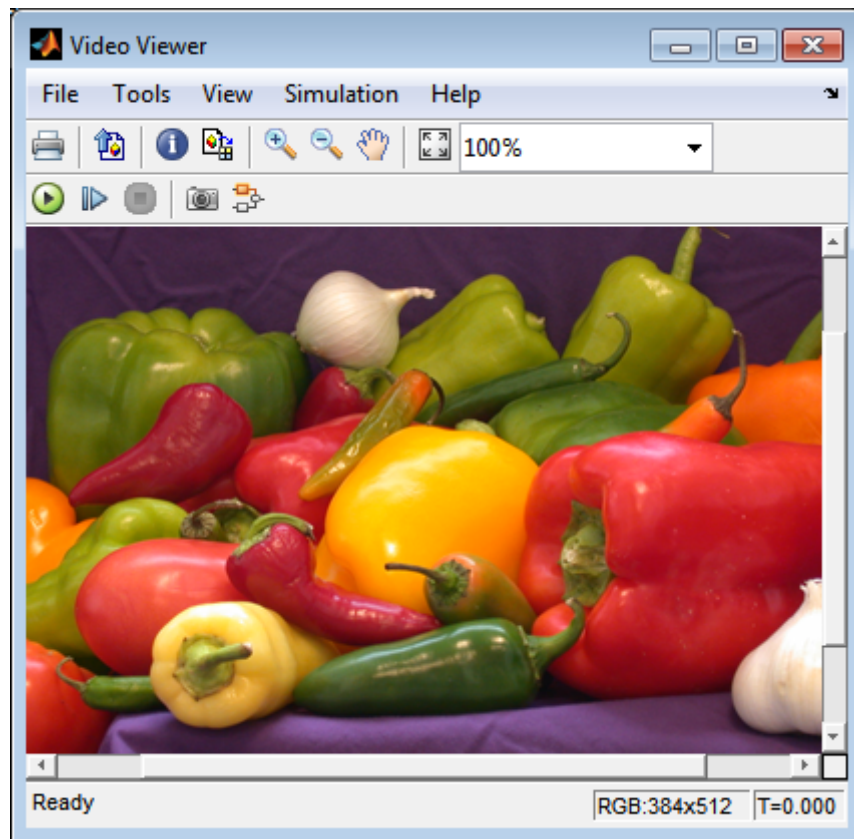
The blocks that support multidimensional arrays meet at least one of the following criteria:

- 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".

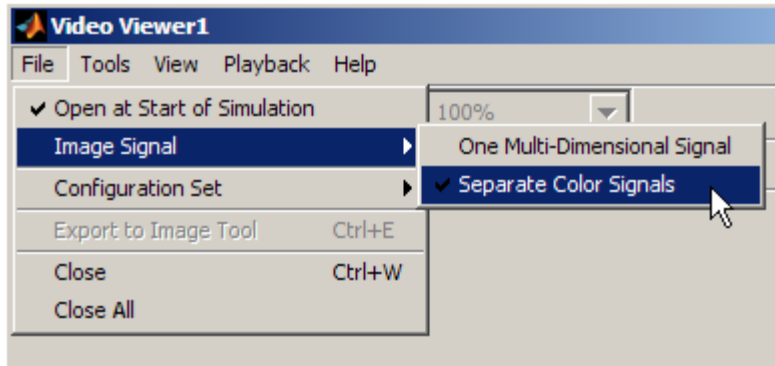
You can also choose to work with the individual color planes of images or video signals. For example, the following model passes a color image from a source block to a sink block using three separate color planes.

`ex_vision_process_individual`





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. 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.

Data Formats

In this section...

“Video Formats” on page 1-51

“Video Data Stored in Column-Major Format” on page 1-52

“Image Formats” on page 1-52

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 2-25

Display

In this section...
“View Streaming Video in MATLAB” on page 2-2
“Preview Video in MATLAB using MPlay Function” on page 2-2
“To Video Display Block” on page 2-3
“View Video with MPlay” on page 2-4
“MPlay” on page 2-7

View Streaming Video in MATLAB

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

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 with Simulink 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 with MPlay

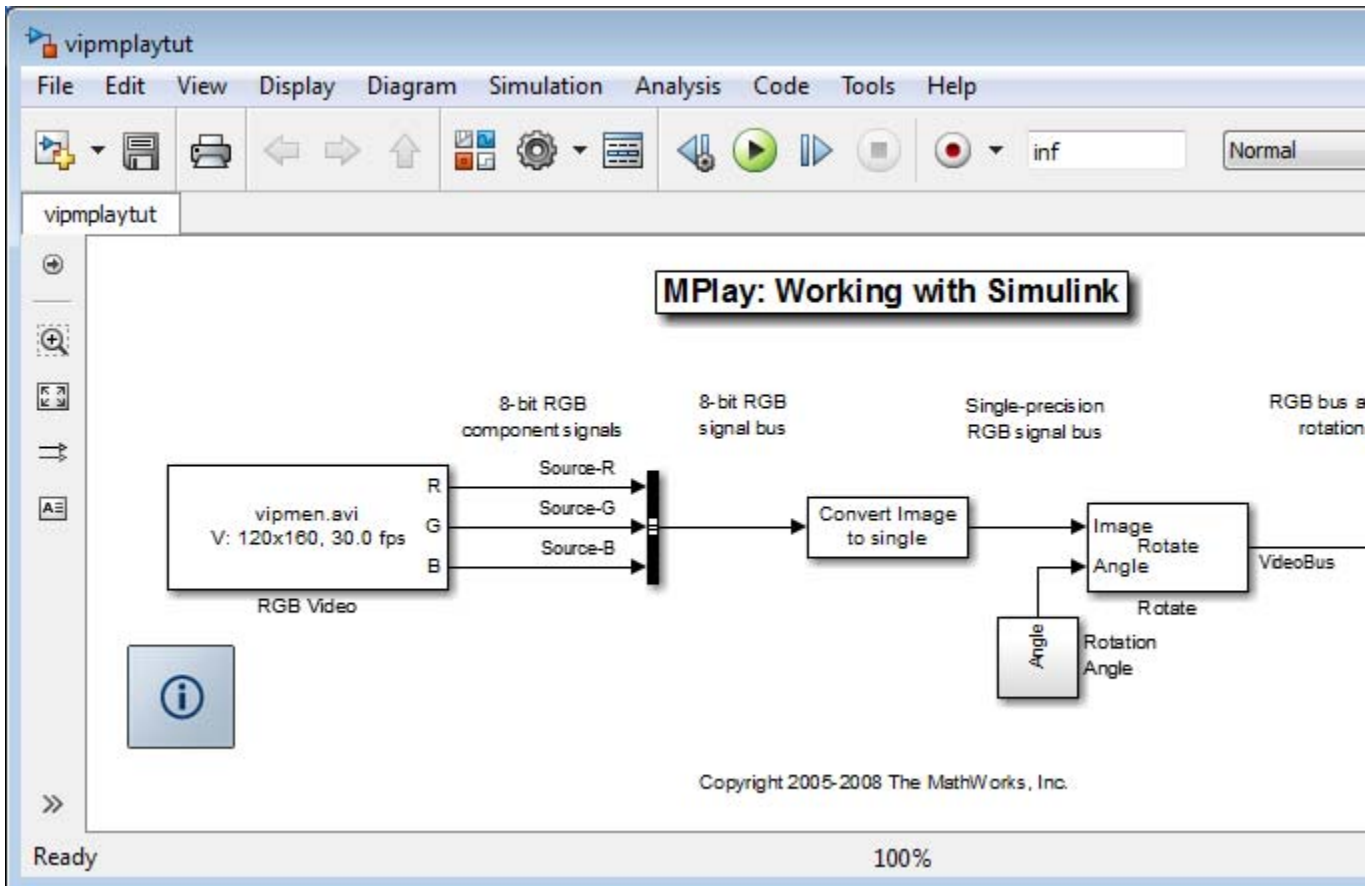
The MPlay GUI enables you to view video signals in Simulink models without adding blocks to your model.


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 9-10.

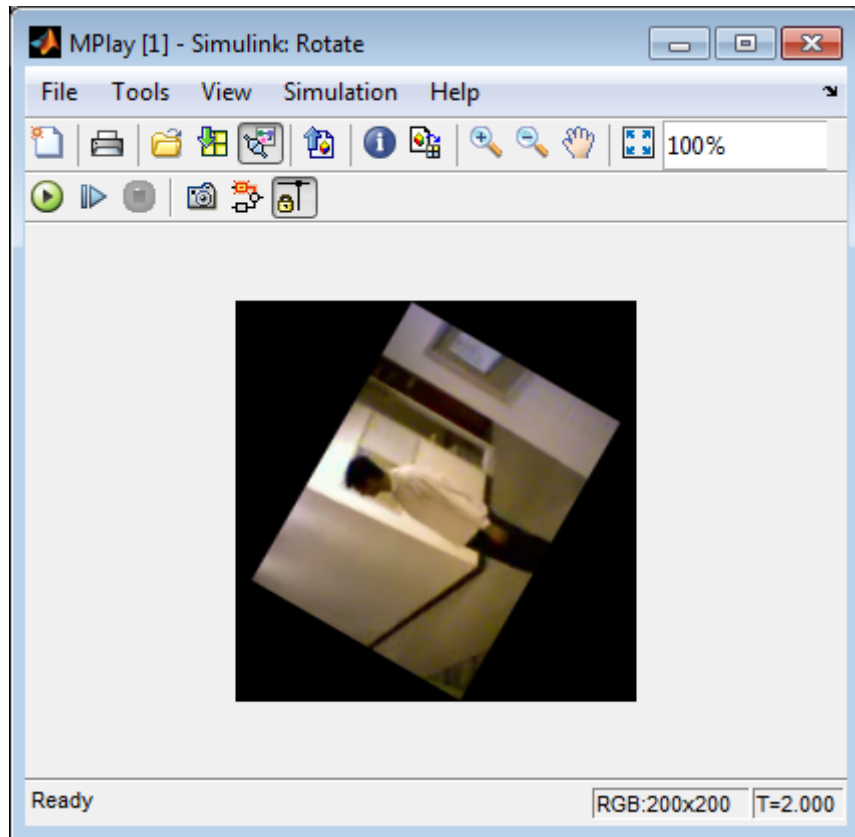
The following procedure shows you how to use MPlay to view a Simulink signal:


- 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.

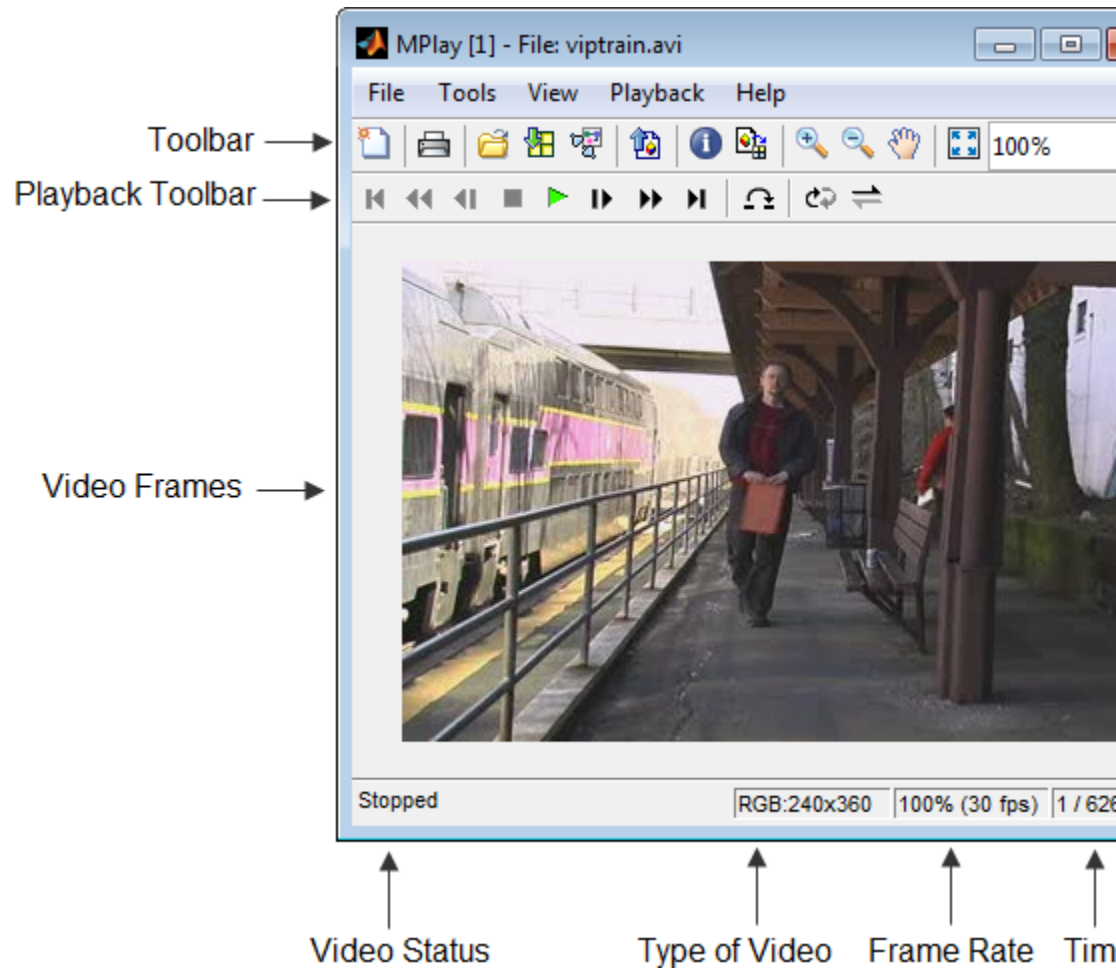


- 6 Change to floating-scope mode by clicking the *persistent connect* GUI element,  button.
- 7 Experiment with selecting different signals and viewing them in the MPlay window. You can also use multiple MPlay GUIs to display different Simulink signals.

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	<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.</p> <p>To print the current window to a figure rather than sending it to your printer, select File > Print to figure.</p>
	File > Print	Ctrl+P	<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.</p> <p>To print the current scope window to a figure rather than sending it to your printer, select File > Print to figure.</p>
	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.
	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.

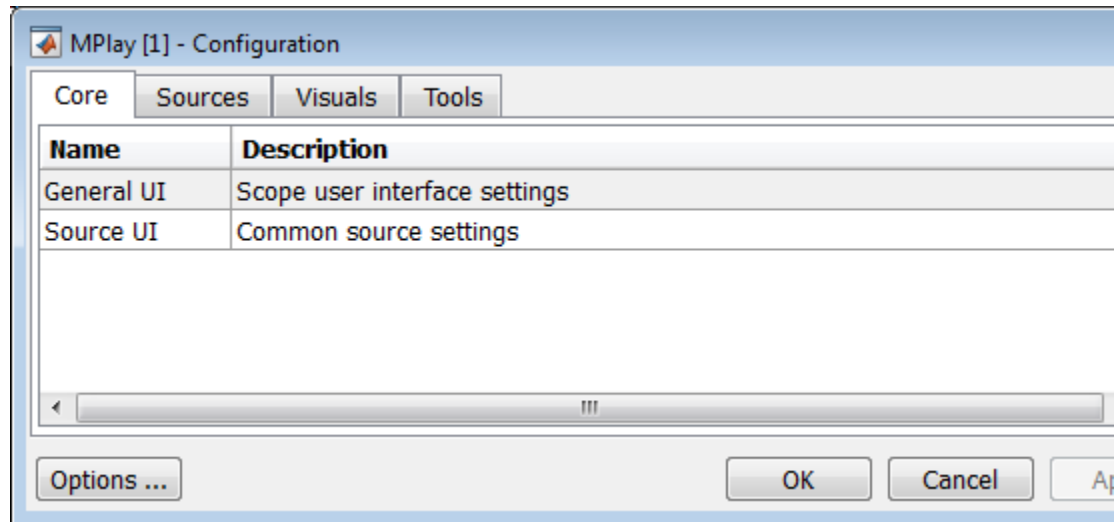
- To open the Configuration dialog box, select **File > Configuration Set > Edit**.
- To save the configuration settings for future use, select **File > Configuration Set > Save as**.

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**.

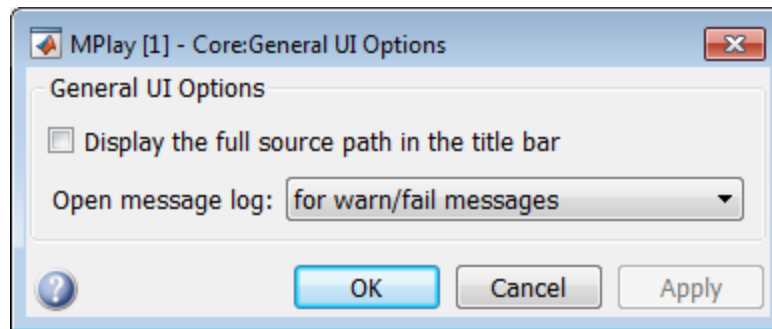
Configuration Core Pane

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.

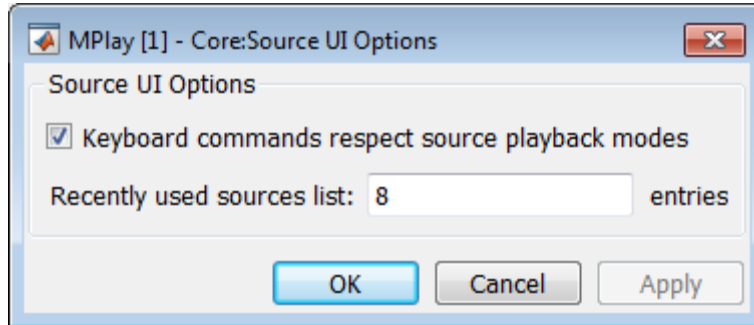


If you select the **Display the full source path in the title bar** check box, the full Simulink path appears in the title bar. Otherwise, the title bar displays a shortened name.

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.

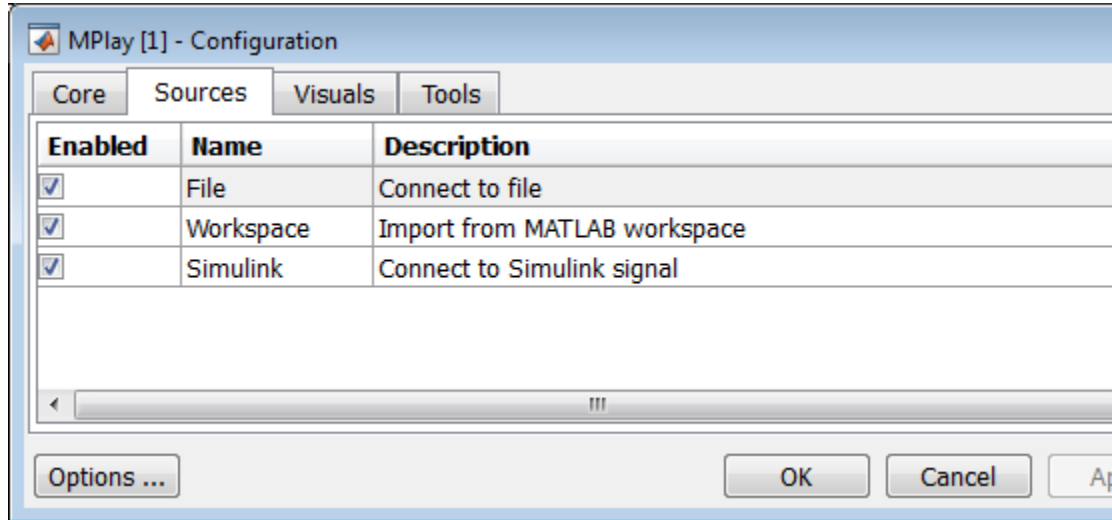
- 1 Select the **Keyboard commands respect playback modes** check box.
- 2 Select the **Backward playback** button.
 - Using the right keyboard arrow key moves the video backward, and using the left keyboard arrow key moves the video forward.
 - With MPlay set to play backwards, the keyboard “forward” performs “forward with the direction the video is playing”.

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

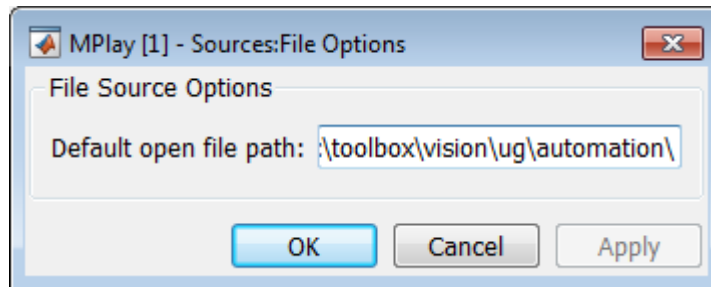
Use the **Recently used sources list** parameter to control the number of sources you see in the **File** menu.

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.

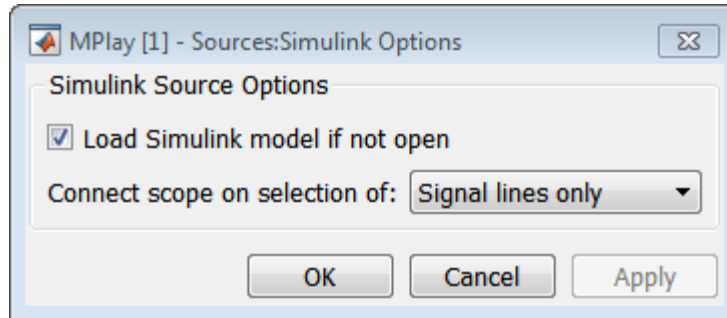


- Click **File**, and then click the **Options** button to open the **Sources:File Options** dialog box.



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.

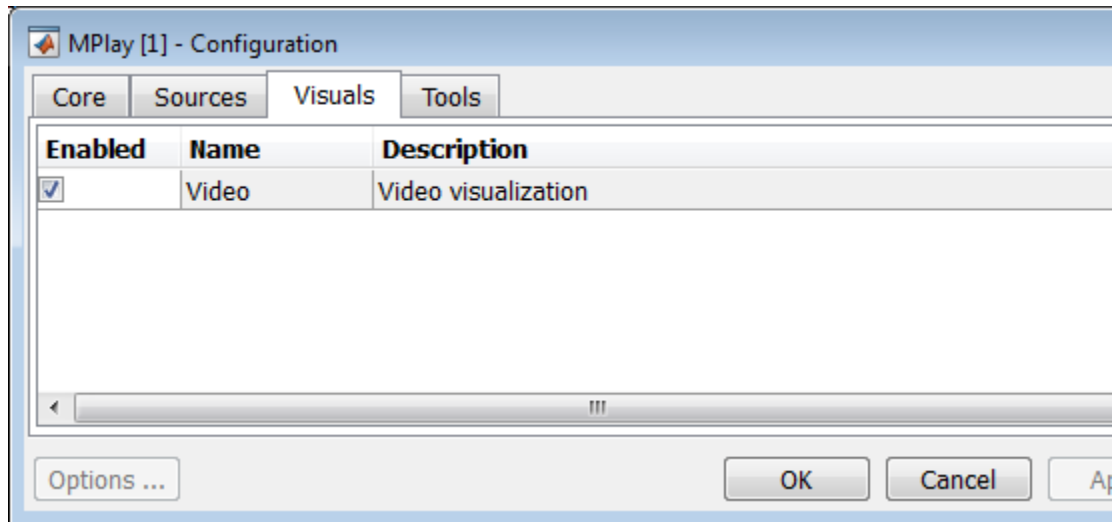


You can have the Simulink model associated with an MPlay GUI to open with MPlay. To do so, select the **Load Simulink model if not open** check box.

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**.

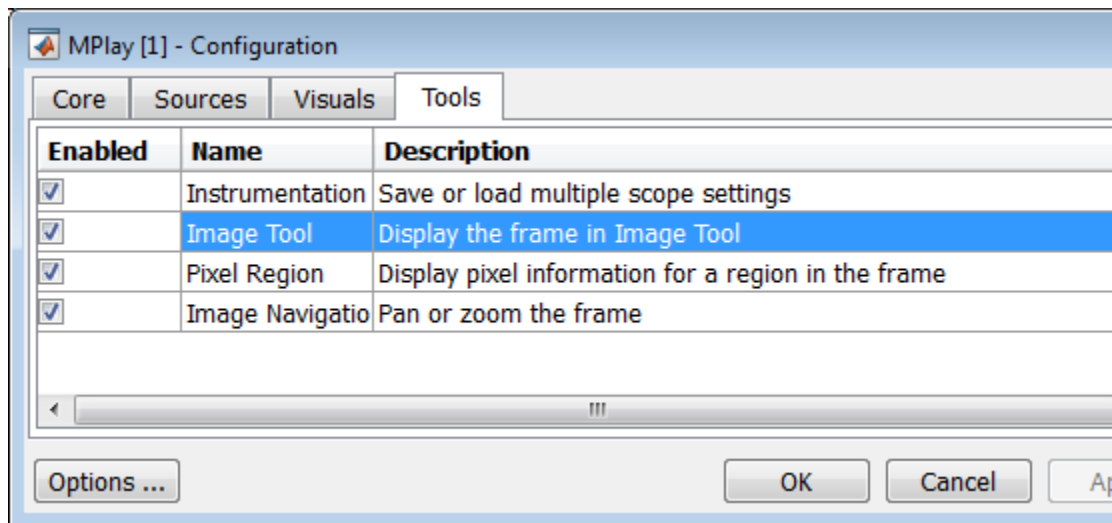
Configuration Visuals Pane

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

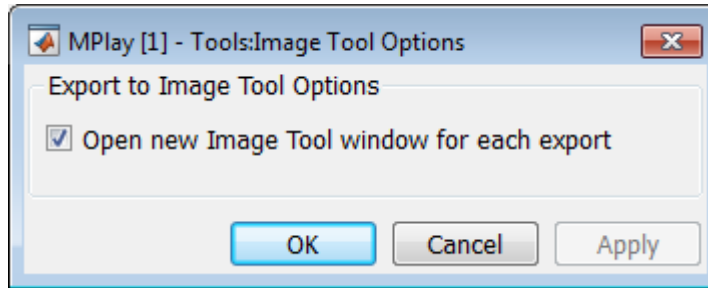


Configuration Tools Pane

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

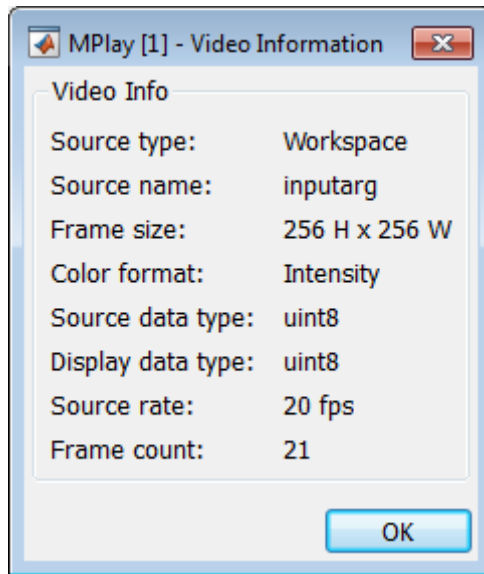
Select the **Image Navigation Tools** check box to enable the pan-and-zoom GUI button.

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.

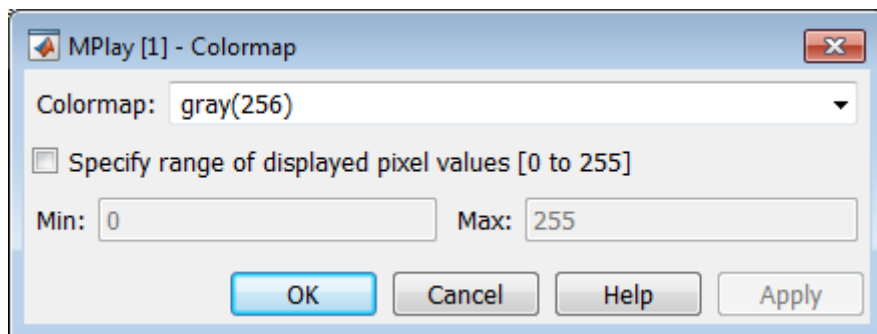
Video Information

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.

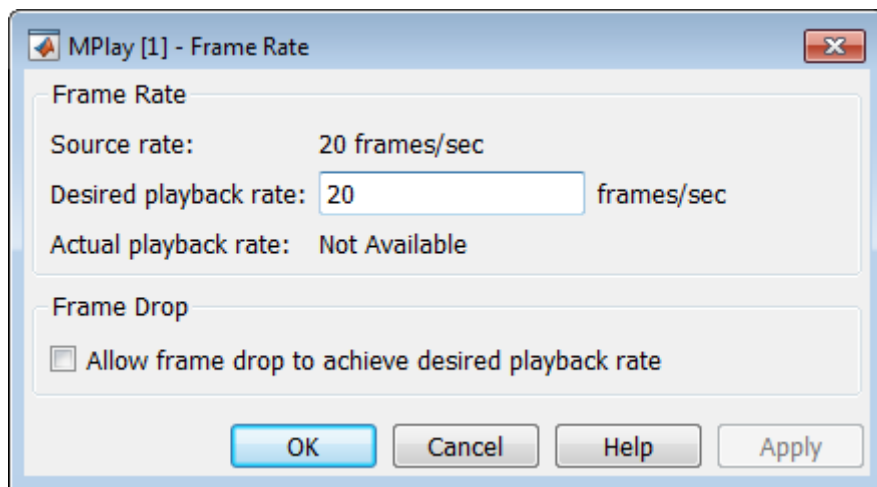
Sometimes, the pixel values do not use the entire data type range. In such cases, you can select the **Specify range of displayed pixel values** check box. You can then enter the range for your data. The dialog box automatically displays the range based on the data type of the pixel values.

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.

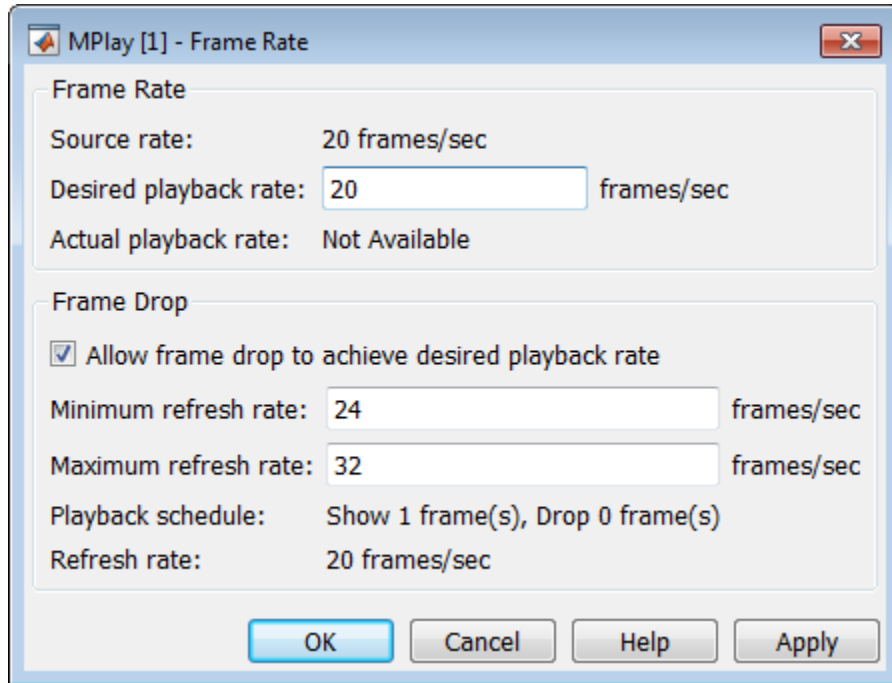
Note This dialog box becomes available when you use the MPlay GUI to view a video signal.

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 2-25
“Abandoned Object Detection” on page 2-29
“Annotate Video Files with Frame Numbers” on page 2-35
“Draw Shapes and Lines” on page 2-37

Abandoned Object Detection

This example shows how to track objects at a train station and to determine 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.

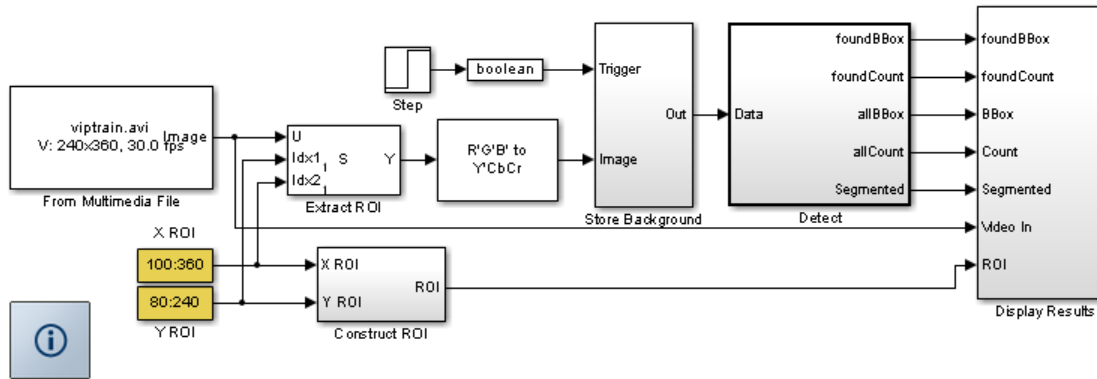
This example illustrates how to use the Blob Analysis and MATLAB Function blocks to design a custom tracking algorithm. The example implements this algorithm using the following steps: 1) Eliminate video areas that are unlikely to contain abandoned objects by extracting a region of interest (ROI). 2) Perform video segmentation using background subtraction. 3) Calculate object statistics using the Blob Analysis block. 4) Track objects based on their area and centroid statistics. 5) Visualize the results.

Watch the Abandoned Object Detection example.

Example Model

The following figure shows the Abandoned Object Detection example model.

Abandoned Object Detection

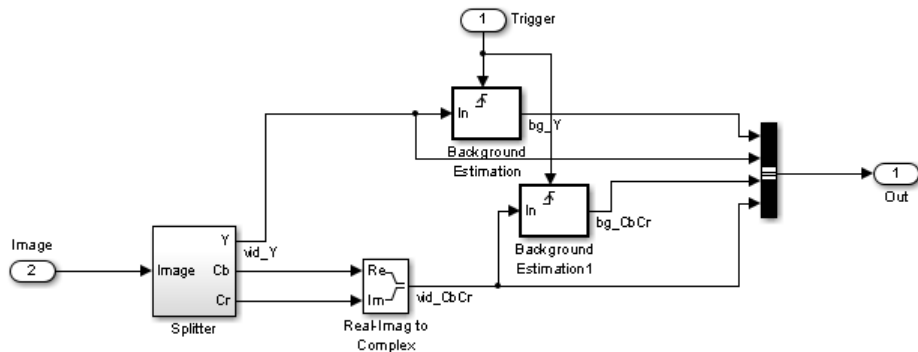


Copyright 2006-2010 The MathWorks, Inc.

Store Background Subsystem

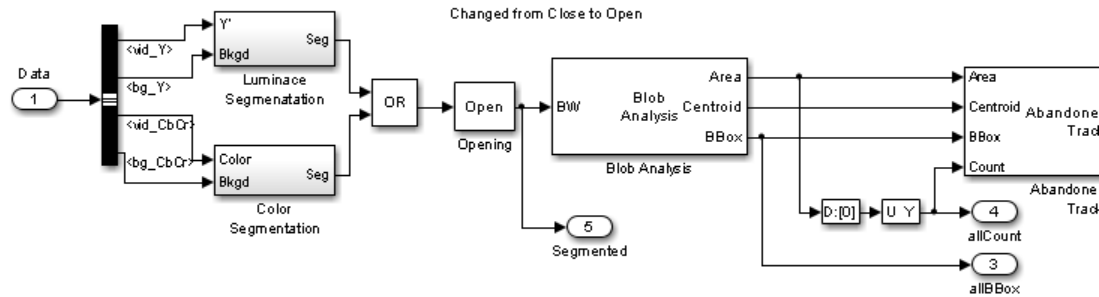
This example uses the first frame of the video as the background. To improve accuracy, the example uses both intensity and color information for the background subtraction operation. During this operation, Cb and Cr color channels are stored in a complex array.

If you are designing a professional surveillance system, you should implement a more sophisticated segmentation algorithm.

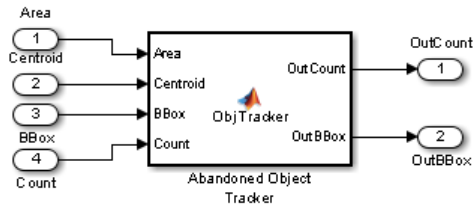


Detect Subsystem

The Detect subsystem contains the main algorithm. Inside this subsystem, the Luminance Segmentation and Color Segmentation subsystems perform background subtraction using the intensity and color data. The example combines these two segmentation results using a binary OR operator. The Blob Analysis block computes statistics of the objects present in the scene.



Abandoned Object Tracker subsystem, shown below, uses the object statistics to determine which objects are stationary. To view the contents of this subsystem, right-click the subsystem and select Look Under Mask. To view the tracking algorithm details, double-click the Abandoned Object Tracker block. The MATLAB code in this block is an example of how to implement your custom code to augment Computer Vision System Toolbox™ functionality.

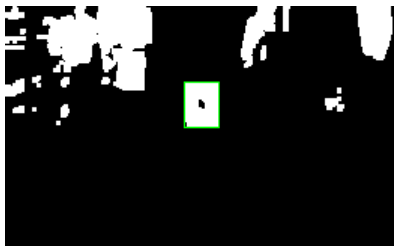


Abandoned Object Detection Results

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.



The Abandoned Objects window highlights the abandoned objects with a red box.



Abandoned Object Detection

This example shows how to track objects at a train station and it 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 upper 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 YCbCr format.

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

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

```
hAutothreshold = vision.Autothresher('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', 'Position', pos);
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);

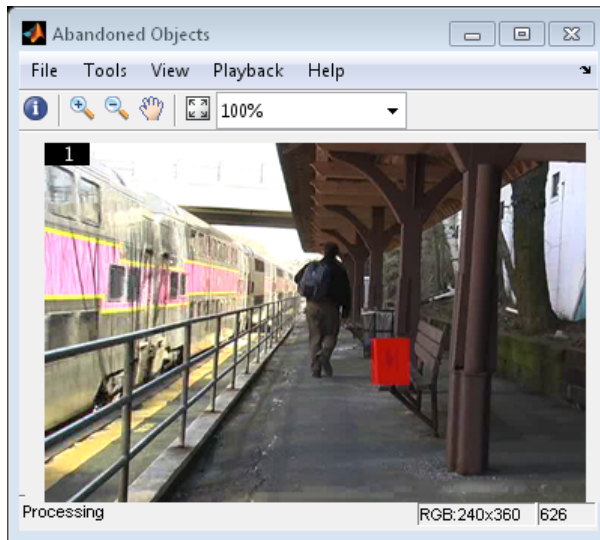
BlobCount = size(BBox,1);

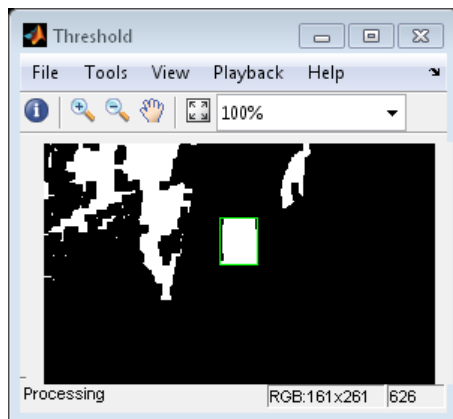
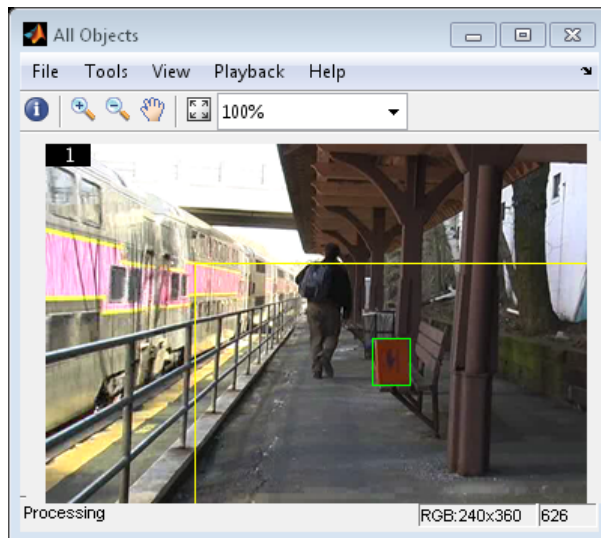
BBoxOffset = BBox + int32(repmat([roi(1) roi(2) 0 0],[BlobCount 1]));
Imr = step(hDrawRectangles2, Im, BBoxOffset);

% 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

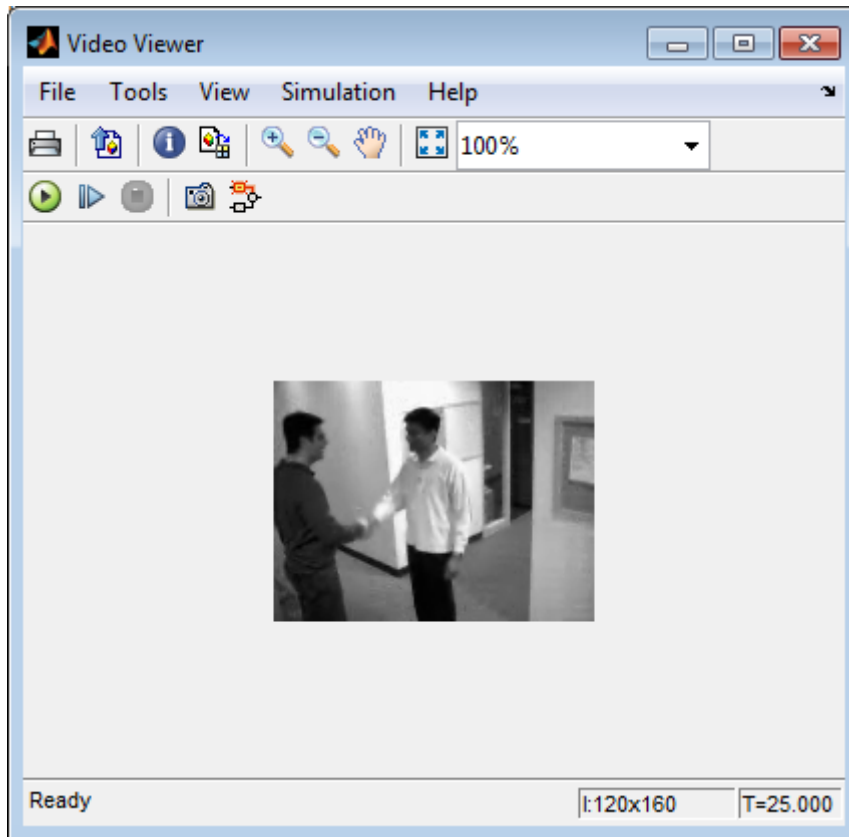
You can use the `vision.TextInserter` System object in MATLAB, or the `Insert Text` block in a Simulink model, to overlay text on video streams. In this Simulink model example, you add a running count of the number of video frames to a video using the `Insert Text` block. The model contains the `From 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_file_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

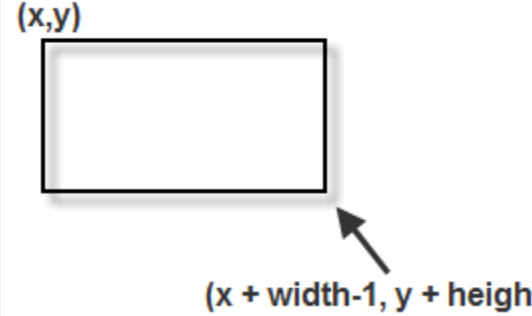
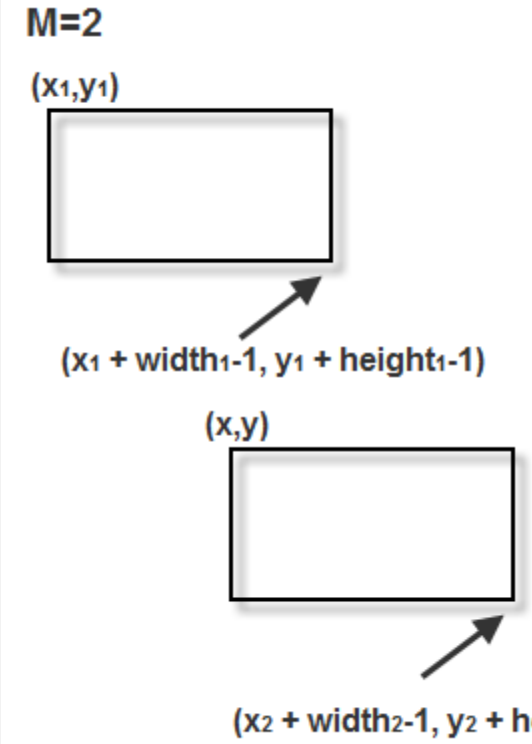
Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

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

Draw Shapes and Lines

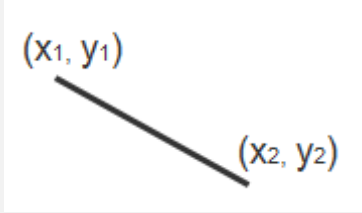
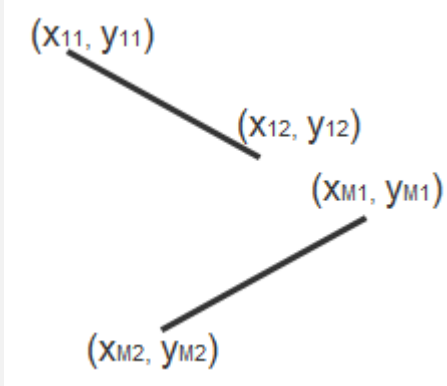
When you specify the type of shape to draw, you must also specify its location on the image. The table shows the format for the points input for the different shapes.

Rectangle

Shape	PTS input	Drawn Shape
Single Rectangle	<p>Four-element row vector $[x \ y \ width \ height]$ where</p> <ul style="list-style-type: none"> • x and y are the one-based coordinates of the upper-left corner of the rectangle. • $width$ and $height$ are the width, in pixels, and height, in pixels, of the rectangle. The values of $width$ and $height$ must be greater than 0. 	 <p>(x,y)</p> <p>$(x + width-1, y + height-1)$</p>
M Rectangles	<p>M-by-4 matrix</p> $\begin{bmatrix} x_1 & y_1 & width_1 & height_1 \\ x_2 & y_2 & width_2 & height_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_M & y_M & width_M & height_M \end{bmatrix}$ <p>where each row of the matrix corresponds to a different rectangle and is of the same form as the vector for a single rectangle.</p>	 <p>$M=2$</p> <p>(x_1,y_1)</p> <p>$(x_1 + width_1-1, y_1 + height_1-1)$</p> <p>$(x,y)$</p> <p>$(x_2 + width_2-1, y_2 + height_2-1)$</p>

Line and Polyline

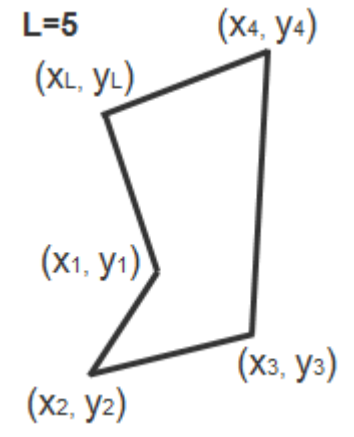
You can draw one or more lines, and one or more polylines. A polyline contains a series of connected line segments.

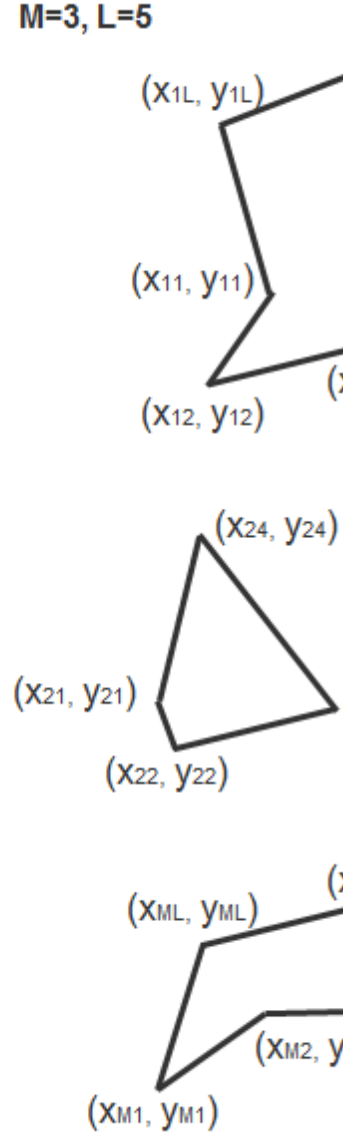
Shape	PTS input	Drawn Shape
Single Line	Four-element row vector $[x_1 \ y_1 \ x_2 \ y_2]$ where <ul style="list-style-type: none"> x_1 and y_1 are the coordinates of the beginning of the line. x_2 and y_2 are the coordinates of the end of the line. 	
M Lines	M -by-4 matrix $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} \\ x_{21} & y_{21} & x_{22} & y_{22} \\ \vdots & \vdots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} \end{bmatrix}$ where each row of the matrix corresponds to a different line and is of the same form as the vector for a single line.	

Shape	PTS input	Drawn Shape
<p>Single Polyline with $(L-1)$ Segments</p>	<p>Vector of size $2L$, where L is the number of vertices, with format, $[x_1, y_1, x_2, y_2, \dots, x_L, y_L]$.</p> <ul style="list-style-type: none"> x_1 and y_1 are the coordinates of the beginning of the first line segment. x_2 and y_2 are the coordinates of the end of the first line segment and the beginning of the second line segment. x_L and y_L are the coordinates of the end of the $(L-1)^{\text{th}}$ line segment. <p>The polyline always contains $(L-1)$ number of segments because the first and last vertex points do not connect. The block produces an error message when the number of rows is less than two or not a multiple of two.</p>	
<p>M Polylines with $(L-1)$ Segments</p>	<p>$2L$-by-N matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different polyline and is of the same form as the vector for a single polyline. When you require one polyline to contain less than $(L-1)$ number of segments, fill the matrix by repeating the coordinates of the last vertex.</p> <p>The block produces an error message if the number of rows is less than two or not a multiple of two.</p>	

Polygon

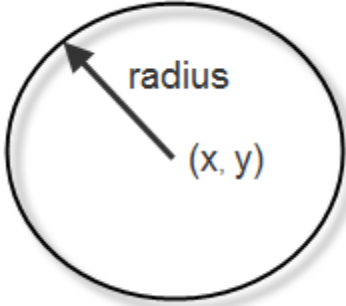
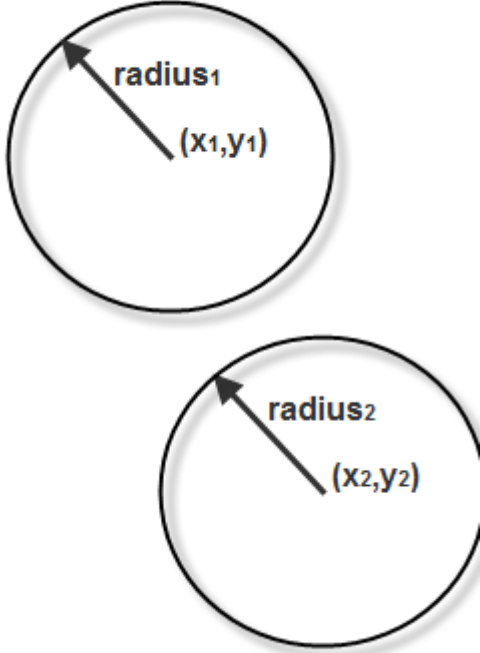
You can draw one or more polygons.

Shape	PTS input	Drawn Shape
Single Polygon with L line segments	<p>Row vector of size $2L$, where L is the number of vertices, with format, $[x_1 \ y_1 \ x_2 \ y_2 \ \dots \ x_L \ y_L]$ where</p> <ul style="list-style-type: none"> x_1 and y_1 are the coordinates of the beginning of the first line segment. x_2 and y_2 are the coordinates of the end of the first line segment and the beginning of the second line segment. x_L and y_L are the coordinates of the end of the $(L-1)^{\text{th}}$ line segment and the beginning of the L^{th} line segment. <p>The block connects $[x_1 \ y_1]$ to $[x_L \ y_L]$ to complete the polygon. The block produces</p>	

Shape	PTS input	Drawn Shape
	an error if the number of rows is negative or not a multiple of two.	
<p>M Polygons with the largest number of line segments in any line being L</p>	<p>M-by-$2L$ matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different polygon and is of the same form as the vector for a single polygon. If some polygons are shorter than others, repeat the ending coordinates to fill the polygon matrix.</p> <p>The block produces an error message if the number of rows is less than two or is not a multiple of two.</p>	<p>M=3, L=5</p> 

Circle

You can draw one or more circles.

Shape	PTS input	Drawn Shape
Single Circle	<p>Three-element row vector $[x \ y \ radius]$ where</p> <ul style="list-style-type: none"> • x and y are coordinates for the center of the circle. • $radius$ is the radius of the circle, which must be greater than 0. 	 <p>A diagram showing a single circle. An arrow points from the center, labeled (x, y), to the circumference. The arrow is labeled "radius".</p>
M Circles	<p>M-by-3 matrix</p> $\begin{bmatrix} x_1 & y_1 & radius_1 \\ x_2 & y_2 & radius_2 \\ \vdots & \vdots & \vdots \\ x_M & y_M & radius_M \end{bmatrix}$ <p>where each row of the matrix corresponds to a different circle and is of the same form as the vector for a single circle.</p>	<p>$M=2$</p>  <p>A diagram showing two circles. The top circle has center (x_1, y_1) and radius $radius_1$. The bottom circle has center (x_2, y_2) and radius $radius_2$. The label $M=2$ is positioned above the top circle.</p>

Registration and Stereo Vision

- “Computer Vision Workflow” on page 3-2
- “Feature Detection, Extraction, and Matching” on page 3-3
- “Stereo Image Rectification” on page 3-42
- “Stereo Vision” on page 3-54
- “Find Fundamental Matrix Describing Epipolar Geometry” on page 3-71

Computer Vision Workflow

XXXXXXXX — work in progress

Feature Detection, Extraction, and Matching

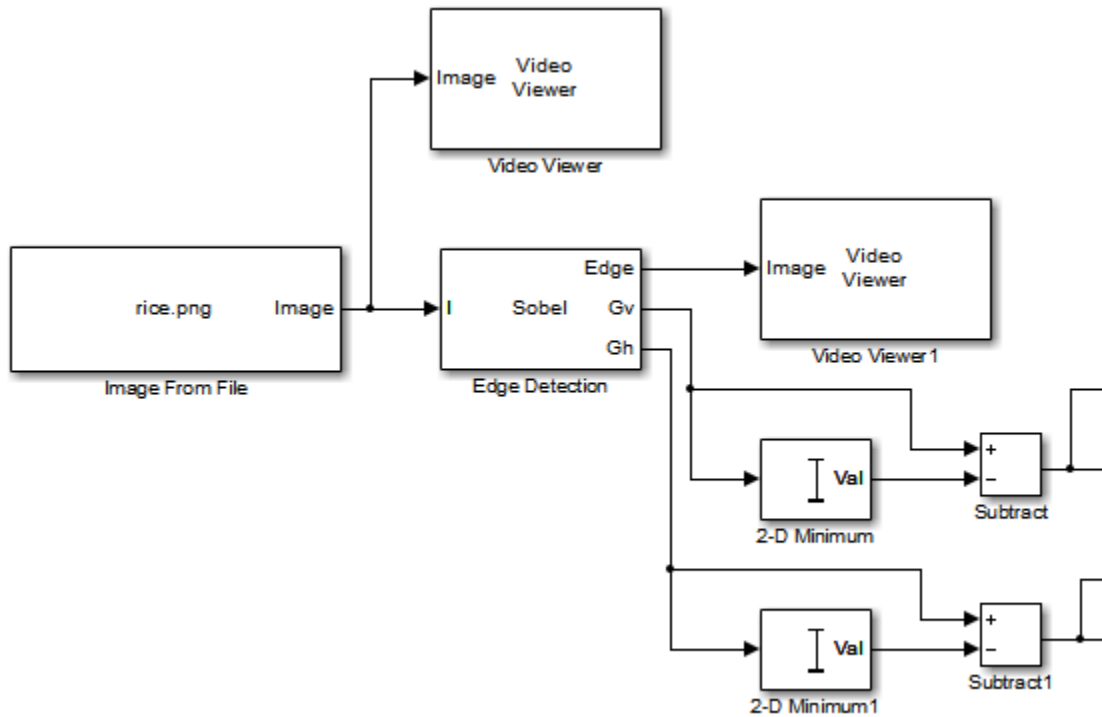
In this section...
“Detect Edges in Images” on page 3-3
“Detect Lines in Images” on page 3-11
“Video Stabilization Using Point Feature Matching” on page 3-15
“Video Stabilization” on page 3-28
“Measure Angle Between Lines” on page 3-33

Detect Edges in Images

This example shows how to find 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`



Set block parameters

Block	Parameter setting
Image From File	<ul style="list-style-type: none"> • File name to rice.png. • Output data type to single.
Edge Detection	<p>Use the Edge Detection block to find the edges in the image.</p> <ul style="list-style-type: none"> • 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.

Set configuration parameters

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

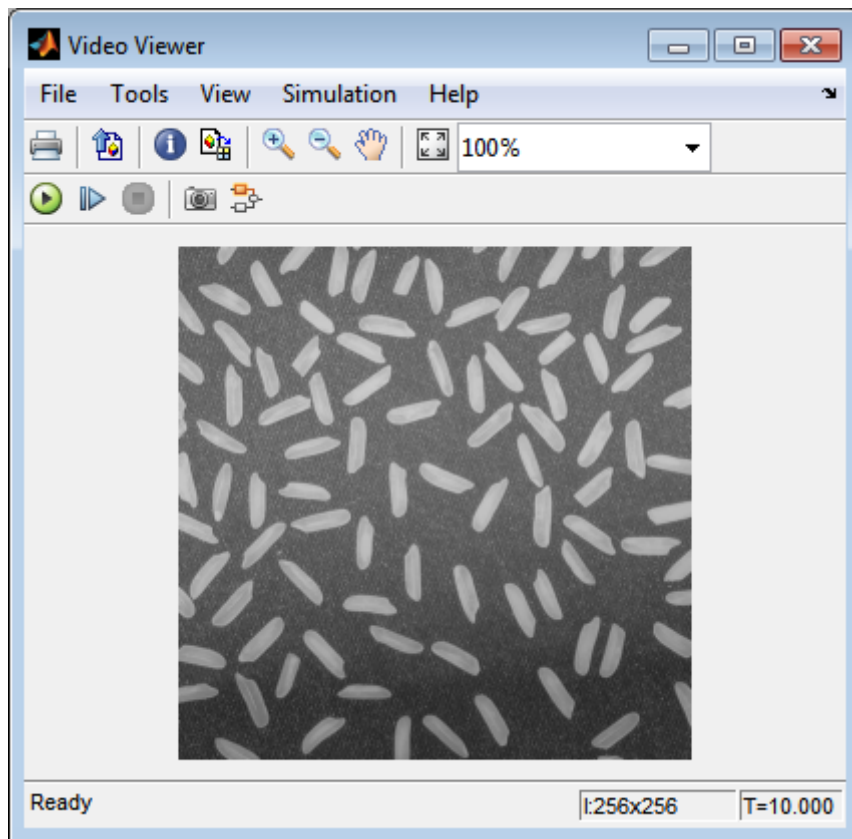
- Solver pane, **Stop time** = 0
- Solver pane, **Type** = Fixed-step
- Solver pane, **Solver** = Discrete (no continuous states)

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

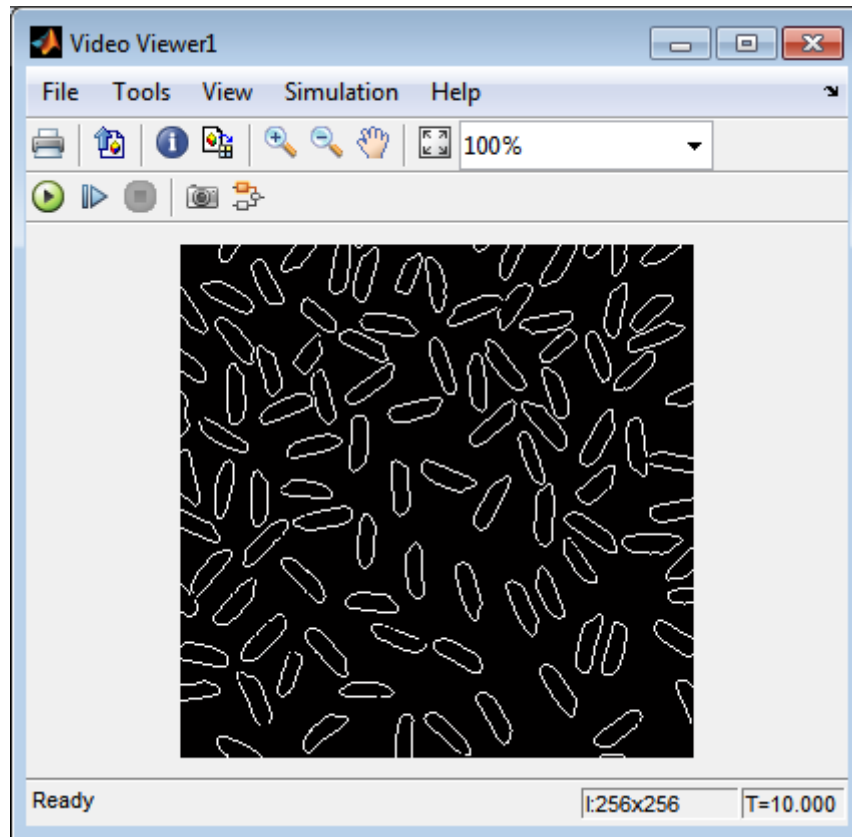
View edge detection results

Run your model.

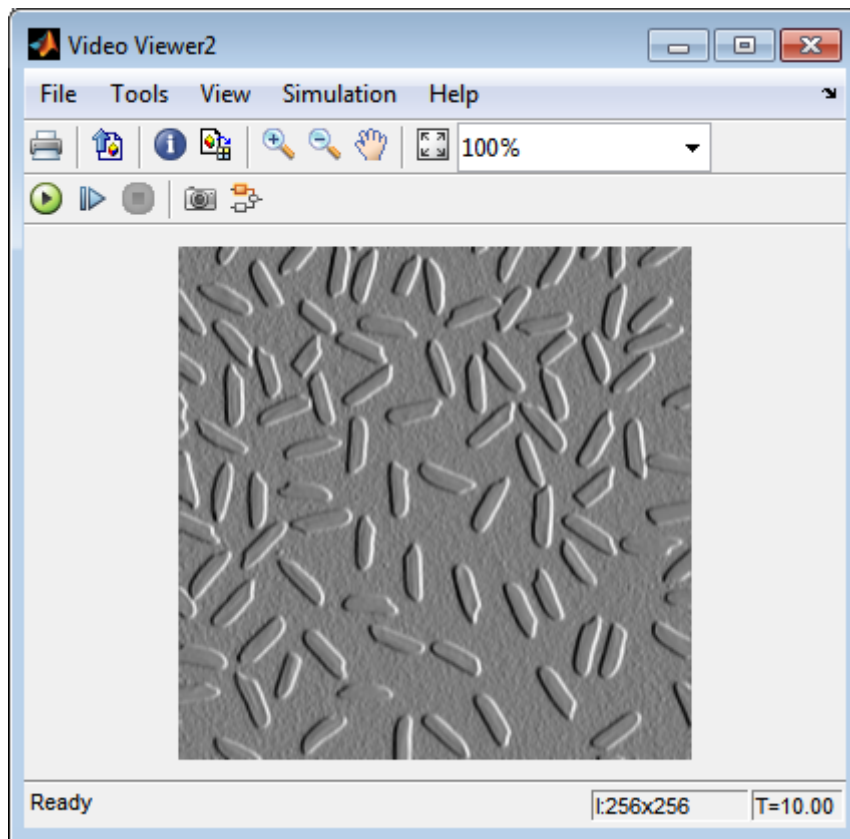
The Video Viewer window displays the original image.



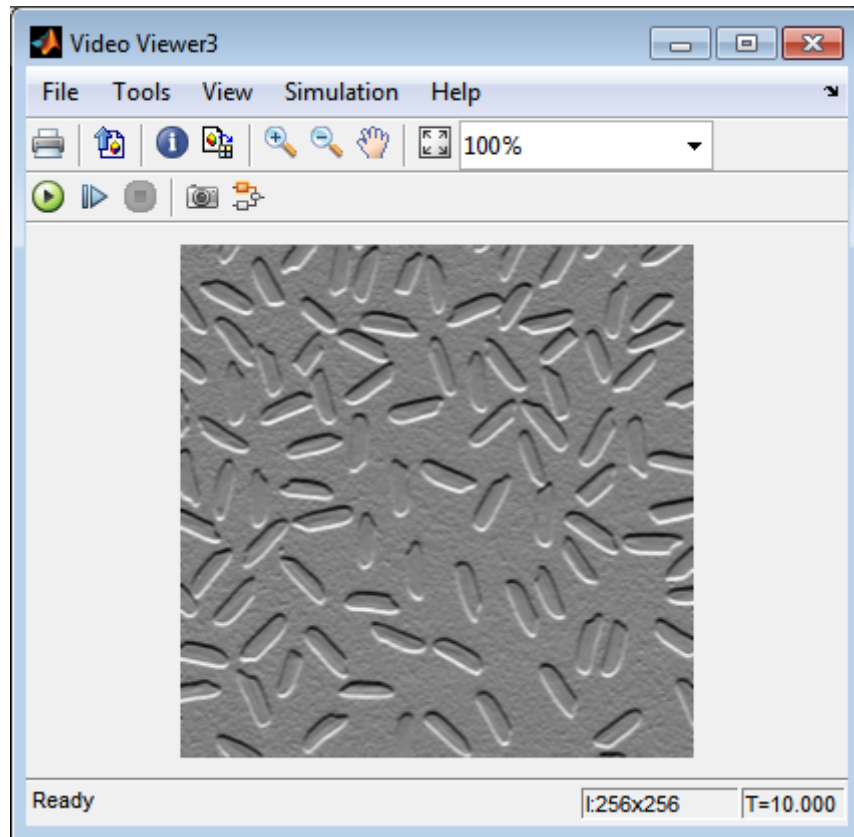
The Video Viewer1 window displays the edges of the rice grains in white and the background in black.



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 **G_h** and **G_v** 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 **G_v** and **G_h** output ports of the Edge Detection block are double-precision floating-point. These matrix values need to be scaled

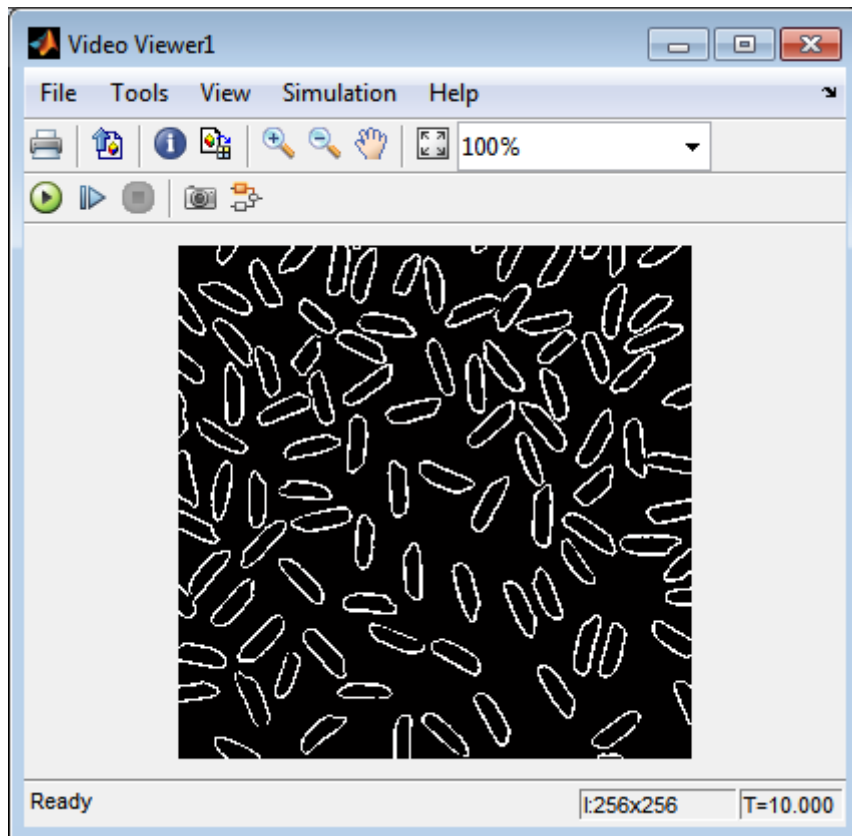
between 0 and 1 in order to display them using the Video Viewer blocks. This is done with the Statistics and Math Operation blocks.

Run 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*.

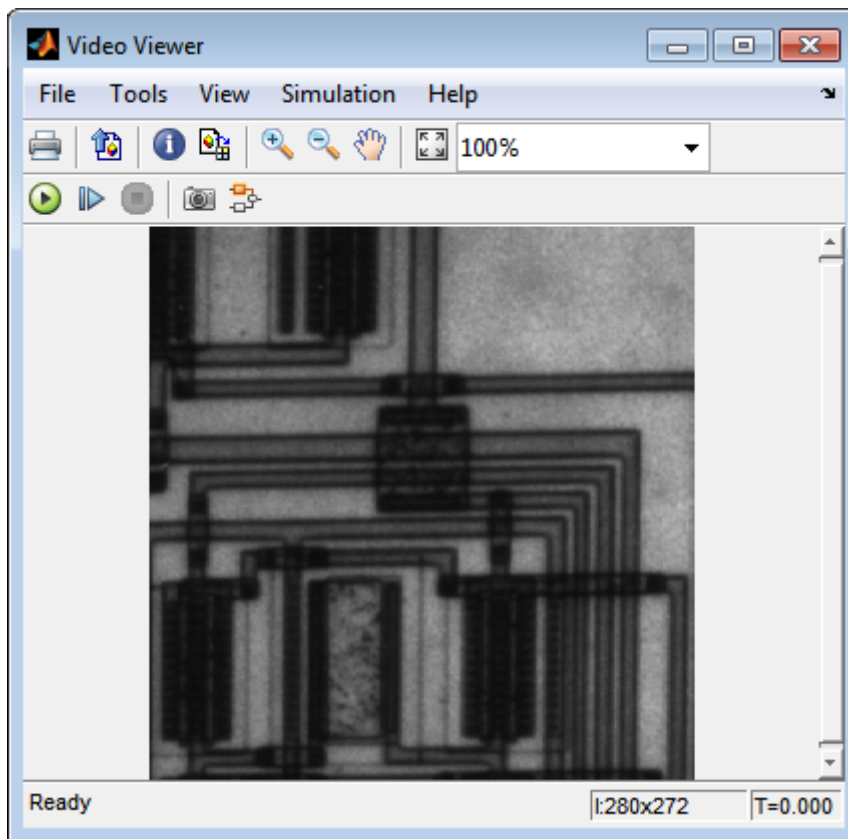
Detect Lines in Images

Finding lines within images enables you to detect, measure, and recognize objects. In this example, you use the Hough Transform, Find Local Maxima, Edge Detection and 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
```

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 examples. You can open these examples by typing the title at the MATLAB command prompt:

Example	MATLAB	Simulink model-based
Lane Departure Warning System	videoldws	vipldws
Rotation Correction	videorotationcorrectio	viiphough

Setting Block Parameters

Block	Parameter setting
Hough Transform	<p>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:</p> <ul style="list-style-type: none"> • Theta resolution (radians) = $\pi/360$ • Select the Output theta and rho values check box.
Find Local Maxima	<p>The Find Local Maxima block finds the location of the maximum value in the Hough matrix. The block parameters are set as follows:</p> <ul style="list-style-type: none"> • Maximum number of local maxima = 1 • Input is Hough matrix spanning full theta range

Block	Parameter setting
Selector	<p>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:</p> <ul style="list-style-type: none"> • Number of input dimensions: 1 • Index mode = One-based • Index Option = Index vector (port) • Input port size = 2
Variable Selector	<p>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:</p> <ul style="list-style-type: none"> • 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. <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • Shape = Lines • Border color = White

Configuration Parameters

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)
- Solver pane, **Fixed-step size (fundamental sample time)**: = 0.2

Video Stabilization Using Point Feature Matching

This example shows how to stabilize a video that was captured from a jittery platform. 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 will use System objects in the Computer Vision System Toolbox™, both for the algorithm and for display.

This example is similar to the Video Stabilization Example. The main difference is that the Video Stabilization Example is given a region to track while this example is given no such knowledge. Both examples use the same video.

Step 1. Read Frames from a Movie File

Here we read in the first two frames of a video sequence. We read them as intensity images since color is not necessary for the stabilization algorithm, and because using grayscale images improves speed. Below we show both frames side by side, and we produce a red-cyan color composite to illustrate the pixel-wise difference between them. There is obviously a large vertical and horizontal offset between the two frames.

```
filename = 'shaky_car.avi';
hVideoSrc = vision.VideoFileReader(filename, ...
    'ImageColorSpace', 'Intensity',...
    'VideoOutputDataType', 'single');

imgA = step(hVideoSrc); % Read first frame into imgA
imgB = step(hVideoSrc); % Read second frame into imgB

figure; imshowpair(imgA, imgB, 'montage');
title(['Frame A', repmat(' ', [1 70]), 'Frame B']);
```



```
figure; imshowpair(imgA,imgB,'ColorChannels','red-cyan');  
title('Color composite (frame A = red, frame B = cyan)');
```

Color composite (frame A = red, frame B = cyan)



Step 2. Collect Salient Points from Each Frame

Our goal is to determine a transformation that will correct for the distortion between the two frames. We can use the `GeometricTransformEstimator`

System object for this, which will return an affine transform. As input we must provide this object with a set of point correspondences between the two frames. To generate these correspondences, we first collect points of interest from both frames, then select likely correspondences between them.

In this step we produce these candidate points for each frame. To have the best chance that these points will have corresponding points in the other frame, we want points around salient image features such as corners. For this we use the `CornerDetector` System object. The FAST corner detector algorithm, which we have selected below is one of the fastest options.

The detected points from both frames are shown in the figure below. Observe how many of them cover the same image features, such as points along the tree line, the corners of the large road sign, and the corners of the cars.

```
maxPts = 150;
ptThresh = 1e-3;
hCD = vision.CornerDetector( ...
    'Method','Local intensity comparison (Rosen & Drummond)', ...
    'MaximumCornerCount', maxPts, ...
    'CornerThreshold', ptThresh, ...
    'NeighborhoodSize', [9 9]);
pointsA = step(hCD, imgA);
pointsB = step(hCD, imgB);

% Display corners found in images A and B.
figure; imshow(imgA); hold on;
plot(pointsA(:,1), pointsA(:,2), 'o', 'MarkerSize', 4, ...
    'MarkerEdgeColor', 'w', 'MarkerFaceColor', 'r');
title('Corners in A');

figure; imshow(imgB); hold on;
plot(pointsB(:,1), pointsB(:,2), 'o', 'MarkerSize', 4, ...
    'MarkerEdgeColor', 'w', 'MarkerFaceColor', 'r');
title('Corners in B');
```

Corners in A



Corners in B



Step 3. Select Correspondences Between Points

Next we pick correspondences between the points derived above. For each point, we extract a 9-by-9 block centered around it. The matching cost we use between points is the sum of squared differences (SSD) between their respective image regions. Points in frame A and frame B are matched

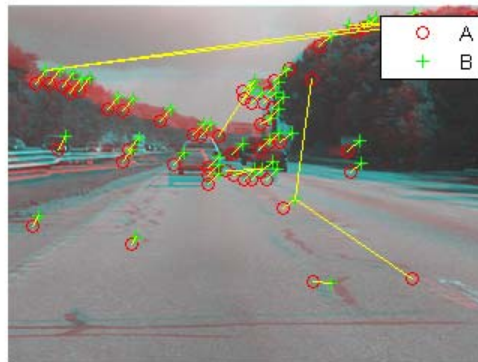
putatively. Note that there is no uniqueness constraint, so points from frame B can correspond to multiple points in frame A.

```
% Extract features for the corners
blockSize = 9; % Block size.
[featuresA, pointsA] = extractFeatures(imgA, pointsA, ...
    'BlockSize', blockSize);
[featuresB, pointsB] = extractFeatures(imgB, pointsB, ...
    'BlockSize', blockSize);

% Match features which were found in the current and the previous frames
indexPairs = matchFeatures(featuresA, featuresB, 'Metric', 'SSD');
pointsA = pointsA(indexPairs(:, 1), :);
pointsB = pointsB(indexPairs(:, 2), :);
```

The image below shows the same color composite given above, but added are the points from frame A in red, and the points from frame B in green. Yellow lines are drawn between points to show the correspondences selected by the above procedure. Many of these correspondences are correct, but there is also a significant number of outliers.

```
figure; showMatchedFeatures(imgA, imgB, pointsA, pointsB);
legend('A', 'B');
```



Step 4. Estimating Transform from Noisy Correspondences

Many of the point correspondences obtained in the previous step are incorrect. But we can still derive a robust estimate of the geometric transform between the two images using the Random Sample Consensus (RANSAC) algorithm [1], which is implemented in the `GeometricTransformEstimator` System object. This object, when given a set of point correspondences, will search for the valid inlier correspondences. From these it will then derive the affine transform that makes the inliers from the first set of points match most closely with the inliers from the second set. This affine transform will be a 3-by-3 matrix of the form:

$$\begin{bmatrix} a_1 & a_3 & t_r; \\ a_2 & a_4 & t_c; \\ 0 & 0 & 1 \end{bmatrix}$$

The parameters a define scale, rotation, and sheering effects of the transform, while the parameters t are translation parameters. This transform can be used to warp the images such that their corresponding features will be moved to the same image location.

A limitation of the affine transform is that it can only alter the imaging plane. Thus it is ill-suited to finding the general distortion between two frames taken of a 3-D scene, such as with this video taken from a moving car. But it does work under certain conditions that we shall describe shortly.

We implement this procedure below. For added robustness, we run the `GeometricTransformEstimator` System object multiple times and calculate a cost for each result. This cost is obtained by projecting frame B onto frame A according to the derived transform, and taking the sum of absolute difference (SAD) between the two images. We take the best transform as the one that minimizes this cost.

```
hGTE = vision.GeometricTransformEstimator;
hGT = vision.GeometricTransformer;
hGTPrj = vision.GeometricTransformer;

% Run multiple RANSAC trials
nRansacTrials = 1;
Ts = cell(1,nRansacTrials);
costs = zeros(1,nRansacTrials);
```

```
nPts = int32(size(pointsA,2));
inliers = cell(1,nRansacTrials);

for j=1:nRansacTrials
    % Estimate affine transform
    [Ts{j},inliers{j}] = step(hGTE, pointsB, pointsA);

    % Warp image and compute error metric.
    imgBp = step(hGT, imgB, Ts{j});
    costs(j) = sum(sum(imabsdiff(imgBp, imgA)));
end
% Take best result.
[~,ix] = min(costs);
imgBp = step(hGT, imgB, Ts{ix});
pointsBp = [single(pointsB), ones(size(pointsB,1), 1)] * Ts{ix};
H = [Ts{ix} [0 0 1]'];
```

Below is a color composite showing frame A overlaid with the reprojected frame B, along with the reprojected point correspondences. The results are excellent, with the inlier correspondences nearly exactly coincident. The cores of the images are both well aligned, such that the red-cyan color composite becomes almost purely black-and-white in that region.

Note how the inlier correspondences are all in the background of the image, not in the foreground, which itself is not aligned. This is because the background features are distant enough that they behave as if they were on an infinitely distant plane. Thus, even though the affine transform is limited to altering only the imaging plane, here that is sufficient to align the background planes of both images. Furthermore, if we assume that the background plane has not moved or changed significantly between frames, then this transform is actually capturing the camera motion. Therefore correcting for this will stabilize the video. This condition will hold as long as the motion of the camera between frames is small enough, or, conversely, if the sample time of the video is high enough.

```
figure;
showMatchedFeatures(imgA, imgBp, pointsA(inliers{ix},:), pointsBp(inliers{i
legend('A', 'B');
```




Step 5. Transform Approximation and Smoothing

Given a set of video frames T_i , $i = 0, 1, 2, \dots$, we can now use the above procedure to estimate the distortion between all frames T_i and T_{i+1} as affine transforms, H_i . Thus the cumulative distortion of a frame i relative to the first frame will be the product of all the preceding inter-frame transforms, or

$$H_{cumulative,i} = \prod_{j=0}^{i-1} H_j$$

We could use all the six parameters of the affine transform above, but, for numerical simplicity and stability, we choose to re-fit the matrix as a simpler scale-rotation-translation transform. This has only four free parameters compared to the full affine transform's six: one scale factor, one angle, and two translations. This new transform matrix is of the form:

$$\begin{bmatrix} s \cdot \cos(\text{ang}) & s \cdot \sin(\text{ang}) & t_x \\ s \cdot \sin(\text{ang}) & s \cdot \cos(\text{ang}) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

We show this conversion procedure below by fitting the above-obtained transform H with a scale-rotation-translation equivalent, H_{sRt} . To show that the error of converting the transform is minimal, we reproject frame B with both transforms and show the two images below as a red-cyan color

composite. As the image appears black and white, obviously the pixel-wise difference between the different reprojections is negligible.

```
% Extract scale and rotation part sub-matrix.
R = H(1:2,1:2);
% Compute theta from mean of two possible arctangents
theta = mean([atan2(R(2),R(1)) atan2(-R(3),R(4))]);
% Compute scale from mean of two stable mean calculations
scale = mean(R([1 4])/cos(theta));
% Translation remains the same:
translation = H(3, 1:2);
% Reconstitute new s-R-t transform:
HsRt = [[scale*[cos(theta) -sin(theta); sin(theta) cos(theta)]; ...
        translation], [0 0 1]'];

imgBold = step(hGTPrj, imgB, H);
imgBsRt = step(hGTPrj, imgB, HsRt);
figure(2), clf;
imshowpair(imgBold,imgBsRt,'ColorChannels','red-cyan'), axis image;
title('Color composite of affine and s-R-t transform outputs');
```

Color composite of affine and s-R-t transform outputs



Step 6. Run on the Full Video

Now we apply the above steps to smooth a video sequence. For readability, the above procedure of estimating the transform between two images has been placed in the MATLAB function `cvxEstStabilizationTform`. The function `cvxTformToSRT` also converts a general affine transform into a scale-rotation-translation transform.

At each step we calculate the transform H between the present frames. We fit this as an s-R-t transform, H_{sRt} . Then we combine this the cumulative transform, $H_{cumulative}$, which describes all camera motion since the first frame. The last two frames of the smoothed video are shown in a Video Player as a red-cyan composite.

With this code, you can also take out the early exit condition to make the loop process the entire video.

```
% Reset the video source to the beginning of the file.
reset(hVideoSrc);
hGTE = vision.GeometricTransformEstimator;
hGT = vision.GeometricTransformer;
hGTPrj = vision.GeometricTransformer;

hVPlayer = vision.VideoPlayer; % Create video viewer

hCD = vision.CornerDetector( ...
    'Method','Local intensity comparison (Rosen & Drummond)', ...
    'MaximumCornerCount', maxPts, ...
    'CornerThreshold', ptThresh, ...
    'NeighborhoodSize', [9 9]);

% Process all frames in the video
movMean = step(hVideoSrc);
imgB = movMean;
imgBp = imgB;
correctedMean = imgBp;
ii = 2;
Hcumulative = eye(3);
while ~isDone(hVideoSrc) && ii < 10
    % Read in new frame
    imgA = imgB; % z^-1
    imgAp = imgBp; % z^-1
```

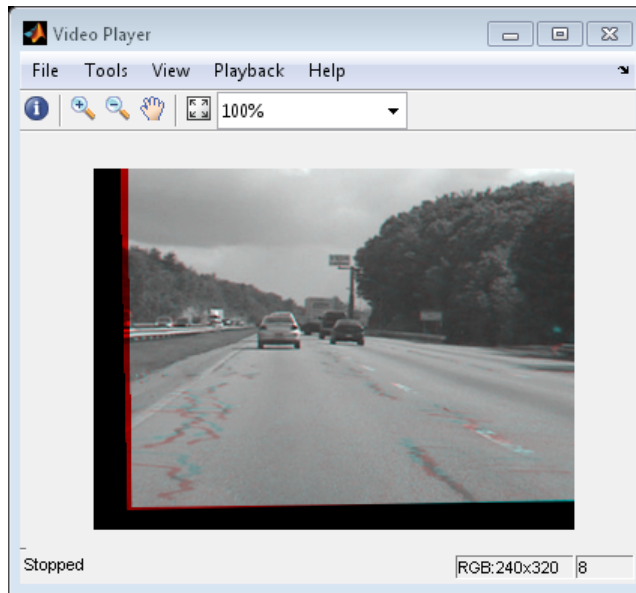
```
imgB = step(hVideoSrc);
movMean = movMean + imgB;

% Estimate transform from frame A to frame B, and fit as an s-R-t
H = cvxEstStabilizationTform(imgA,imgB,hCD,hGT,hGTE);
HsRt = cvxTformToSRT(H);
Hcumulative = HsRt * Hcumulative;
imgBp = step(hGTPrj, imgB, Hcumulative);

% Display as color composite with last corrected frame
step(hVPlayer, imfuse(imgAp,imgBp,'ColorChannels','red-cyan'));
correctedMean = correctedMean + imgBp;

ii = ii+1;
end
correctedMean = correctedMean/(ii-2);
movMean = movMean/(ii-2);

% Here you call the release method on the objects to close any open files
% and release memory.
release(hVideoSrc);
release(hVPlayer);
```



During computation, we computed the mean of the raw video frames and of the corrected frames. These mean values are shown side-by-side below. The left image shows the mean of the raw input frames, proving that there was a great deal of distortion in the original video. The mean of the corrected frames on the right, however, shows the image core with almost no distortion. While foreground details have been blurred (as a necessary result of the car's forward motion), this shows the efficacy of the stabilization algorithm.

```
figure; imshowpair(movMean, correctedMean, 'montage');
title(['Raw input mean', repmat(' ', [1 50]), 'Corrected sequence mean']);
```



References

[1] Tordoff, B; Murray, DW. "Guided sampling and consensus for motion estimation." European Conference n Computer Vision, 2002.

[2] Lee, KY; Chuang, YY; Chen, BY; Ouhyoung, M. "Video Stabilization using Robust Feature Trajectories." National Taiwan University, 2009.

[3] Litvin, A; Konrad, J; Karl, WC. "Probabilistic video stabilization using Kalman filtering and mosaicking." IS&T/SPIE Symposium on Electronic Imaging, Image and Video Communications and Proc., 2003.

[4] Matsushita, Y; Ofek, E; Tang, X; Shum, HY. "Full-frame Video Stabilization." Microsoft Research Asia. CVPR 2005.

Video Stabilization

This example shows how to remove the effect of camera motion from a video stream.

Introduction

In this example we first define the target to track. In this case, it is the back of a car and the license plate. We also establish a dynamic search region,

whose position is determined by the last known target location. We then search for the target only within this search region, which reduces the number of computations required to find the target. In each subsequent video frame, we determine how much the target has moved relative to the previous frame. We use this information to remove unwanted translational camera motions and generate a stabilized video.

Initialization

Create a System object to read video from a multimedia file. We set the output to be of intensity only video.

```
% Input video file which needs to be stabilized.
filename = 'shaky_car.avi';

hVideoSource = vision.VideoFileReader(filename, ...
                                     'ImageColorSpace', 'Intensity',...
                                     'VideoOutputDataType', 'double');
```

Create a geometric translator System object used to compensate for the camera movement.

```
hTranslate = vision.GeometricTranslator( ...
                                     'OutputSize', 'Same as input image', ...
                                     'OffsetSource', 'Input port');
```

Create a template matcher System object to compute the location of the best match of the target in the video frame. We use this location to find translation between successive video frames.

```
hTM = vision.TemplateMatcher('ROIInputPort', true, ...
                             'BestMatchNeighborhoodOutputPort', true);
```

Create a System object to highlight the search region and target in the video by drawing rectangles around them.

```
hShapeInserter = vision.ShapeInserter('BorderColor', 'White');
```

Create a System object to show the displacement values on the video.

```
hTextInserter = vision.TextInserter('(%+05.1f,%+05.1f)', ...
```

```
'Location', [191 215], ...  
'Color', 1, ...  
'FontSize', 16);
```

Create a System object to display the original video and the stabilized video.

```
hVideoOut = vision.VideoPlayer('Name', 'Video Stabilization');  
hVideoOut.Position(1) = round(0.4*hVideoOut.Position(1));  
hVideoOut.Position(2) = round(1.5*(hVideoOut.Position(2)));  
hVideoOut.Position(3:4) = [650 350];
```

Here we initialize some variables used in the processing loop.

```
pos.template_orig = [109 100]; % [x y] upper left corner  
pos.template_size = [22 18]; % [width height]  
pos.search_border = [15 10]; % max horizontal and vertical displacement  
pos.template_center = floor((pos.template_size-1)/2);  
pos.template_center_pos = (pos.template_orig + pos.template_center - 1);  
fileInfo = info(hVideoSource);  
W = fileInfo.VideoSize(1); % Width in pixels  
H = fileInfo.VideoSize(2); % Height in pixels  
BorderCols = [1:pos.search_border(1)+4 W-pos.search_border(1)+4:W];  
BorderRows = [1:pos.search_border(2)+4 H-pos.search_border(2)+4:H];  
sz = fileInfo.VideoSize;  
TargetRowIndices = ...  
    pos.template_orig(2)-1:pos.template_orig(2)+pos.template_size(2)-2;  
TargetColIndices = ...  
    pos.template_orig(1)-1:pos.template_orig(1)+pos.template_size(1)-2;  
SearchRegion = pos.template_orig - pos.search_border - 1;  
Offset = [0 0];  
Target = zeros(18,22);  
firstTime = true;
```

Stream Processing Loop

This is the main processing loop which uses the objects we instantiated above to stabilize the input video.

```
while ~isDone(hVideoSource)  
    input = step(hVideoSource);
```



```

% Find location of Target in the input video frame
if firstTime
    Idx = int32(pos.template_center_pos);
    MotionVector = [0 0];
    firstTime = false;
else
    IdxPrev = Idx;

    ROI = [SearchRegion, pos.template_size+2*pos.search_border];
    Idx = step(hTM, input, Target, ROI);

    MotionVector = double(Idx-IdxPrev);
end

[Offset, SearchRegion] = updatesearch(sz, MotionVector, ...
    SearchRegion, Offset, pos);

% Translate video frame to offset the camera motion
Stabilized = step(hTranslate, input, fliplr(Offset));

Target = Stabilized(TargetRowIndices, TargetColIndices);

% Add black border for display
Stabilized(:, BorderCols) = 0;
Stabilized(BorderRows, :) = 0;

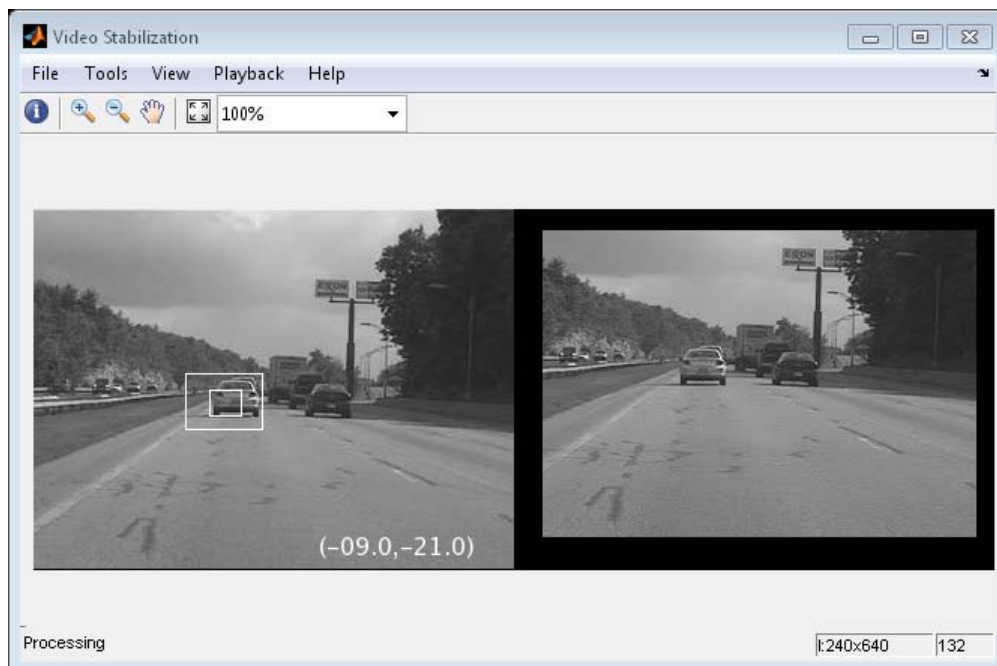
TargetRect = [pos.template_orig-Offset, pos.template_size];
SearchRegionRect = [SearchRegion, pos.template_size + 2*pos.search_bord

% Draw rectangles on input to show target and search region
input = step(hShapeInserter, input, [TargetRect; SearchRegionRect]);

% Display the offset values on the input image
input = step(hTextInserter, input, Offset);

% Display video
step(hVideoOut, [input Stabilized]);
end

```



Release

Here you call the release method on the objects to close any open files and devices.

```
release(hVideoSource);
```

Conclusion

Using the Computer Vision System Toolbox™ functionality from MATLAB command line it is easy to implement complex systems like video stabilization.

Appendix

The following helper function is used in this example.

- updatesearch.m

Measure 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.

`ex_vision_measure_angle_btwn_lines`

- 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
Edge Detection	Computer Vision System Toolbox > Analysis & Enhancement	1
Hough Transform	Computer Vision System Toolbox > Transforms	1
Hough Lines	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

Block	Library	Quantity
Submatrix	DSP System Toolbox > Math Functions > Matrices and Linear Algebra > Matrix Operations	4
Selector	Simulink > Signal Routing	4
MATLAB Function	Simulink > User-Defined Functions	1
Terminator	Simulink > Sinks	1
Display	Simulink > Sinks	1

- 2 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
- 3 Use the Color Space Conversion block to convert the RGB image into the YCbCr 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 YCbCr color space.

- 4 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

5 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

6 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.

7 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** checkbox.

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

- 8 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]
 - Select the **Input is Hough matrix spanning full theta range** checkbox.
 - Uncheck the **Output variable size signal** checkbox.

Because you are expecting two lines, leave the **Maximum number of local maxima (N)** parameter set to 2.

- 9 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.

- 10 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.

- 11** 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.

- 12** 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.

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

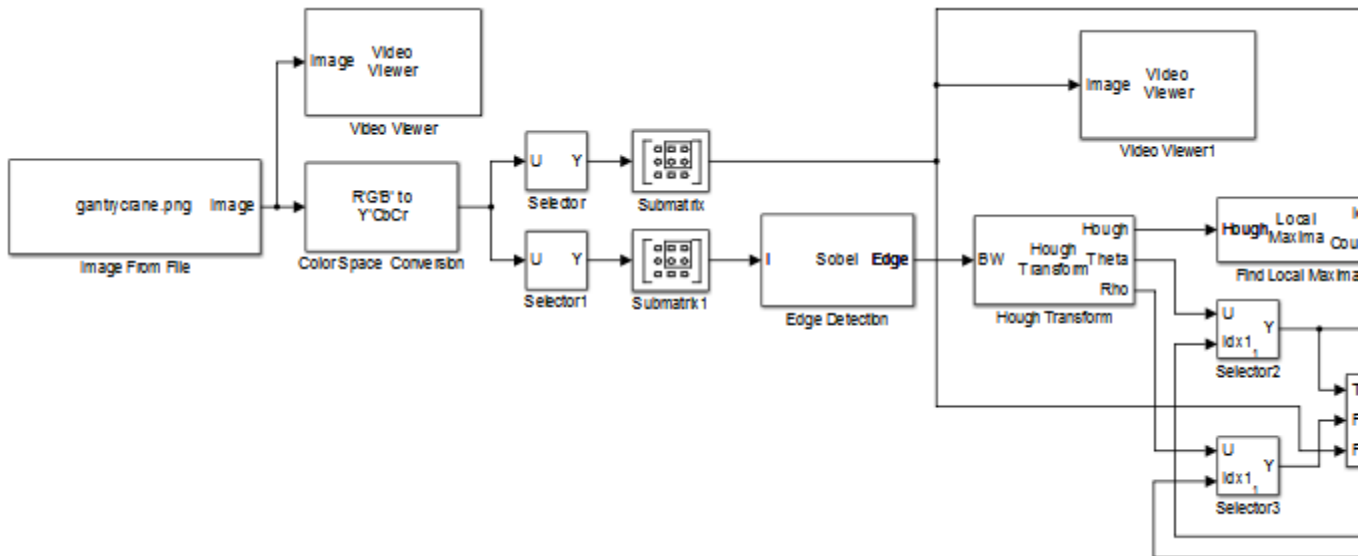
- **Shape** = Lines
- **Border color** = White

- 14** 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)

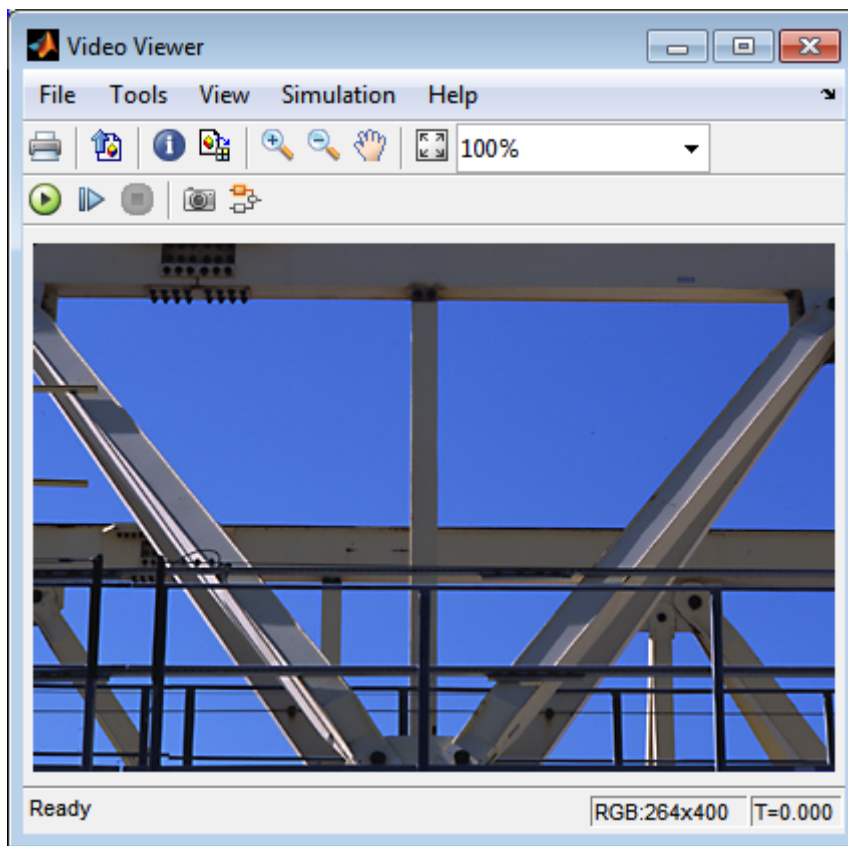
%Compute the angle value in degrees
angle = abs(theta(1)-theta(2))*180/pi;
%Always return an angle value less than 90 degrees
if (angle>90)
    angle = 180-angle;
end
```

- 15 Use the Display block to view the angle between the two lines. Accept the default parameters.
- 16 Use the Video Viewer blocks to view the original image, the ROI, and the annotated ROI. Accept the default parameters.
- 17 Connect the blocks as shown in the following figure.

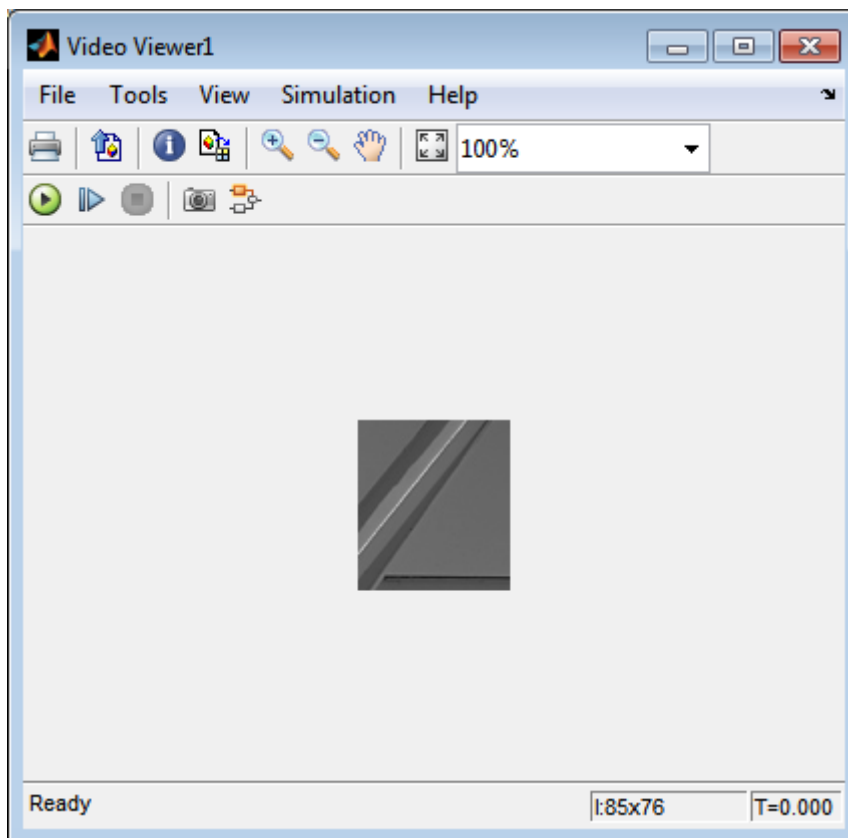


- 18 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:
 - **Solver** pane, **Stop time** = 0
 - **Solver** pane, **Type** = Fixed-step
 - **Solver** pane, **Solver** = Discrete (no continuous states)
- 19 Run the model.

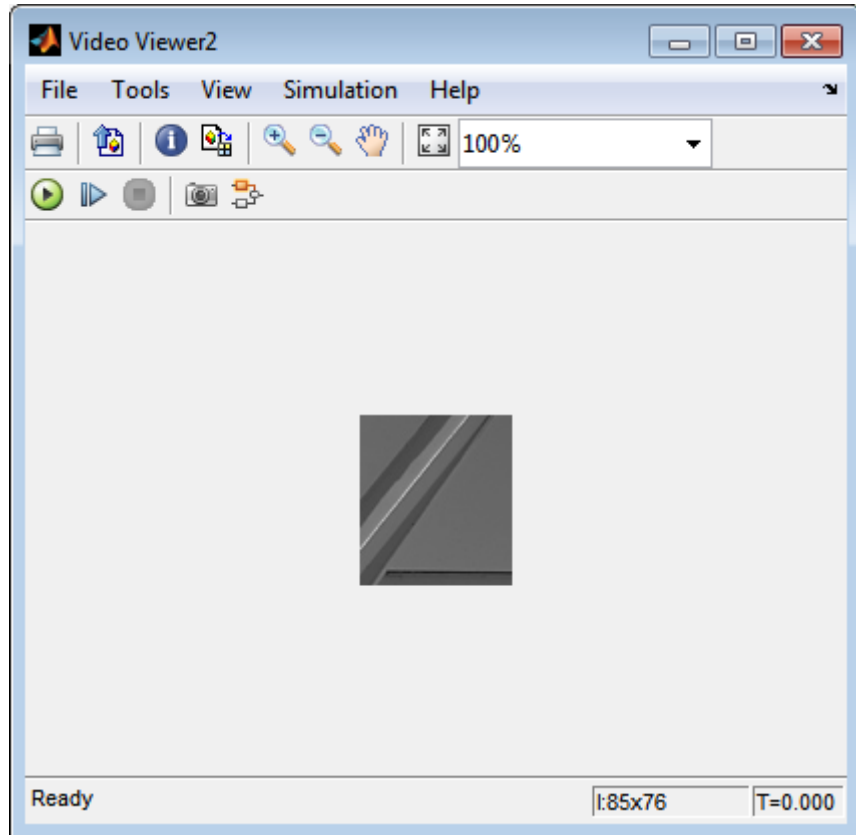
The Video Viewer window displays the original image.



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, which 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.

Stereo Image Rectification

This example shows how to use the `estimateFundamentalMatrix`, `estimateUncalibratedRectification`, and `detectSURFFeatures` functions to compute the rectification of two uncalibrated images, where the camera intrinsics are unknown.

Stereo image rectification projects images onto a common image plane in such a way that the corresponding points have the same row coordinates. This process is useful for stereo vision, because the 2-D stereo correspondence problem is reduced to a 1-D problem. As an example, stereo image rectification is often used as a pre-processing step for computing disparity or creating anaglyph images.

Step 1: Read Stereo Image Pair

Read in two color images of the same scene, which were taken from different positions. Then, convert them to grayscale. Colors are not required for the matching process.

```
I1 = im2double(rgb2gray(imread('yellowstone_left.png')));  
I2 = im2double(rgb2gray(imread('yellowstone_right.png')));
```

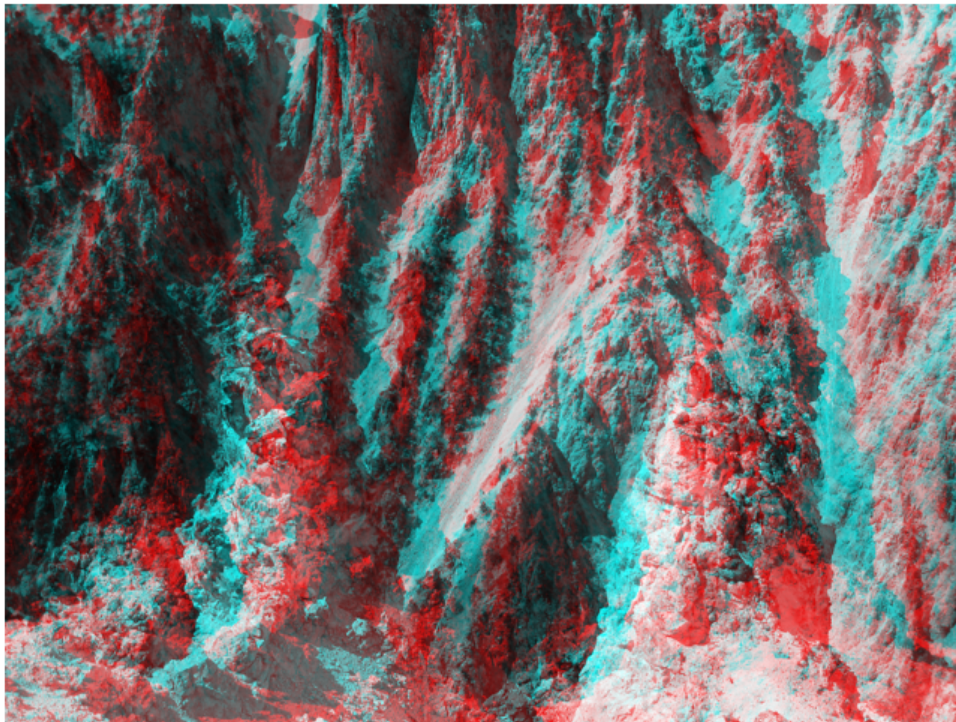
Display both images side by side. Then, display a color composite demonstrating the pixel-wise differences between the images.

```
imshowpair(I1, I2, 'montage');  
title('I1 (left); I2 (right)');  
figure; imshowpair(I1,I2, 'ColorChannels', 'red-cyan');  
title('Composite Image (Red - Left Image, Cyan - Right Image)');
```

I1 (left); I2 (right)



Composite Image (Red - Left Image, Cyan - Right Image)



There is an obvious offset between the images in orientation and position. The goal of rectification is to transform the images, aligning them such that corresponding points will appear on the same rows in both images.

Step 2: Collect Interest Points from Each Image

The rectification process requires a set of point correspondences between the two images. To generate these correspondences, you will collect points of interest from both images, and then choose potential matches between them. Use `detectSURFFeatures` to find blob-like features in both images.

```
blobs1 = detectSURFFeatures(I1, 'MetricThreshold', 2000);
```

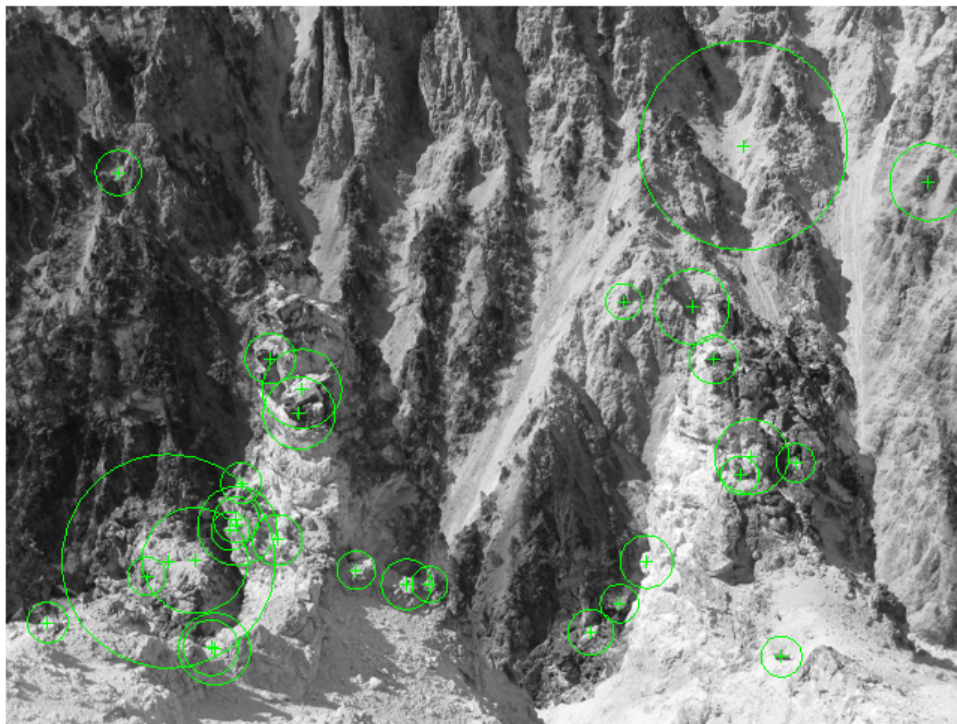
```
blobs2 = detectSURFFeatures(I2, 'MetricThreshold', 2000);
```

Visualize the location and scale of the thirty strongest SURF features in I1 and I2. Notice that not all of the detected features can be matched because they were either not detected in both images or because some of them were not present in one of the images due to camera motion.

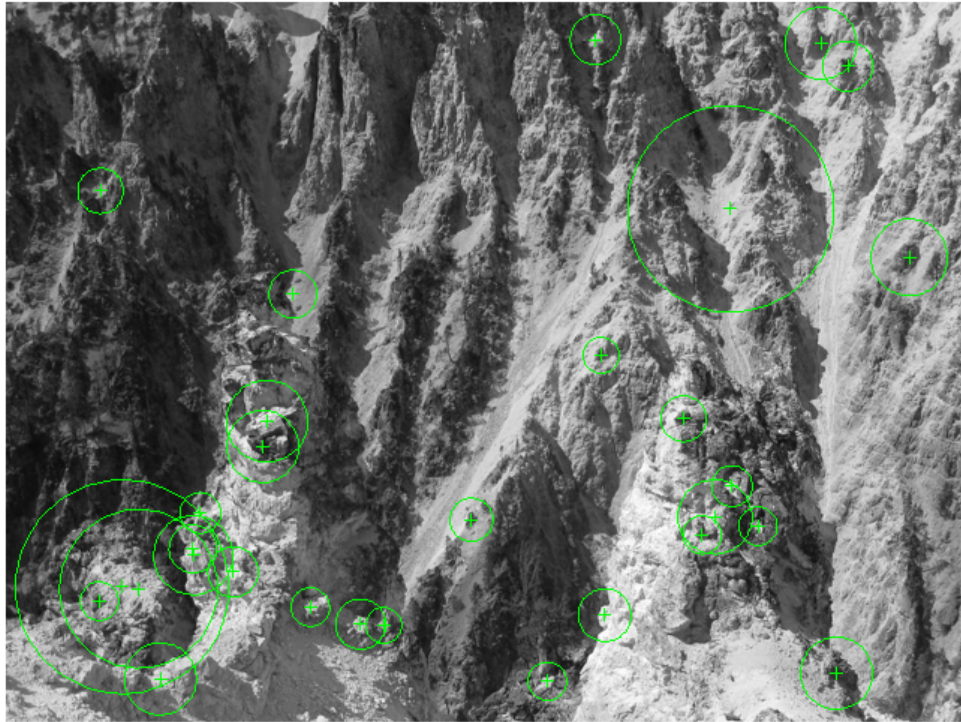
```
figure; imshow(I1); hold on;  
plot(blobs1.selectStrongest(30));  
title('Thirty strongest SURF features in I1');
```

```
figure; imshow(I2); hold on;  
plot(blobs2.selectStrongest(30));  
title('Thirty strongest SURF features in I2');
```

Thirty strongest SURF features in I1



Thirty strongest SURF features in I2



Step 3: Find Putative Point Correspondences

Use the `extractFeatures` and `matchFeatures` functions to find putative point correspondences. For each blob, compute the SURF feature vectors (descriptors).

```
[features1, validBlobs1] = extractFeatures(I1, blobs1);  
[features2, validBlobs2] = extractFeatures(I2, blobs2);
```

Use the sum of absolute differences (SAD) metric to determine indices of matching features.

```
indexPairs = matchFeatures(features1, features2, 'Metric', 'SAD', ...
```

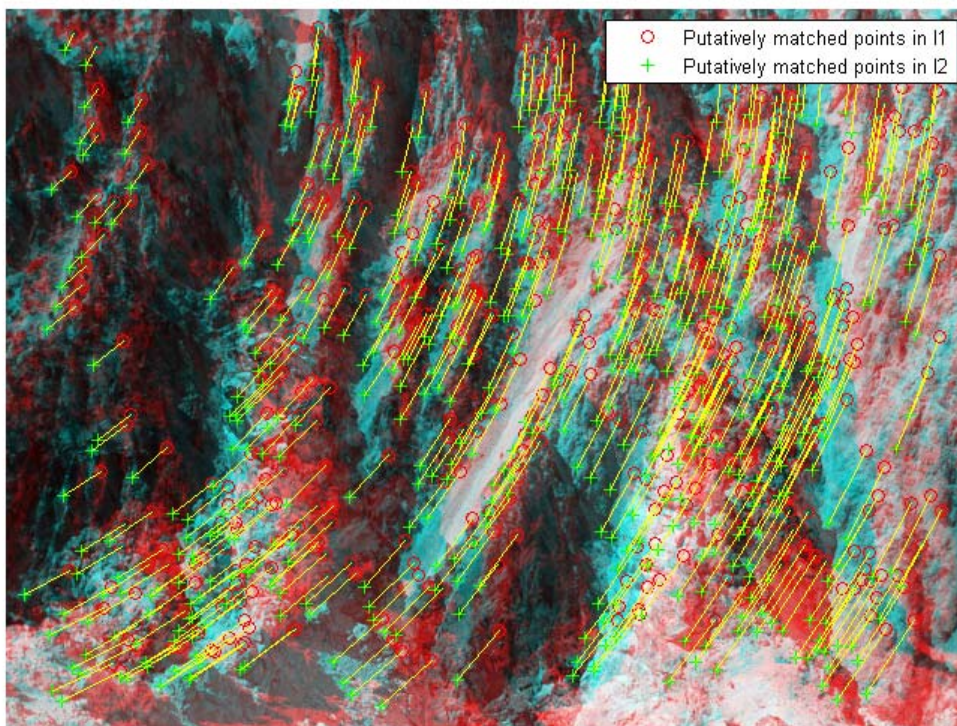
```
'MatchThreshold', 5);
```

Retrieve locations of matched points for each image

```
matchedPoints1 = validBlobs1.Location(indexPairs(:,1),:);  
matchedPoints2 = validBlobs2.Location(indexPairs(:,2),:);
```

Show matching points on top of the composite image, which combines stereo images. Notice that most of the matches are correct, but there are still some outliers.

```
figure; showMatchedFeatures(I1, I2, matchedPoints1, matchedPoints2);  
legend('Putatively matched points in I1', 'Putatively matched points in I2')
```



Step 4: Remove Outliers Using Epipolar Constraint

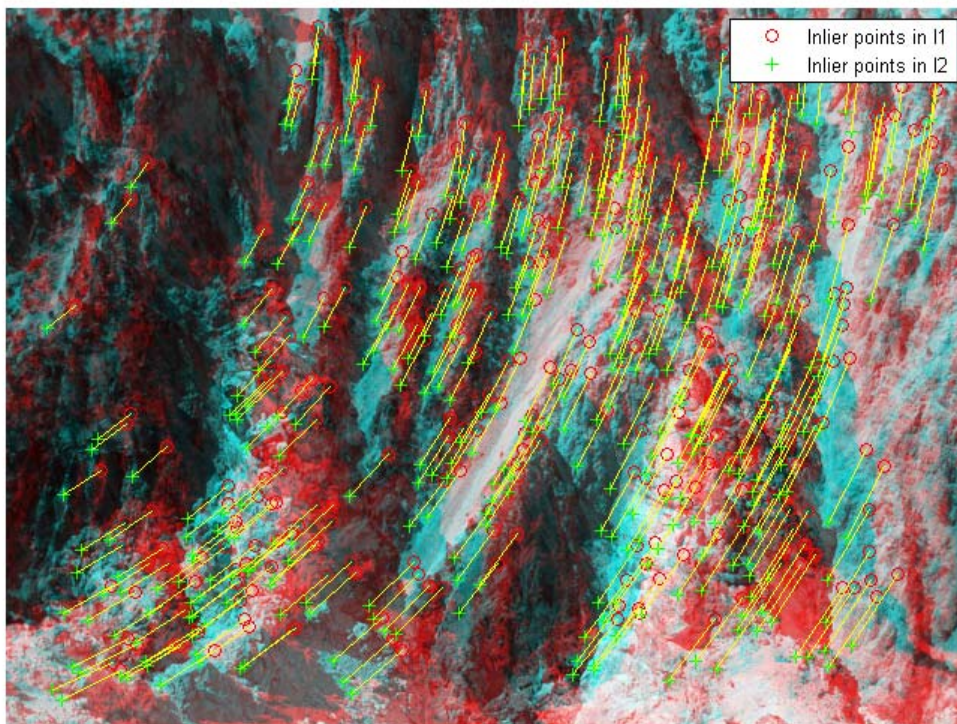
The correctly matched points must satisfy epipolar constraints. This means that a point must lie on the epipolar line determined by its corresponding point. You will use the `estimateFundamentalMatrix` function to compute the fundamental matrix and find the inliers that meet the epipolar constraint.

```
[fMatrix, epipolarInliers, status] = estimateFundamentalMatrix(...
    matchedPoints1, matchedPoints2, 'Method', 'RANSAC', ...
    'NumTrials', 10000, 'DistanceThreshold', 0.1, 'Confidence', 99.99);

if status ~= 0 || isEpipoleInImage(fMatrix, size(I1)) ...
    || isEpipoleInImage(fMatrix, size(I2))
    error(['For the rectification to succeed, the images must have enough '..
        'corresponding points and the epipoles must be outside the images.']);
end

inlierPoints1 = matchedPoints1(epipolarInliers, :);
inlierPoints2 = matchedPoints2(epipolarInliers, :);

figure; showMatchedFeatures(I1, I2, inlierPoints1, inlierPoints2);
legend('Inlier points in I1', 'Inlier points in I2');
```



Step 5: Rectify Images

Use the `estimateUncalibratedRectification` function to compute the rectification transformations. These can be used to transform the images, such that the corresponding points will appear on the same rows.

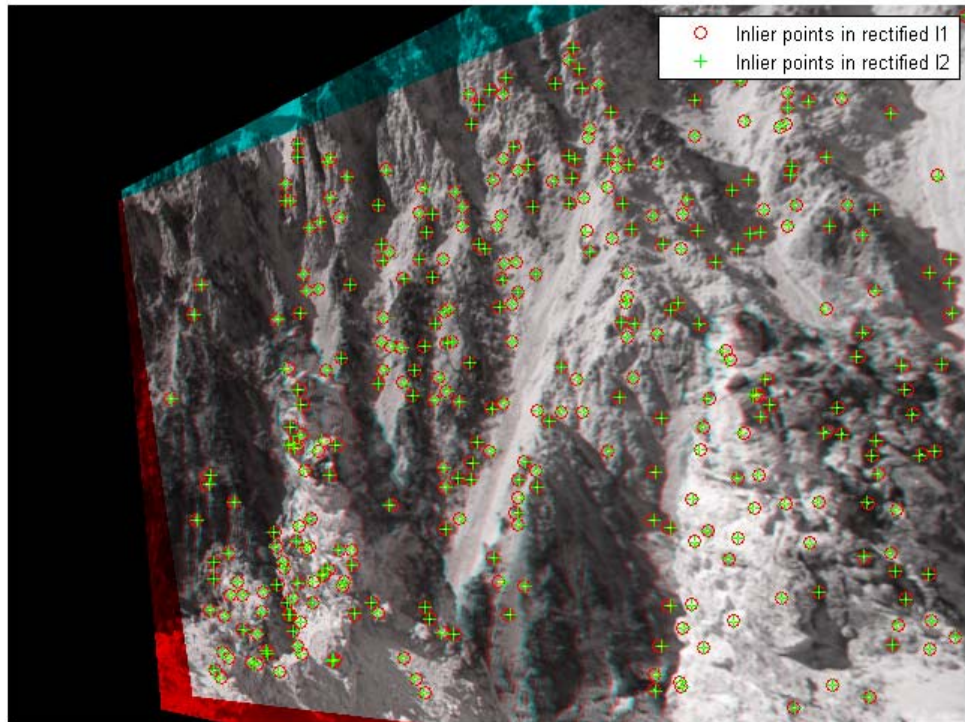
```
[t1, t2] = estimateUncalibratedRectification(fMatrix, ...  
    inlierPoints1, inlierPoints2, size(I2));
```

Rectify the images using projective transformations, `t1` and `t2`. Show a color composite of the rectified images demonstrating point correspondences.

```
geoTransformer = vision.GeometricTransformer('TransformMatrixSource', 'Input');
```



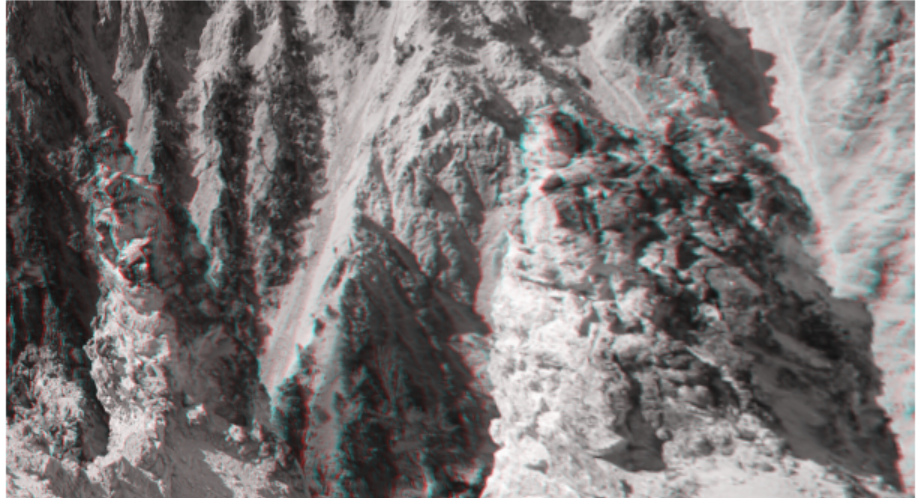
```
I1Rect = step(geoTransformer, I1, t1);  
I2Rect = step(geoTransformer, I2, t2);  
  
% transform the points to visualize them together with the rectified images  
pts1Rect = tformfwd(double(inlierPoints1), maketform('projective', double(t1)));  
pts2Rect = tformfwd(double(inlierPoints2), maketform('projective', double(t2)));  
  
figure; showMatchedFeatures(I1Rect, I2Rect, pts1Rect, pts2Rect);  
legend('Inlier points in rectified I1', 'Inlier points in rectified I2');
```



Crop the overlapping area of the rectified images. You can use red-cyan stereo glasses to see the 3D effect.

```
Irectified = cvexTransformImagePair(I1, t1, I2, t2);  
figure, imshow(Irectified);  
title('Rectified Stereo Images (Red - Left Image, Cyan - Right Image)');
```

Rectified Stereo Images (Red - Left Image, Cyan - Right Image)



Step 6: Generalize The Rectification Process

The parameters used in the above steps have been set to fit the two particular stereo images. To process other images, you can use the `cvexRectifyStereoImages` function, which contains additional logic to automatically adjust the rectification parameters. The image below shows the result of processing a pair of images using this function.

```
cvexRectifyImages('parkinglot_left.png', 'parkinglot_right.png');
```

Rectified Stereo Images (Red - Left Image, Cyan - Right Image)



References

- [1] Trucco, E; Verri, A. "Introductory Techniques for 3-D Computer Vision." Prentice Hall, 1998.
- [2] Hartley, R; Zisserman, A. "Multiple View Geometry in Computer Vision." Cambridge University Press, 2003.
- [3] Hartley, R. "In Defense of the Eight-Point Algorithm." IEEE Transactions on Pattern Analysis and Machine Intelligence, v.19 n.6, June 1997.
- [4] Fischler, MA; Bolles, RC. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography." Comm. Of the ACM 24, June 1981.

Stereo Vision

This example shows how to compute the depth map between two rectified stereo images. See the Image Rectification Example to learn about the details behind rectification. In this example we use block matching, which is the standard algorithm for high-speed stereo vision in hardware systems [8]. We first explore basic block matching, and then apply dynamic programming to improve accuracy, and image pyramiding to improve speed.

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.

Step 1. Read Stereo Image Pair

Here we read in the color stereo image pair and convert the images to gray scale for the matching process. Using color images may provide some improvement in accuracy, but it is more efficient to work with only one-channel images. For this we use the `ImageDataTypeConverter` and the `ColorSpaceConverter` System objects. Below we show the left camera image and a color composite of both images so that one can easily see the disparity between them.

```
hIdtc = vision.ImageDataTypeConverter;
hCsc = vision.ColorSpaceConverter('Conversion','RGB to intensity');
leftI3chan = step(hIdtc,imread('vipstereo_hallwayLeft.png'));
leftI = step(hCsc,leftI3chan);
rightI3chan = step(hIdtc,imread('vipstereo_hallwayRight.png'));
rightI = step(hCsc,rightI3chan);

figure(1), clf;
imshow(rightI3chan), title('Right image');

figure(2), clf;
imshowpair(rightI,leftI,'ColorChannels','red-cyan'), axis image;
title('Color composite (right=red, left=cyan)');
```


Right image



Color composite (right=red, left=cyan)



Step 2. Basic Block Matching

Next we perform basic block matching. For every pixel in the right image, we extract the 7-by-7-pixel block around it and search along the same row in the left image for the block that best matches it. Here we search in a range of ± 15 pixels around the pixel's location in the first image, and we use the sum of absolute differences (SAD) to compare the image regions. We need only search over columns and not over rows because the images are rectified. We use the `TemplateMatcher System` object to perform this block matching between each block and the region of interest.

```
Dbasic = zeros(size(leftI), 'single');
disparityRange = 15;
% Selects (2*halfBlockSize+1)-by-(2*halfBlockSize+1) block.
halfBlockSize = 3;
blockSize = 2*halfBlockSize+1;
% Allocate space for all template matcher System objects.
tmats = cell(blockSize);

% Initialize progress bar
hWaitBar = waitbar(0, 'Performing basic block matching...');
nRowsLeft = size(leftI, 1);

% Scan over all rows.
for m=1:nRowsLeft
    % Set min/max row bounds for image block.
    minr = max(1,m-halfBlockSize);
    maxr = min(nRowsLeft,m+halfBlockSize);
    % Scan over all columns.
    for n=1:size(leftI,2)
        minc = max(1,n-halfBlockSize);
        maxc = min(size(leftI,2),n+halfBlockSize);
        % Compute disparity bounds.
        mind = max( -disparityRange, 1-minc );
        maxd = min( disparityRange, size(leftI,2)-maxc );

        % Construct template and region of interest.
        template = rightI(minr:maxr,minc:maxc);
        templateCenter = floor((size(template)+1)/2);
        roi = [minc+templateCenter(2)+mind-1 ...
              minr+templateCenter(1)-1 ...
              maxd-mind+1 1];
```

```

    % Lookup proper TemplateMatcher object; create if empty.
    if isempty(tmats{size(template,1),size(template,2)})
        tmats{size(template,1),size(template,2)} = ...
            vision.TemplateMatcher('ROIInputPort',true);
    end
    thisTemplateMatcher = tmats{size(template,1),size(template,2)};

    % Run TemplateMatcher object.
    loc = step(thisTemplateMatcher, leftI, template, roi);
    Dbasic(m,n) = loc(1) - roi(1) + mind;
end
waitbar(m/nRowsLeft,hWaitBar);
end

close(hWaitBar);

```

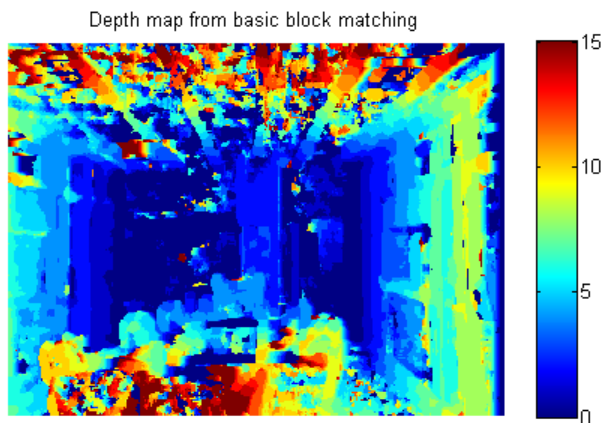
In the results below, the basic block matching does well, as the correct shape of the stereo scene is recovered. However, there are noisy patches and bad depth estimates everywhere, especially on the ceiling. These are caused when no strong image features appear inside of the 7-by-7-pixel windows being compared. Then the matching process is subject to noise since each pixel chooses its disparity independently of all the other pixels.

For display purposes, we saturate the depth map to have only positive values. In general, slight angular misalignment of the stereo cameras used for image acquisition can allow both positive and negative disparities to appear validly in the depth map. In this case, however, the stereo cameras were near perfectly parallel, so the true disparities have only one sign. Thus this correction is valid.

```

figure(3), clf;
imshow(Dbasic,[]), axis image, colormap('jet'), colorbar;
caxis([0 disparityRange]);
title('Depth map from basic block matching');

```



Step 3. Sub-pixel Estimation

The disparity estimates returned by block matching are all integer-valued, so the above depth map exhibits contouring effects where there are no smooth transitions between regions of different disparity. This can be ameliorated by incorporating sub-pixel computation into the matching metric. Previously we only took the location of the minimum cost as the disparity, but now we take into consideration the minimum cost and the two neighboring cost values. We fit a parabola to these three values, and analytically solve for the minimum to get the sub-pixel correction.

```

DbasicSubpixel= zeros(size(leftI), 'single');
tmats = cell(2*halfBlockSize+1);
hWaitBar=waitbar(0,'Performing sub-pixel estimation...');
for m=1:nRowsLeft
    % Set min/max row bounds for image block.
    minr = max(1,m-halfBlockSize);
    maxr = min(nRowsLeft,m+halfBlockSize);
    % Scan over all columns.
    for n=1:size(leftI,2)
        minc = max(1,n-halfBlockSize);

```

```

maxc = min(size(leftI,2),n+halfBlockSize);
% Compute disparity bounds.
mind = max( -disparityRange, 1-minc );
maxd = min( disparityRange, size(leftI,2)-maxc );

% Construct template and region of interest.
template = rightI(minr:maxr,minc:maxc);
templateCenter = floor((size(template)+1)/2);
roi = [minc+templateCenter(2)+mind-1 ...
       minr+templateCenter(1)-1 ...
       maxd-mind+1 1];
% Lookup proper TemplateMatcher object; create if empty.
if isempty(tmats{size(template,1),size(template,2)})
    tmats{size(template,1),size(template,2)} = ...
        vision.TemplateMatcher('ROIInputPort',true,...
                                'BestMatchNeighborhoodOutputPort',true);
end
thisTemplateMatcher = tmats{size(template,1),size(template,2)};

% Run TemplateMatcher object.
[loc,a2] = step(thisTemplateMatcher, leftI, template, roi);
ix = single(loc(1) - roi(1) + mind);

% Subpixel refinement of index.
DbasicSubpixel(m,n) = ix - 0.5 * (a2(2,3) - a2(2,1)) ...
    / (a2(2,1) - 2*a2(2,2) + a2(2,3));
end
waitbar(m/nRowsLeft,hWaitBar);
end

close(hWaitBar);

```

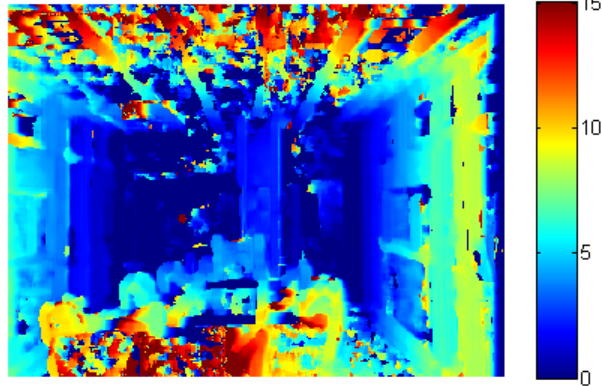
Re-running basic block matching, we achieve the result below where the contouring effects are mostly removed and the disparity estimates are correctly refined. This is especially evident along the walls.

```

figure(1), clf;
imshow(DbasicSubpixel,[], axis image, colormap('jet'), colorbar;
caxis([0 disparityRange]);
title('Basic block matching with sub-pixel accuracy');

```

Basic block matching with sub-pixel accuracy



Step 4. Dynamic Programming

As mentioned above, basic block matching creates a noisy disparity image. This can be improved by introducing a smoothness constraint. Basic block matching chooses the optimal disparity for each pixel based on its own cost function alone. Now we want to allow a pixel to have a disparity with possibly sub-optimal cost for it locally. This extra cost must be offset by increasing that pixel's agreement in disparity with its neighbors. In particular, we constrain each disparity estimate to lie with ± 3 values of its neighbors' disparities, where its neighbors are the adjacent pixels along an image row. The problem of finding the optimal disparity estimates for a row of pixels now becomes one of finding the "optimal path" from one side of the image to the other. To find this optimal path, we use the underlying block matching metric as the cost function and constrain the disparities to only change by a certain amount between adjacent pixels. This is a problem that can be solved efficiently using the technique of dynamic programming [3,4].

```
Ddynamic = zeros(size(leftI), 'single');  
finf = 1e3; % False infinity  
disparityCost = finf*ones(size(leftI,2), 2*disparityRange + 1, 'single');
```

```

disparityPenalty = 0.5; % Penalty for disparity disagreement between pixels
hWaitBar = waitbar(0,'Using dynamic programming for smoothing...');
% Scan over all rows.
for m=1:nRowsLeft
    disparityCost(:) = finf;
    % Set min/max row bounds for image block.
    minr = max(1,m-halfBlockSize);
    maxr = min(nRowsLeft,m+halfBlockSize);
    % Scan over all columns.
    for n=1:size(leftI,2)
        minc = max(1,n-halfBlockSize);
        maxc = min(size(leftI,2),n+halfBlockSize);
        % Compute disparity bounds.
        mind = max( -disparityRange, 1-minc );
        maxd = min( disparityRange, size(leftI,2)-maxc );
        % Compute and save all matching costs.
        for d=mind:maxd
            disparityCost(n, d + disparityRange + 1) = ...
                sum(sum(abs(leftI(minr:maxr,(minc:maxc)+d) ...
                    - rightI(minr:maxr,minc:maxc))));
        end
    end
end

% Process scan line disparity costs with dynamic programming.
optimalIndices = zeros(size(disparityCost), 'single');
cp = disparityCost(end,:);
for j=size(disparityCost,1)-1:-1:1
    % False infinity for this level
    cfinf = (size(disparityCost,1) - j + 1)*finf;
    % Construct matrix for finding optimal move for each column
    % individually.
    [v,ix] = min([cfinf cfinf cp(1:end-4)+3*disparityPenalty;
        cfinf cp(1:end-3)+2*disparityPenalty;
        cp(1:end-2)+disparityPenalty;
        cp(2:end-1);
        cp(3:end)+disparityPenalty;
        cp(4:end)+2*disparityPenalty cfinf;
        cp(5:end)+3*disparityPenalty cfinf cfinf],[,],1);
    cp = [cfinf disparityCost(j,2:end-1)+v cfinf];
    % Record optimal routes.
end

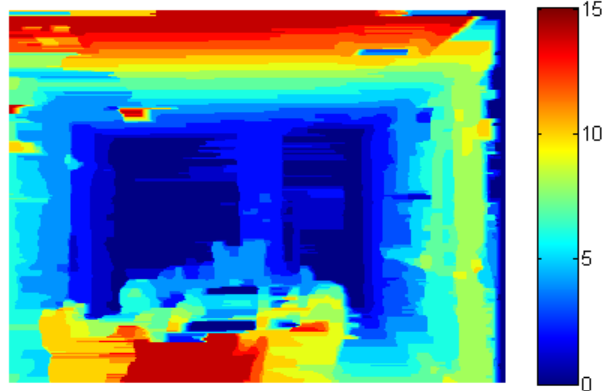
```

```
        optimalIndices(j,2:end-1) = (2:size(disparityCost,2)-1) + (ix - 4);
    end
    % Recover optimal route.
    [~,ix] = min(cp);
    Ddynamic(m,1) = ix;
    for k=1:size(Ddynamic,2)-1
        Ddynamic(m,k+1) = optimalIndices(k, ...
            max(1, min(size(optimalIndices,2), round(Ddynamic(m,k)) ) ) );
    end
    waitbar(m/nRowsLeft, hWaitBar);
end
close(hWaitBar);
Ddynamic = Ddynamic - disparityRange - 1;
```

The image below shows the stereo result refined by applying dynamic programming to each row individually. Dynamic programming does introduce errors of its own by blurring the edges around object boundaries due to the smoothness constraint. Also, it does nothing to smooth "between" rows, which is why a striation pattern now appears on the left side foreground chair. Despite these limitations, the result is significantly improved, with the noise along the walls and ceiling nearly completely removed, and with many of the foreground objects being better reconstructed.

```
figure(3), clf;
imshow(Ddynamic,[]), axis image, colormap('jet'), colorbar;
caxis([0 disparityRange]);
title('Block matching with dynamic programming');
```


Block matching with dynamic programming



Step 5. Image Pyramiding

While dynamic programming can improve the accuracy of the stereo image, basic block matching is still an expensive operation, and dynamic programming only adds to the burden. One solution is to use image pyramiding and telescopic search to guide the block matching [5,7]. With the full-size image, we had to search over a ± 15 -pixel range to properly detect the disparities in the image. If we had down-sized the image by a factor of two, however, this search could have been reduced to ± 7 pixels on an image a quarter of the area, meaning this step would cost a factor of 8 less. Then we use the disparity estimates from this down-sized operation to seed the search on the larger image, and therefore we only need to search over a smaller range of disparities.

The below example performs this telescoping stereo matching using a four-level image pyramid. We use the `Pyramid` and `GeometricScaler` System objects, and we have wrapped up the preceding block matching code into the function `vipstereo_blockmatch.m` for simplicity. The disparity search range is only ± 3 pixels at each level, making it over 5x faster to compute than basic block matching. Yet the results compare favorably.

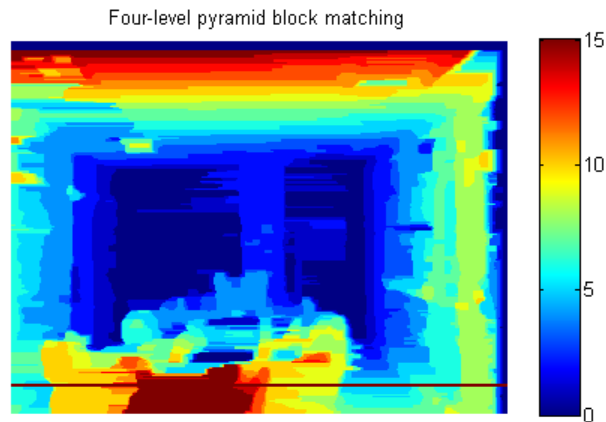
```

% Construct a four-level pyramid
pyramids = cell(1,4);
pyramids{1}.L = leftI;
pyramids{1}.R = rightI;
for i=2:length(pyramids)
    hPyr = vision.Pyramid('PyramidLevel',1);
    pyramids{i}.L = single(step(hPyr,pyramids{i-1}.L));
    pyramids{i}.R = single(step(hPyr,pyramids{i-1}.R));
end
% Declare original search radius as +/-4 disparities for every pixel.
smallRange = single(3);
disparityMin = repmat(-smallRange, size(pyramids{end}.L));
disparityMax = repmat( smallRange, size(pyramids{end}.L));
% Do telescoping search over pyramid levels.
for i=length(pyramids):-1:1
    Dpyramid = vipstereo_blockmatch(pyramids{i}.L,pyramids{i}.R, ...
        disparityMin,disparityMax,...
        false,true,3,...
        true,... % Waitbar
        ['Performing level-',num2str(length(pyramids)-i+1),...
        ' pyramid block matching...']); %Waitbar title

    if i > 1
        % Scale disparity values for next level.
        hGzca = vision.GeometricScaler(...
            'InterpolationMethod','Nearest neighbor',...
            'SizeMethod','Number of output rows and columns',...
            'Size',size(pyramids{i-1}.L));
        Dpyramid = 2*step(hGzca, Dpyramid);
        % Maintain search radius of +/-smallRange.
        disparityMin = Dpyramid - smallRange;
        disparityMax = Dpyramid + smallRange;
    end
end

figure(3), clf;
imshow(Dpyramid,[]), colormap('jet'), colorbar, axis image;
caxis([0 disparityRange]);
title('Four-level pyramid block matching');

```



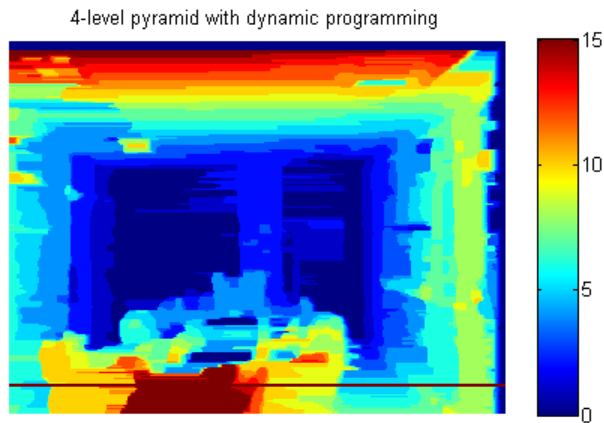
Step 6. Combined Pyramid and Dynamic Programming

Finally we merge the above techniques and run dynamic programming along with image pyramiding, where the dynamic programming is run on the disparity estimates output by every pyramid level. The results compare well with the highest-quality results we have obtained so far, and are still achieved at a reduced computational burden versus basic block matching.

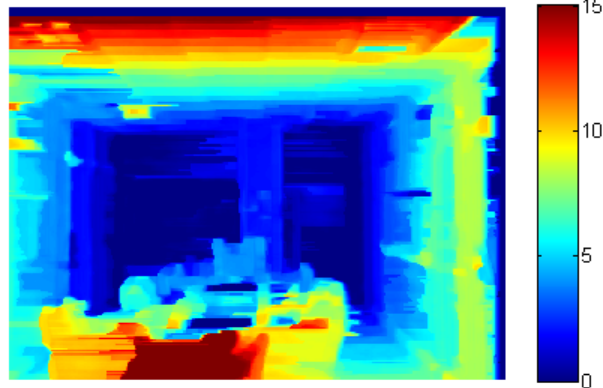
It is also possible to use sub-pixel methods with dynamic programming, and we show the results of all three techniques in the second image. As before, subpixel refinement reduces contouring effects and clearly improves accuracy. The previous code has been bundled into `vipstereo_blockmatch_combined.m`, which exposes all of the options previously presented as parameter-value pairs.

```
DpyramidDynamic = vipstereo_blockmatch_combined(leftI,rightI, ...
    'NumPyramids',3, 'DisparityRange',4, 'DynamicProgramming',true, ...
    'Waitbar', true, ...
    'WaitbarTitle', 'Performing combined pyramid and dynamic programming');
figure(3), clf;
imshow(DpyramidDynamic,[],) axis('image'), colorbar, colormap jet;
```

```
caxis([0 disparityRange]);  
title('4-level pyramid with dynamic programming');  
  
DdynamicSubpixel = vipstereo_blockmatch_combined(leftI,rightI, ...  
    'NumPyramids',3, 'DisparityRange',4, 'DynamicProgramming',true, ...  
    'Subpixel', true, ...  
    'Waitbar', true, ...  
    'WaitbarTitle', ['Performing combined pyramid and dynamic ',...  
    'programming with sub-pixel estimation']);  
  
figure(4), clf;  
imshow(DdynamicSubpixel,[]), axis image, colormap('jet'), colorbar;  
caxis([0 disparityRange]);  
title('Pyramid with dynamic programming and sub-pixel accuracy');
```



Pyramid with dynamic programming and sub-pixel accuracy



Step 7. Backprojection

With a stereo depth map and knowledge of the intrinsic parameters of the camera, it is possible to backproject image pixels into 3D points [1,2]. One way to compute the camera intrinsics is with the MATLAB Camera Calibration Toolbox [6] from the California Institute of Technology. Such a tool will produce an intrinsics matrix, K , of the form:

$$K = \begin{bmatrix} \text{focal_length_x} & \text{skew_x} & \text{camera_center_x}; \\ 0 & \text{focal_length_y} & \text{camera_center_y}; \\ 0 & 0 & 1 \end{bmatrix};$$

This relates 3D world coordinates to homogenized camera coordinates via:

$$[x_{\text{camera}} \ y_{\text{camera}} \ 1]^T = K \cdot [x_{\text{world}} \ y_{\text{world}} \ z_{\text{world}}]^T$$

With the intrinsics matrix, we can backproject each image pixel into a 3D ray that describes all the world points that could have been projected onto that pixel on the image. This leaves unknown the distance of that point to the camera. This is provided by the disparity measurements of the stereo depth map as:

$$z_{world} = focal_length \cdot \frac{1 + stereo_baseline}{disparity}$$

Note that unitless pixel disparities cannot be used directly in this equation. Also, if the stereo baseline (the distance between the two cameras) is not well-known, it introduces more unknowns. Thus we transform this equation into the general form:

$$z_{world} = a + \frac{b}{disparity}$$

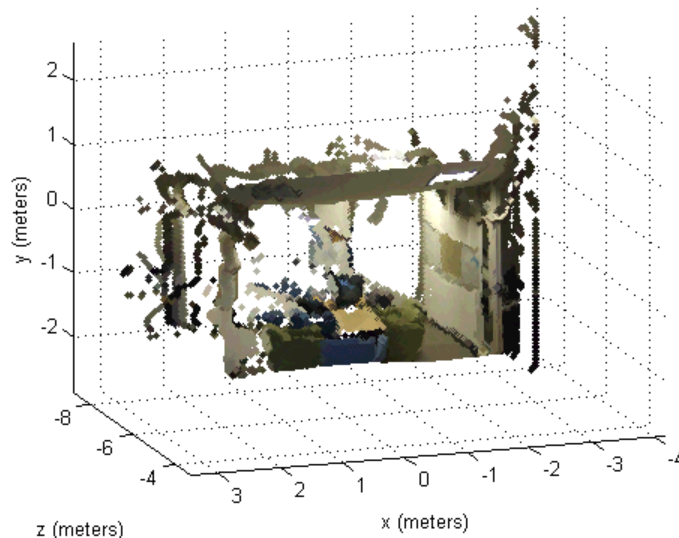
We solve for the two unknowns via least squares by collecting a few corresponding depth and disparity values from the scene and using them as tie points. The full technique is shown below.

```
% Camera intrinsics matrix
K = [409.4433      0 204.1225
      0 416.0865 146.4133
      0      0  1.0000];
% Create a sub-sampled grid for backprojection.
dec = 2;
[X,Y] = meshgrid(1:dec:size(leftI,2),1:dec:size(leftI,1));
P = K\[X(:)'; Y(:)'; ones(1,numel(X), 'single')];
Disp = max(0,DdynamicSubpixel(1:dec:size(leftI,1),1:dec:size(leftI,2)));
hMedF = vision.MedianFilter('NeighborhoodSize',[5 5]);
Disp = step(hMedF,Disp); % Median filter to smooth out noise.
% Derive conversion from disparity to depth with tie points:
knownDs = [15  9  2]'; % Disparity values in pixels
knownZs = [4  4.5 6.8]';
% World z values in meters based on scene measurements.
ab = [1./knownDs ones(size(knownDs), 'single')] \ knownZs; % least squares
% Convert disparity to z (distance from camera)
ZZ = ab(1)./Disp(:)' + ab(2);
% Threshold to [0,8] meters.
ZZdisp = min(8,max(0, ZZ ));
Pd = bsxfun(@times,P,ZZ);
% Remove near points
bad = Pd(3,:) > 8 | Pd(3,:) < 3;
Pd = Pd(:,~bad);
```

In the reprojection, the walls, ceiling, and floor all appear mutually orthogonal, and the scene is well reconstructed. Since camera calibration also gives intrinsics with units, we can assign units to the backprojected points. The dimensions of the plot are given in meters, and one can verify that the sizes of objects and the scene appear correct.

```
% Collect quantized colors for point display
Colors = rightI3chan(1:dec:size(rightI,1),1:dec:size(rightI,2),:);
Colors = reshape(Colors,[size(Colors,1)*size(Colors,2) size(Colors,3)]);
Colors = Colors(~bad,:);
cfac = 20;
C8 = round(cfac*Colors);
[U,I,J] = unique(C8,'rows');
C8 = C8/cfac;

figure(2), clf, hold on, axis equal;
for i=1:size(U,1)
    plot3(-Pd(1,J==i),-Pd(3,J==i),-Pd(2,J==i),'.','Color',C8(I(i),:));
end
view(161,14), grid on;
xlabel('x (meters)'), ylabel('z (meters)'), zlabel('y (meters)');
```

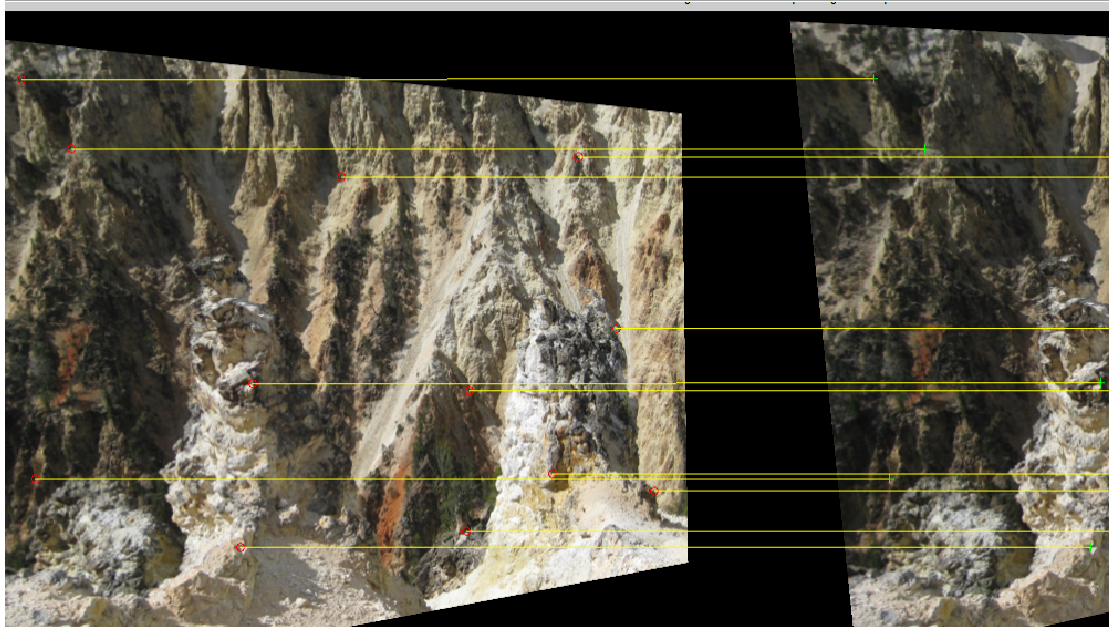


References

- [1] Trucco, E; Verri, A. "Introductory Techniques for 3-D Computer Vision." Prentice Hall, 1998.
- [2] Hartley, R; Zisserman, A. "Multiple View Geometry in Computer Vision." Cambridge University Press, 2003.
- [3] Veksler, O. "Stereo Correspondence by Dynamic Programming on a Tree." University of Western Ontario.
- [4] Park, CS; Park, HW. "A robust stereo disparity estimation using adaptive window search and dynamic programming search." Pattern Recognition, 2000.
- [5] Thevenaz, P; Ruttimann, UE; Unser, M. "A Pyramid Approach to Subpixel Registration Based on Intensity." IEEE Transactions on Image Processing (1998) Vol. 7, No. 1.
- [6] Bouguet, JY. "Camera Calibration Toolbox for Matlab." Computational Vision at the California Institute of Technology. http://www.vision.caltech.edu/bouguetj/calib_doc/
- [7] Koschan, A; Rodehorst, V; Spiller, K. "Color Stereo Vision Using Hierarchical Block Matching and Active Color Illumination." Pattern Recognition, 1996.
- [8] Ambrosch, K; Kubinger, W; Humenberger, M; Steininger, A. "Flexible Hardware-Based Stereo Matching." EURASIP Journal on Embedded Systems, 2008.

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 feature points which are already matched.
I1 = im2double(imread('yellowstone_left.png'));
I2 = im2double(imread('yellowstone_right.png'));
load yellowstone_matched_points;

% Display point correspondences. Notice that the matching points are in
```

```
% different rows, indicating that the stereo pair is not rectified.
showMatchedFeatures(I1, I2, matched_points1, matched_points2, 'montage')
title('Original images and matching feature points');

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

% Compute the rectification transformations.
[t1, t2] = estimateUncalibratedRectification(f, matched_points1, ...
    matched_points2, size(I2));

% Rectify the stereo images using projective transformations t1 and t2.
geoTransf = vision.GeometricTransformer('TransformMatrixSource', ...
    'Input port', 'OutputImagePositionSource', 'Property');
offset = [120 30]; % shift the coordinate system origin by an offset
geoTransf.OutputImagePosition = [-offset size(I1,2)+130 ...
    size(I1,1)+150]; % [x y width height]
I1Rect = step(geoTransf, I1, t1);
I2Rect = step(geoTransf, I2, t2);

% Transform the points to visualize them together with the rectified images.
pts1Rect = tformfwd(matched_points1, maketform('projective', t1));
pts2Rect = tformfwd(matched_points2, maketform('projective', t2));

% Compensate for the shift in the coordinate system origin.
pts1Rect = bsxfun(@plus, pts1Rect, offset);
pts2Rect = bsxfun(@plus, pts2Rect, offset);

% Notice that the matching points now reside on the same rows. This also
% means that the epipolar lines are parallel to the x-axis.
figure; showMatchedFeatures(I1Rect, I2Rect, pts1Rect, pts2Rect, 'montage')
title('Rectified images and the corresponding feature points');
```

Motion Estimation and Tracking

- “Tracking Cars Using Gaussian Mixture Models” on page 4-2
- “Video Mosaicking” on page 4-7
- “Video Mosaicking” on page 4-13
- “Track an Object Using Correlation” on page 4-19
- “Panorama Creation” on page 4-25

Tracking Cars Using Gaussian Mixture Models

This example shows how to detect cars in a video sequence using foreground detection based on gaussian mixture models (GMMs).

Introduction

This example illustrates the use of gaussian mixture models to detect foreground in a video. After foreground detection, the example process the binary foreground images using blob analysis. Finally, bounding boxes are drawn around the detected cars.

Initialization

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

Create a System object to read video from avi file.

```
filename = 'viptraffic.avi';  
hvfr = vision.VideoFileReader(filename, 'ImageColorSpace', 'RGB');
```

Create color space converter System object to convert the image from RGB to intensity format.

```
hcsc = vision.ColorSpaceConverter('Conversion', 'RGB to intensity');
```

Create a System object to detect foreground using gaussian mixture models.

```
hfdet = vision.ForegroundDetector(...  
    'NumTrainingFrames', 5, ...    % only 5 because of short video  
    'InitialVariance', (30/255)^2); % initial standard deviation of 30/
```

Create a blob analysis System object to segment cars in the video.

```
hblob = vision.BlobAnalysis( ...  
    'CentroidOutputPort', false, ...  
    'AreaOutputPort', true, ...  
    'BoundingBoxOutputPort', true, ...  
    'ExtentOutputPort', true, ...  
    'OutputDataType', 'single', ...
```

```

'MinimumBlobArea', 250, ...
'MaximumBlobArea', 3600, ...
'MaximumCount', 80);

```

Create System object for drawing the bounding boxes around detected cars.

```

hshapeins = vision.ShapeInserter( ...
    'BorderColor', 'Custom', ...
    'CustomBorderColor', [0 255 0]);

```

Create and configure a System object to write the number of cars being tracked.

```

htextins = vision.TextInserter( ...
    'Text', '%4d', ...
    'Location', [1 1], ...
    'Color', [255 255 255], ...
    'FontSize', 12);

```

Create System objects to display the results.

```

sz = get(0, 'ScreenSize');
pos = [20 sz(4)-300 200 200];
hVideoOrig = vision.VideoPlayer('Name', 'Original', 'Position', pos);
pos(1) = pos(1)+220; % move the next viewer to the right
hVideoFg = vision.VideoPlayer('Name', 'Foreground', 'Position', pos);
pos(1) = pos(1)+220;
hVideoRes = vision.VideoPlayer('Name', 'Results', 'Position', pos);

line_row = 23; % Define region of interest (ROI)

```

Stream Processing Loop

Create a processing loop to track the cars in the input video. This loop uses the previously instantiated System objects.

When the VideoFileReader object detects the end of the input file, the loop stops.

```

while ~isDone(hvfr)
    image = step(hvfr); % Read input video frame

```

```
y = step(hcsc, image); % Convert color image to intensity

% Remove the effect of sudden intensity changes due to camera's
% auto white balancing algorithm.
y = y-mean(y(:));

fg_image = step(hfdet, y); % Detect foreground

% Estimate the area and bounding box of the blobs in the foreground
% image.
[area, bbox, extent] = step(hblob, fg_image);

image_out = image;
image_out(22:23, :, :) = 255; % Count cars only below this white line
image_out(1:15, 1:30, :) = 0; % Black background for displaying count

Idx = bbox(:,2) > line_row; % Select boxes which are in the ROI.

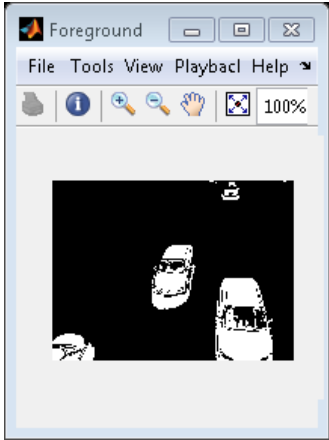
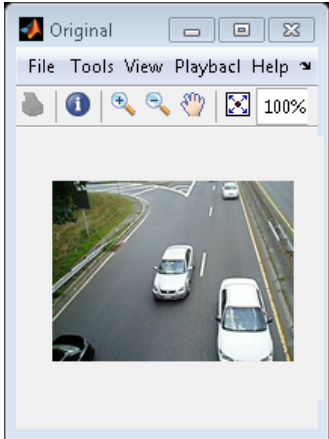
% Based on blob sizes, filter out objects which are unlikely to be
% cars. When the extent property of a blob exceeds 0.4 (40%),
% classify it as a car.
isCar = extent > 0.4;
numCars = sum(isCar); % Number of cars
bbox(~isCar, :) = []; % Keep only the bounding boxes for objects
% classified as cars.

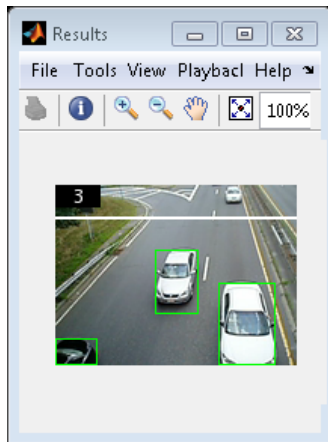
% Draw bounding rectangles around the detected cars.
image_out = step(hshapeins, image_out, bbox);

% Display the number of cars tracked and a white line showing the ROI.
image_out = step(htextins, image_out, int32(numCars));

step(hVideoOrig, image); % Original video
step(hVideoFg, fg_image); % Foreground
step(hVideoRes, image_out); % Bounding boxes around cars
end

% Close the video file
release(hvfr);
```





Summary

The output video displays the bounding boxes around the cars. It also displays the number of cars in the upper left corner of the video.

Video Mosaicking

This example shows how to create a mosaic from a video sequence. 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.

Introduction

This example illustrates how to use the CornerDetector System object, the GeometricTransformEstimator System object, the AlphaBlender System object, and the GeometricTransformer System object to create a mosaic image from a video sequence. First, the example identifies the corners in the first (reference) and second video frame. Then, it calculates the projective transformation matrix that best describes the transformation between corner positions in these frames. Finally, the example overlays the second image onto the first image. The example repeats this process to create a mosaic image of the video scene.

Initialization

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

```
featWinLen    = 9;                % Length of feature window
maxNumPoints  = int32(75);        % Maximum number of points
sizePano      = [400 680];
origPano      = [5 60];
classToUse    = 'single';
```

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

```
hsrc = vision.VideoFileReader('vipmosaicking.avi', 'ImageColorSpace', ...
    'RGB', 'PlayCount', 1);
```

Create a ColorSpaceConverter System object to convert the RGB image to intensity format.

```
hcsc = vision.ColorSpaceConverter('Conversion', 'RGB to intensity');
```

Create a `CornerDetector` System object to identify the corner locations of current and the previous video frames.

```
hcornerdet = vision.CornerDetector( ...  
    'Method', 'Local intensity comparison (Rosen & Drummond)', ...  
    'IntensityThreshold', 0.1, 'MaximumCornerCount', maxNumPoints, ...  
    'CornerThreshold', 0.001, 'NeighborhoodSize', [21 21]);
```

Create an `GeometricTransformEstimator` System object to implement the RANSAC (RANdom SAMple Consensus) algorithm. This System object verifies that the corners have been properly identified. It also calculates the transformation matrix that maps the corner positions in the current frame to the corner positions in previous frame.

```
hestgeotform = vision.GeometricTransformEstimator;
```

Create a `GeometricTransformer` System object.

```
hgeotrans = vision.GeometricTransformer( ...  
    'OutputImagePositionSource', 'Property', 'ROIInputPort', true, ...  
    'OutputImagePosition', [-origPano fliplr(sizePano)]);
```

Create a `AlphaBlender` System object to overlay the consecutive video frames to produce the panorama.

```
halphablender = vision.AlphaBlender( ...  
    'Operation', 'Binary mask', 'MaskSource', 'Input port');
```

Create a `MarkerInserter` System object to draw the corner locations in each video frame.

```
hdrawmarkers = vision.MarkerInserter('Shape', 'Circle', ...  
    'BorderColor', 'Custom', 'CustomBorderColor', [1 0 0]);
```

Create two `VideoPlayer` System objects, one to display the corners of each frame and other to draw the panorama.

```
hVideo1 = vision.VideoPlayer('Name', 'Corners');  
hVideo1.Position(1) = hVideo1.Position(1) - 350;  
  
hVideo2 = vision.VideoPlayer('Name', 'Mosaic');
```

```
hVideo2.Position(1) = hVideo1.Position(1) + 400;
hVideo2.Position([3 4]) = [750 500];
```

Stream Processing Loop

Create a processing loop to create panorama from the input video. This loop uses the System objects you instantiated above.

```
points = zeros([0 2], classToUse);
features = zeros([0 featWinLen^2], classToUse);

while ~isDone(hsrc)
    % Save the points and features computed from the previous image
    pointsPrev = points;
    featuresPrev = features;

    % To speed up mosaicking, select and process every 5th image
    for i = 1:5
        rgb = step(hsrc);
        if isDone(hsrc)
            break;
        end
    end
    I = step(hcsc, rgb);
    roi = int32([2 2 size(I, 2)-2 size(I, 1)-2]);

    % Detect corners in the image
    cornerPoints = step(hcornerdet, I);
    cornerPoints = cast(cornerPoints, classToUse);

    % Extract the features for the corners
    [features, points] = extractFeatures(I, ...
        cornerPoints, 'BlockSize', featWinLen);

    % Match features computed from the current and the previous images
    indexPairs = matchFeatures(features, featuresPrev);

    % Check if there are enough corresponding points in the current and the
    % previous images
    isMatching = false;
```

```
if size(indexPairs, 1) > 2
    matchedPoints = points(indexPairs(:, 1), :);
    matchedPointsPrev = pointsPrev(indexPairs(:, 2), :);

    % Find corresponding points in the current and the previous images,
    % and compute a geometric transformation from the corresponding
    % points
    [tform, inlier] = step(hestgeotform, matchedPoints, matchedPointsPrev);

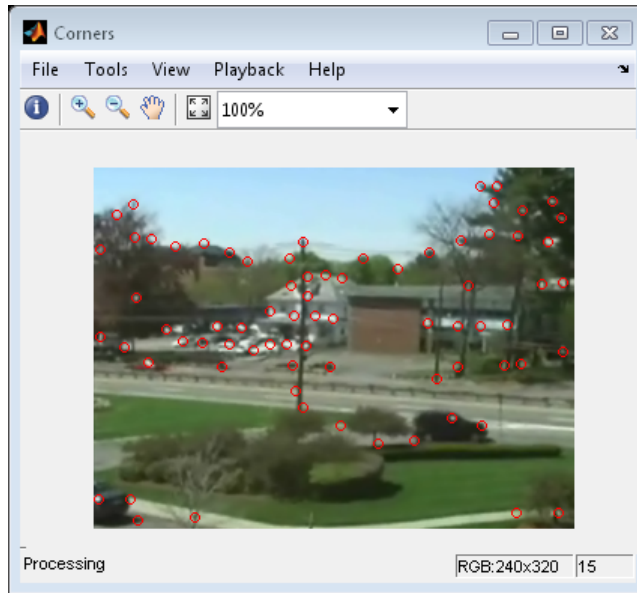
    if sum(inlier) >= 4
        % If there are at least 4 corresponding points, we declare the
        % current and the previous images matching
        isMatching = true;
    end
end

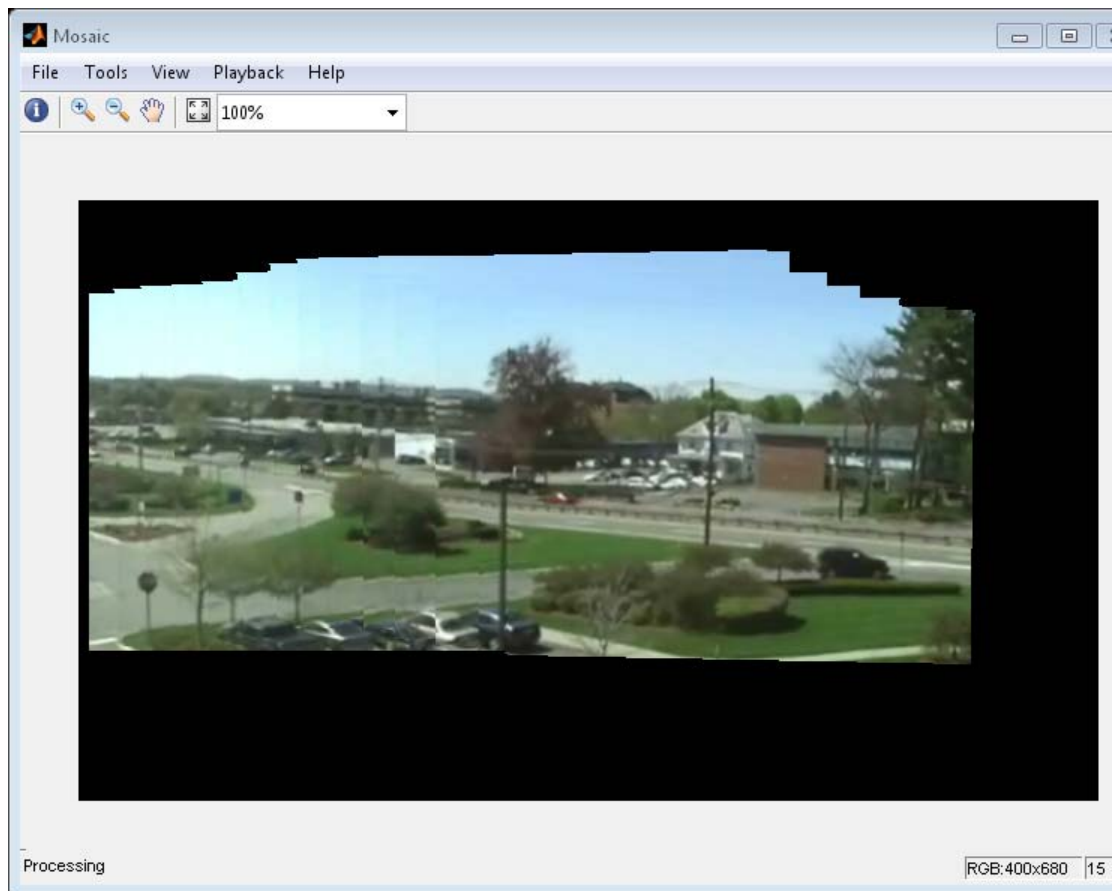
if isMatching
    % If the current image matches with the previous one, compute the
    % transformation for mapping the current image onto the mosaic
    % image
    xtform = xtform * [tform, [0 0 1]'];
else
    % If the current image does not match the previous one, reset the
    % transformation and the mosaic image
    xtform = eye(3, classToUse);
    mosaic = zeros([sizePano,3], classToUse);
end

% Display the current image and the corner points
cornerImage = step(hdrawmarkers, rgb, cornerPoints);
step(hVideo1, cornerImage);

% Warp the current image onto the mosaic image
transformedImage = step(hgeotrans, rgb, xtform, roi);
mosaic = step(halphablender, mosaic, transformedImage, ...
    transformedImage(:,:,1)>0);
step(hVideo2, mosaic);
end

release(hsrc);
```





The Corners window displays the input video along with the detected corners and Mosaic window displays the panorama created from the input video.

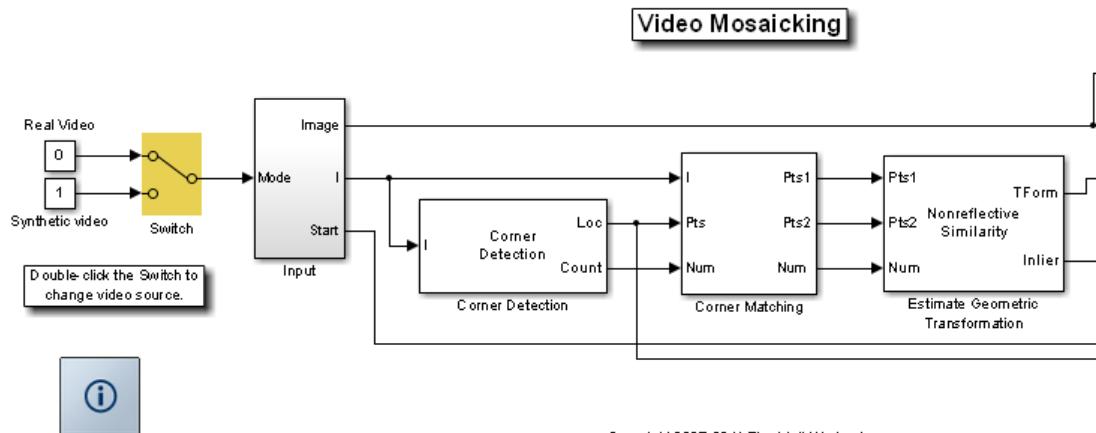
Video Mosaicking

This example shows how to create a mosaic from a video sequence. 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. The Video Mosaicking block is often used in video compression and surveillance applications.

This example illustrates how to use the Corner Detection block, the Estimate Geometric Transformation block, the Projective Transform block, and the Compositing block to create a mosaic image from a video sequence.

Example Model

The following figure shows the Video Mosaicking model:

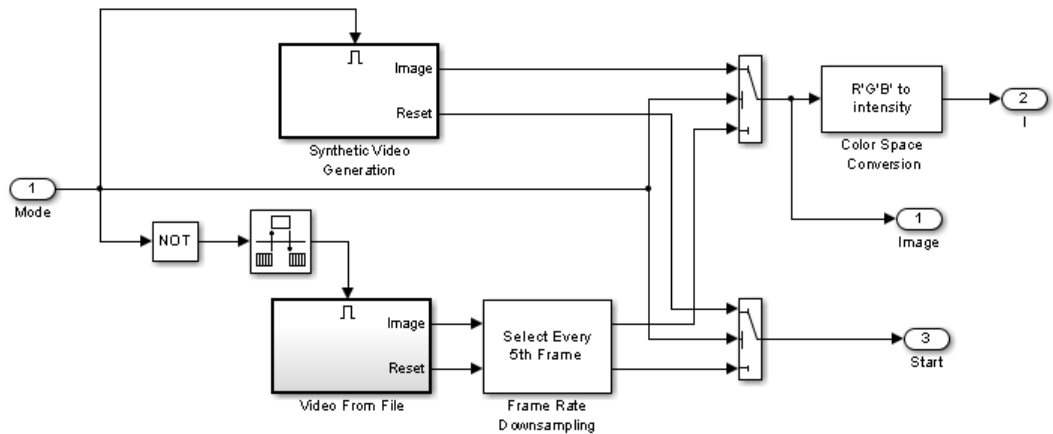


The Input subsystem loads a video sequence from either a file, or generates a synthetic video sequence. The choice is user defined. First, the Corner Detection block finds points that are matched between successive frames by the Corner Matching subsystem. Then the Estimate Geometric Transformation block computes an accurate estimate of the transformation matrix. This block uses the RANSAC algorithm to eliminate outlier input points, reducing error along the seams of the output mosaic image. Finally,

the Mosaicking subsystem overlays the current video frame onto the output image to generate a mosaic.

Input Subsystem

The Input subsystem can be configured to load a video sequence from a file, or to generate a synthetic video sequence.

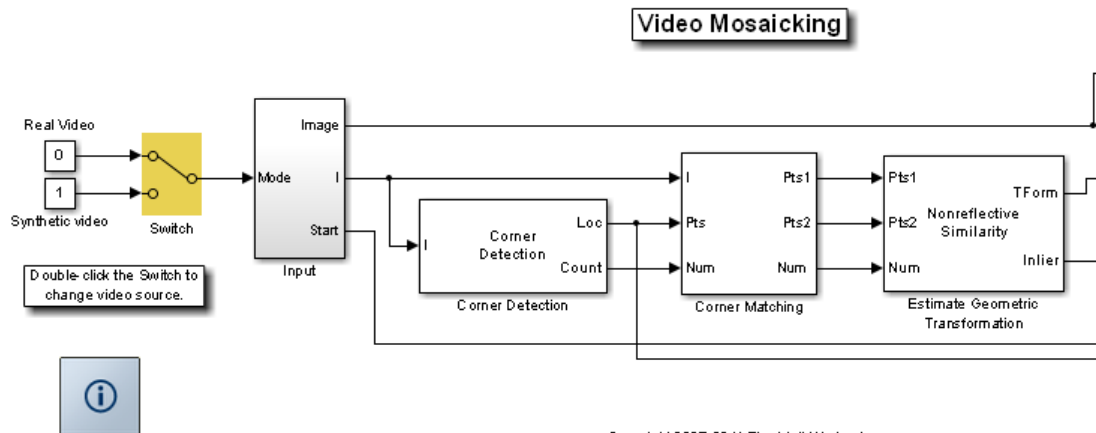


If you choose to use a video sequence from a file, you can reduce computation time by processing only some of the video frames. This is done by setting the downsampling rate in the Frame Rate Downsampling subsystem.

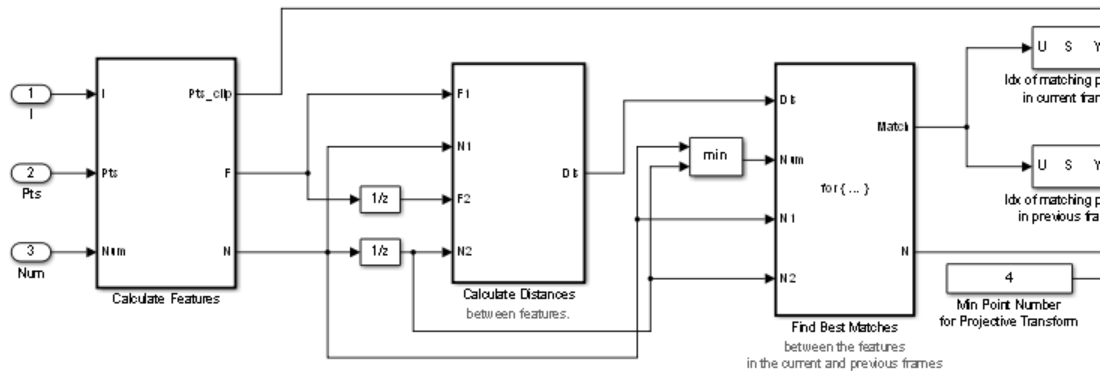
If you choose a synthetic video sequence, you can set the speed of translation and rotation, output image size and origin, and the level of noise. The output of the synthetic video sequence generator mimics the images captured by a perspective camera with arbitrary motion over a planar surface.

Corner Matching Subsystem

The subsystem finds corner features in the current video frame in one of three methods. The example uses Local intensity comparison (Rosen & Drummond), which is the fastest method. The other methods available are the Harris corner detection (Harris & Stephens) and the Minimum Eigenvalue (Shi & Tomasi).



Copyright 2007-2011 The MathWorks, Inc.

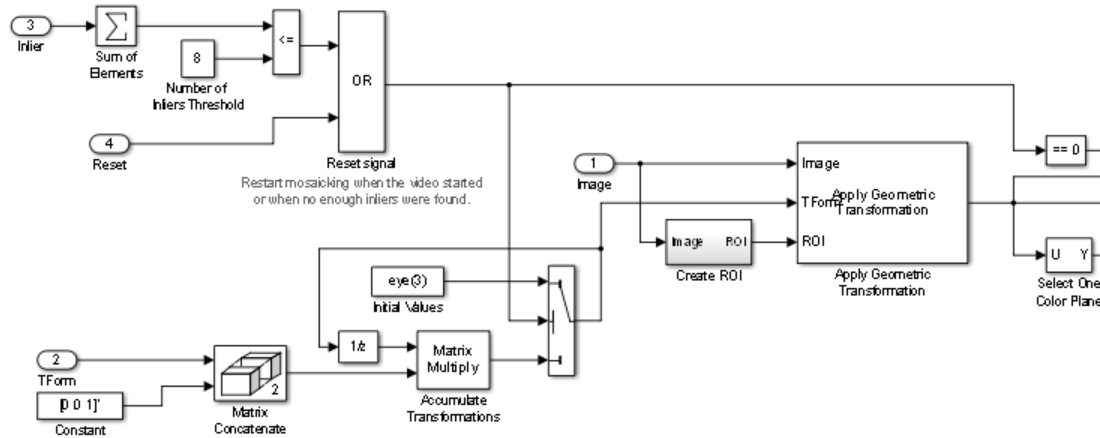


The Corner Matching Subsystem finds the number of corners, location, and their metric values. The subsystem then calculates the distances between all features in the current frame with those in the previous frame. By searching for the minimum distances, the subsystem finds the best matching features.

Mosaicking Subsystem

By accumulating transformation matrices between consecutive video frames, the subsystem calculates the transformation matrix between the current and the first video frame. The subsystem then overlays the current video frame

on to the output image. By repeating this process, the subsystem generates a mosaic image.



The subsystem is reset when the video sequence rewinds or when the Estimate Geometric Transformation block does not find enough inliers.

Video Mosaicking Using Synthetic Video

The Corners window shows the corner locations in the current video frame.



The Mosaic window shows the resulting mosaic image.



Video Mosaicking Using Captured Video

The Corners window shows the corner locations in the current video frame.



The Mosaic window shows the resulting mosaic image.



Track an Object Using Correlation

You can open the example model by typing

```
ex_vision_track_object
```

on the MATLAB command line.

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:

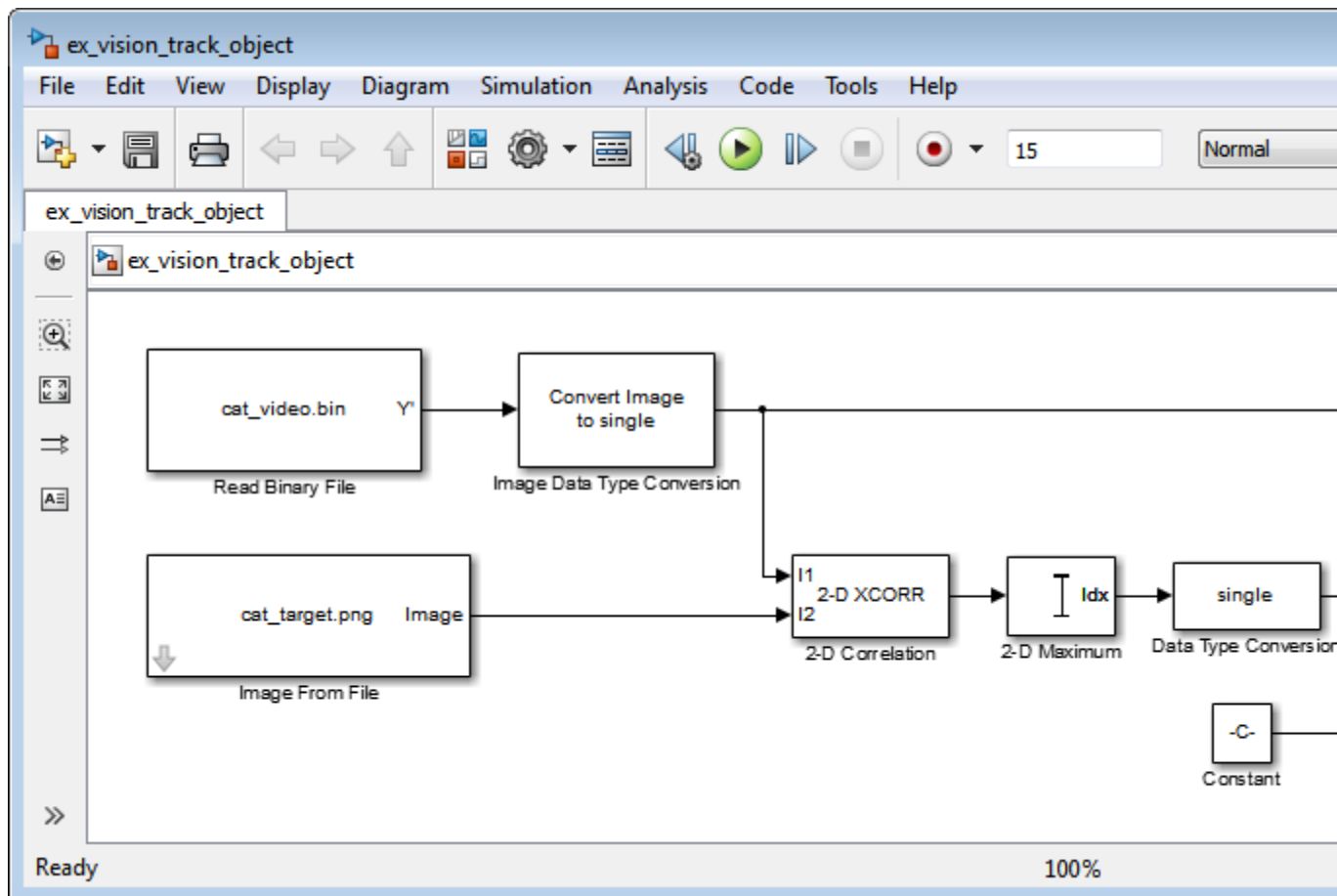
- 1 Create a new Simulink model, and add to it the blocks shown in the following table.

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 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`
- 3 Use the Image Data Type Conversion block to convert the data type of the video to single-precision floating point. Accept the default parameter.
- 4 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`
- 5 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.
- 6 Use the 2-D Maximum block to find the index of the maximum value in each input matrix. Set the **Mode** parameter to `Index`. This block outputs the zero-based location of the maximum value as a two-element vector of 32-bit unsigned integers at the **Idx** port.
- 7 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`.
- 8 Use the Constant block to define the size of the image of the cat sculpture. Set the **Constant value** parameter to `single([41 41])`.

- 9** 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.
- 10** 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.
- 11** Use the Video Viewer block to display the video stream with the ROI displayed on it. Accept the default parameters. This 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.
- 12** Connect the blocks as shown in the following figure.

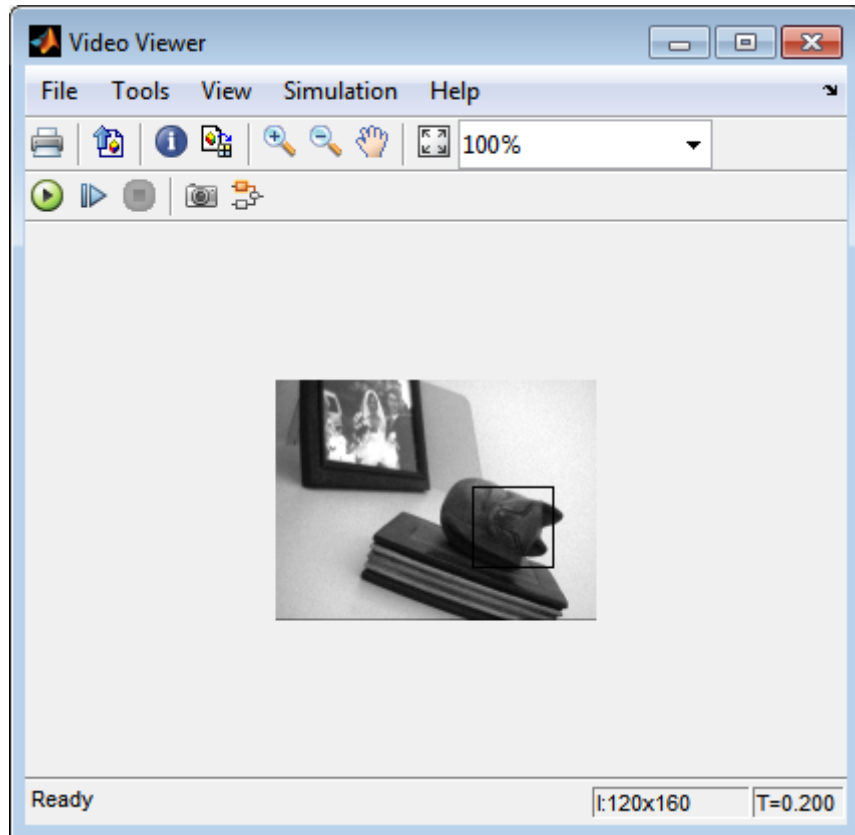


13 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

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

14 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.

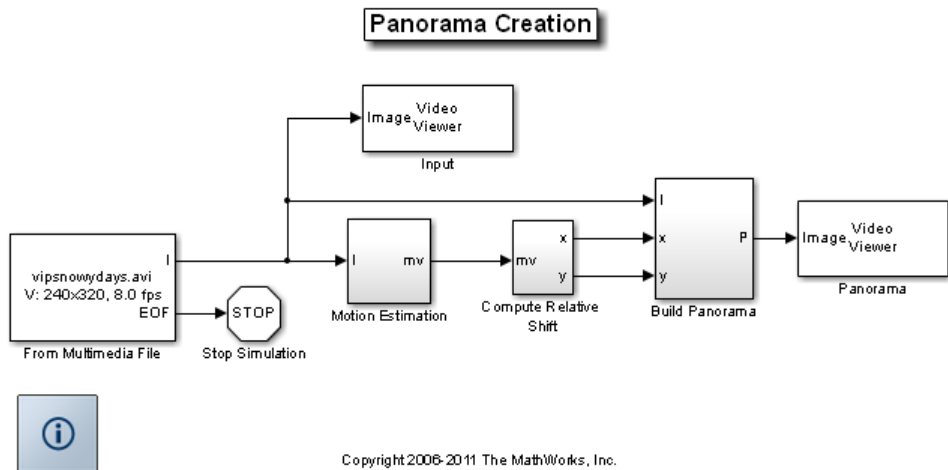
Note This example model does not provide an indication of whether or not the sculpture is present in each video frame. For an example of this type of model, type `vipattern` at the MATLAB command prompt.

Panorama Creation

This example shows how to create a panorama from a video sequence. The model calculates the motion vector between two adjacent video frames and uses it to find the portion of each frame that best matches the previous frame. Then it selects the matching portion and concatenates it with the previous frame. By repeating this process, it builds a panoramic image out of the video sequence.

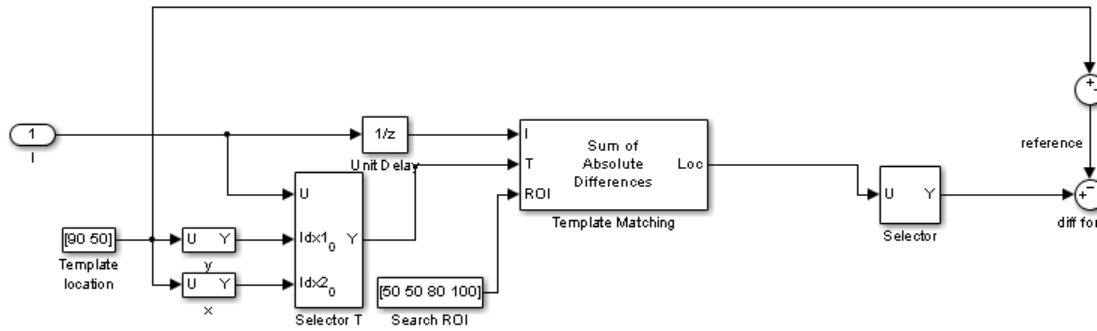
Example Model

The following figure shows the Panorama Creation model:



Motion Estimation Subsystem

This model computes the Sum of Absolute Differences (SAD) using Template Matching block to estimate the motion between consecutive video frames. Then it 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.



Panorama Creation Results

The model takes the video sequence in the Input window and creates a panorama, which it displays in the Panorama window. Note that this method of panoramic picture creation assumes there is no zooming or rotational variation in the video.



**Available Example Versions**

Windows only: `vippanorama_win.mdl`

Platform independent: `vippanorama_all.mdl`

Windows-only example models might contain compressed multimedia files or To Video Display blocks, both of which are only supported on Windows platforms. The To Video Display block supports code generation, and its performance is optimized for Windows.

Geometric Transformations

- “Rotate an Image” on page 5-2
- “Resize an Image” on page 5-8
- “Crop an Image” on page 5-12
- “Interpolation Methods” on page 5-17
- “Video Stabilization” on page 5-21
- “Video Stabilization” 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.

Note Running this example requires a DSP System Toolbox license.

ex_vision_rotate_image

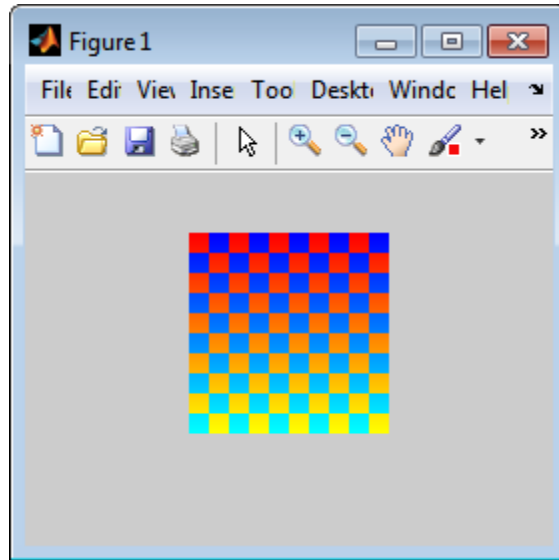
- 1 Define an RGB image in the MATLAB workspace. At the MATLAB command prompt, type

```
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 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.

- 5 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.

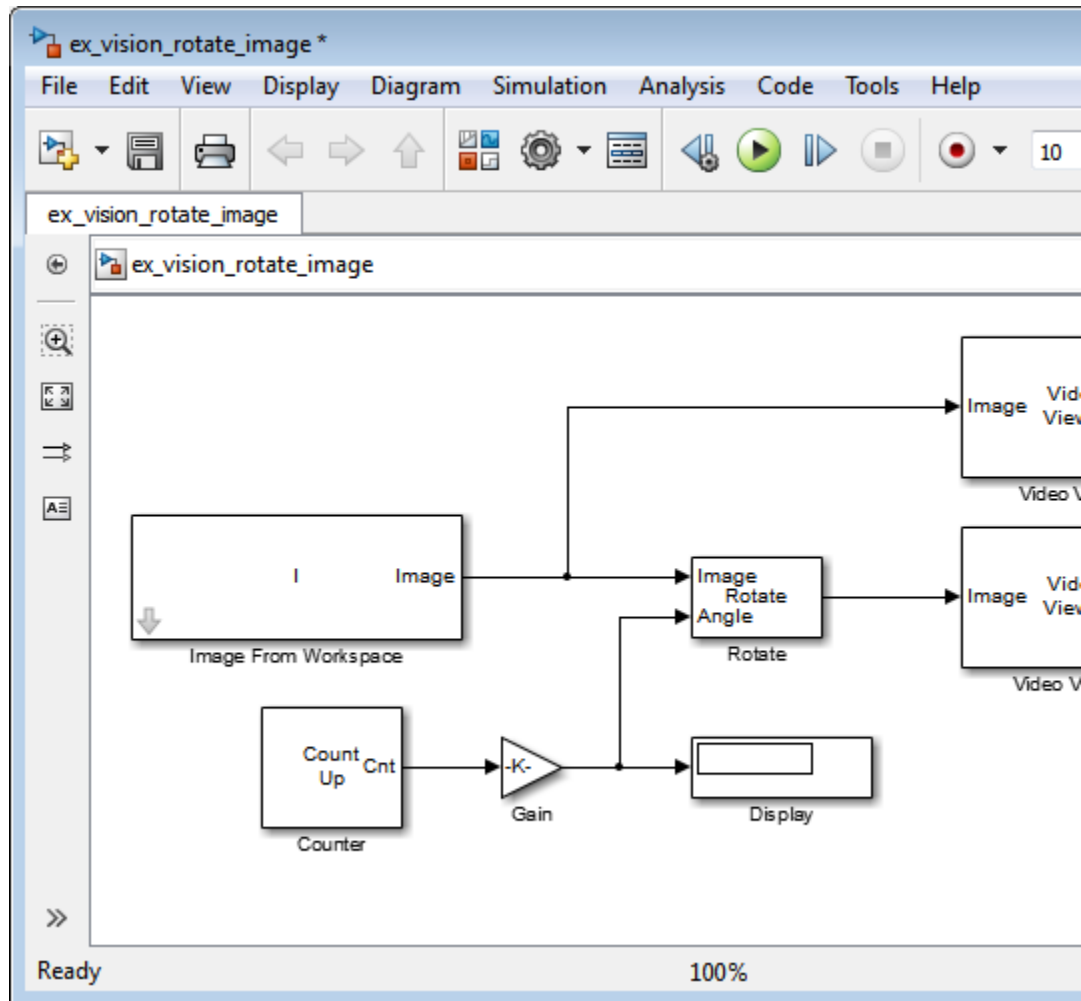
- 6 Use the Rotate block to rotate the image. Set the block parameters as follows:
 - **Rotation angle source** = Input port
 - **Sine value computation method** = Trigonometric function

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.

- 7 Use the Video Viewer1 block to display the rotating image. Accept the default parameters.
- 8 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 its output data type.

- 9 Use the Gain block to convert the output of the Counter block from degrees to radians. Set the **Gain** parameter to $\pi/180$.
- 10 Connect the blocks as shown in the following figure.

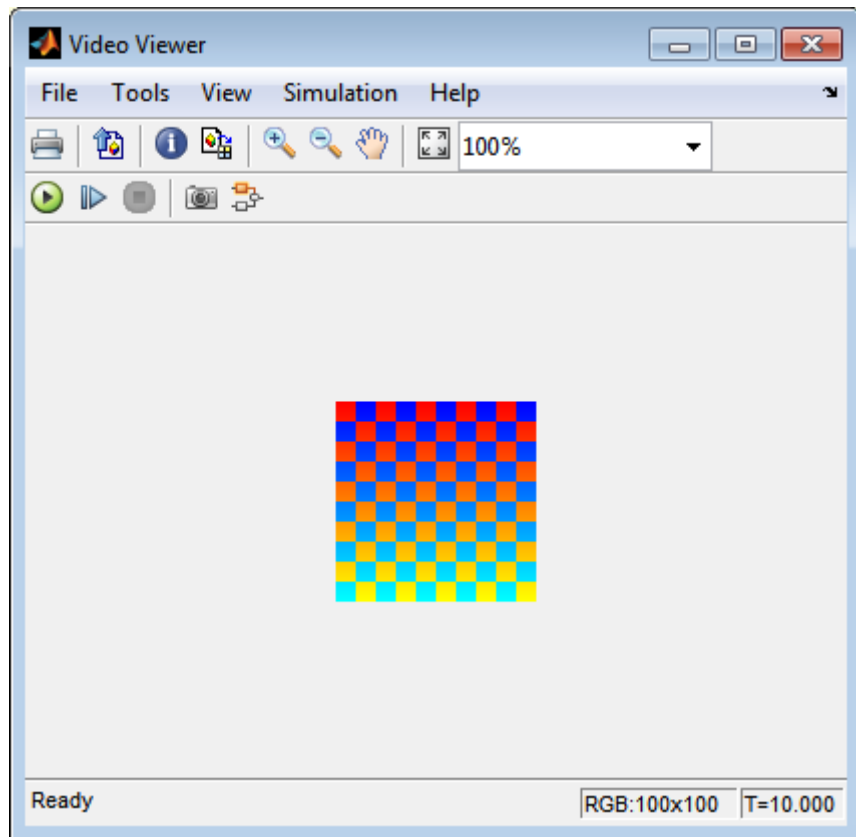


- 11 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

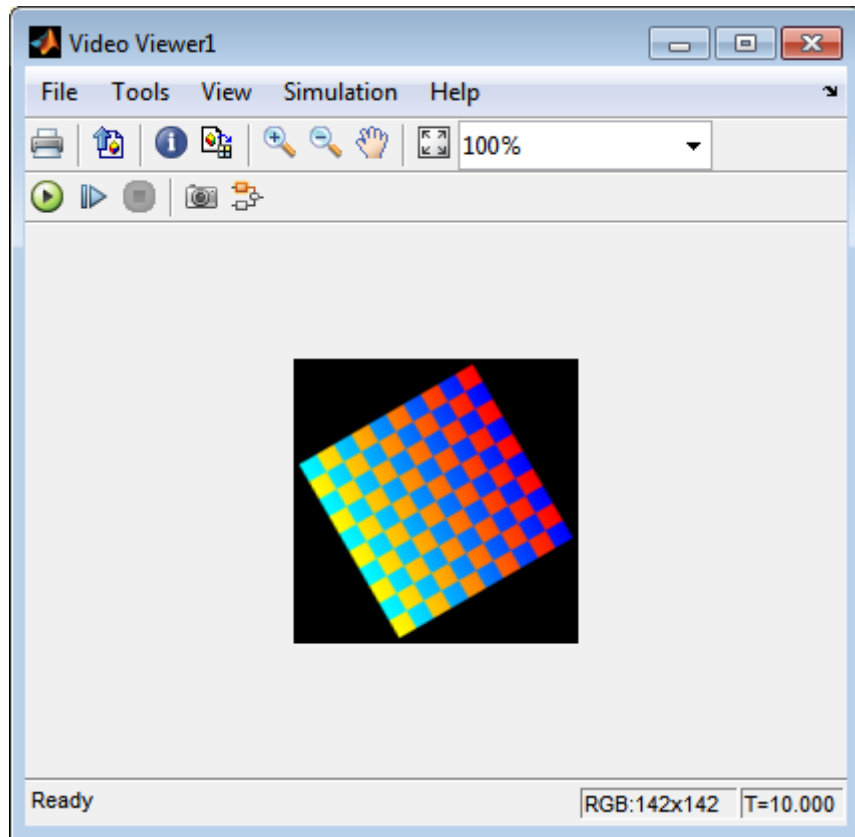
- Solver pane, **Stop time** = inf
- Solver pane, **Type** = Fixed-step
- Solver pane, **Solver** = Discrete (no continuous states)

12 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

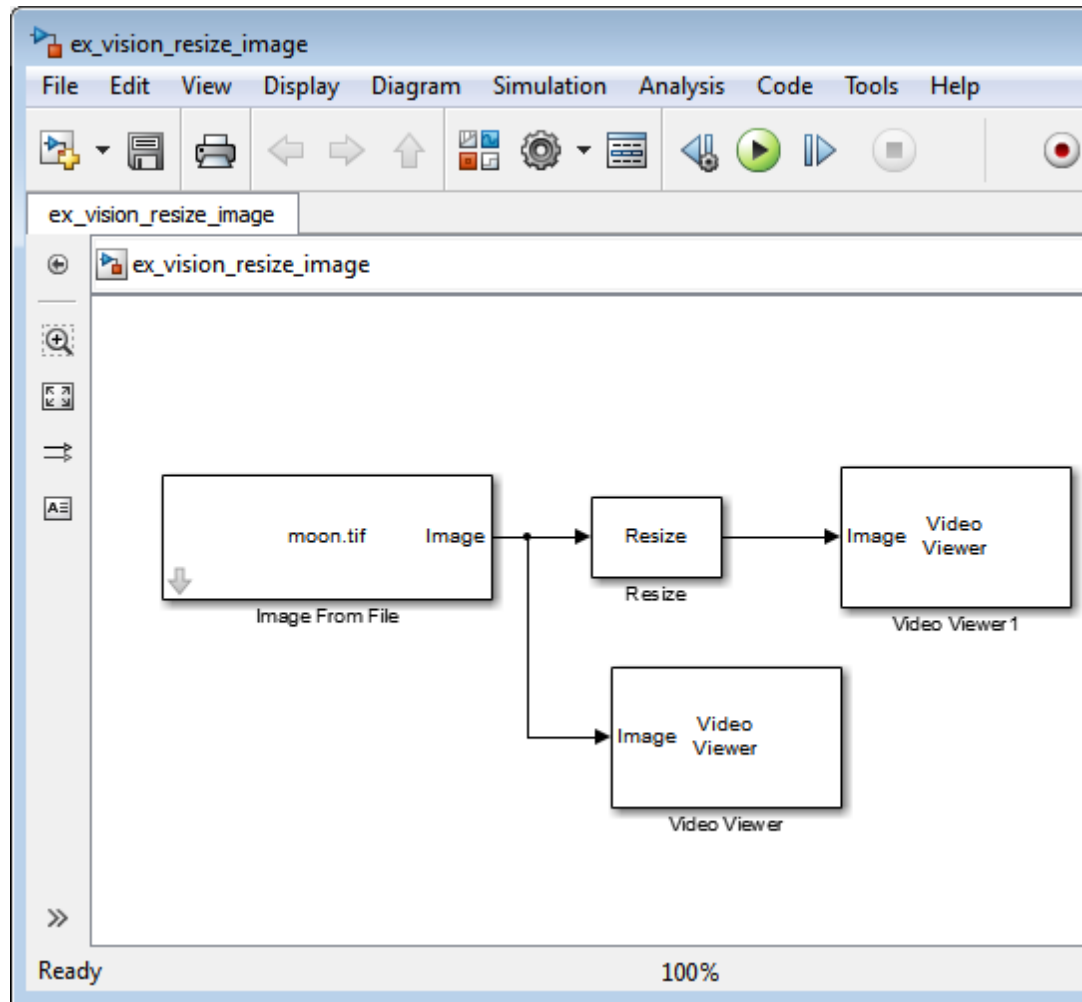
You can use the Resize block to change the size of your image or video stream. In this example, you learn how to use the Resize block to reduce the size of an image:

`ex_vision_resize_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 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.
- 3 Use the Video Viewer block to display the original image. Accept the default parameters. This block automatically displays the original image in the Video Viewer window when you run the model.
- 4 Use the Resize block to shrink the image. Set the **Resize factor in %** parameter to 50. This shrinks the image to half its original size.
- 5 Use the Video Viewer1 block to display the modified image. Accept the default parameters.
- 6 Connect the blocks as shown in the following figure.

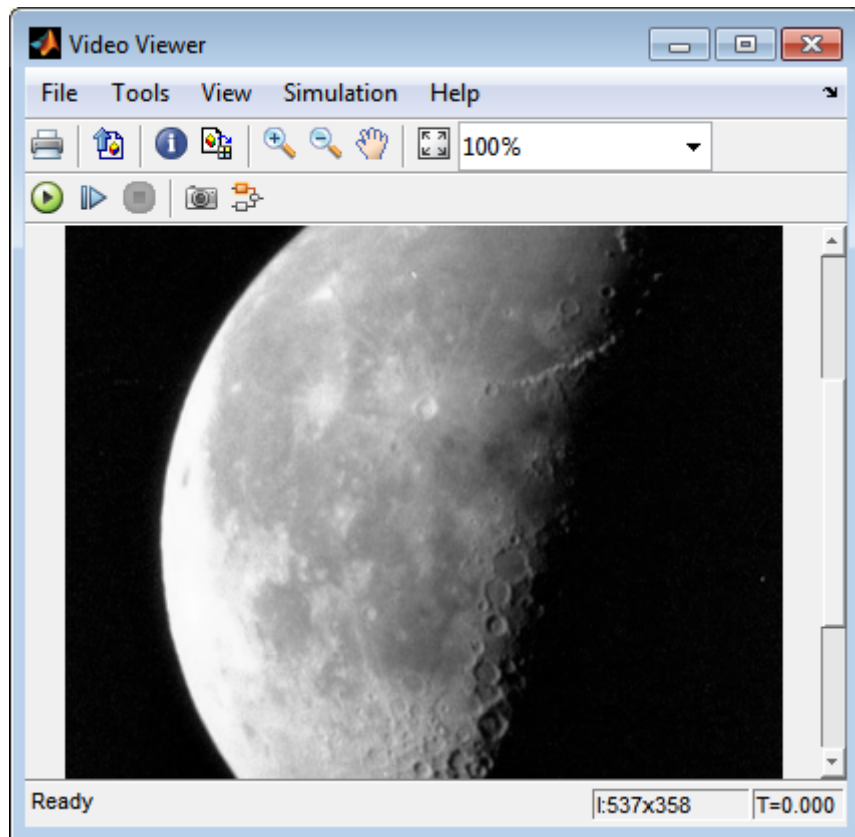


7 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

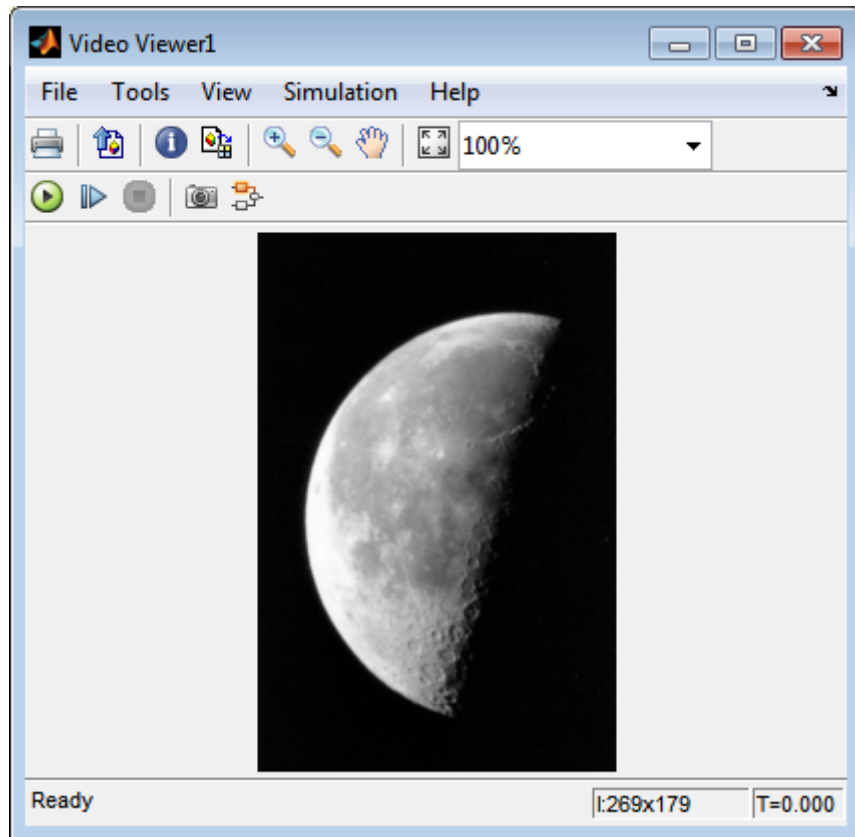
- **Solver** pane, **Stop time** = 0
- **Solver** pane, **Type** = Fixed-step
- **Solver** pane, **Solver** = Discrete (no continuous states)

8 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, Apply Geometric Transformation, Estimate Geometric Transformation, 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:

ex_vision_crop_image

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

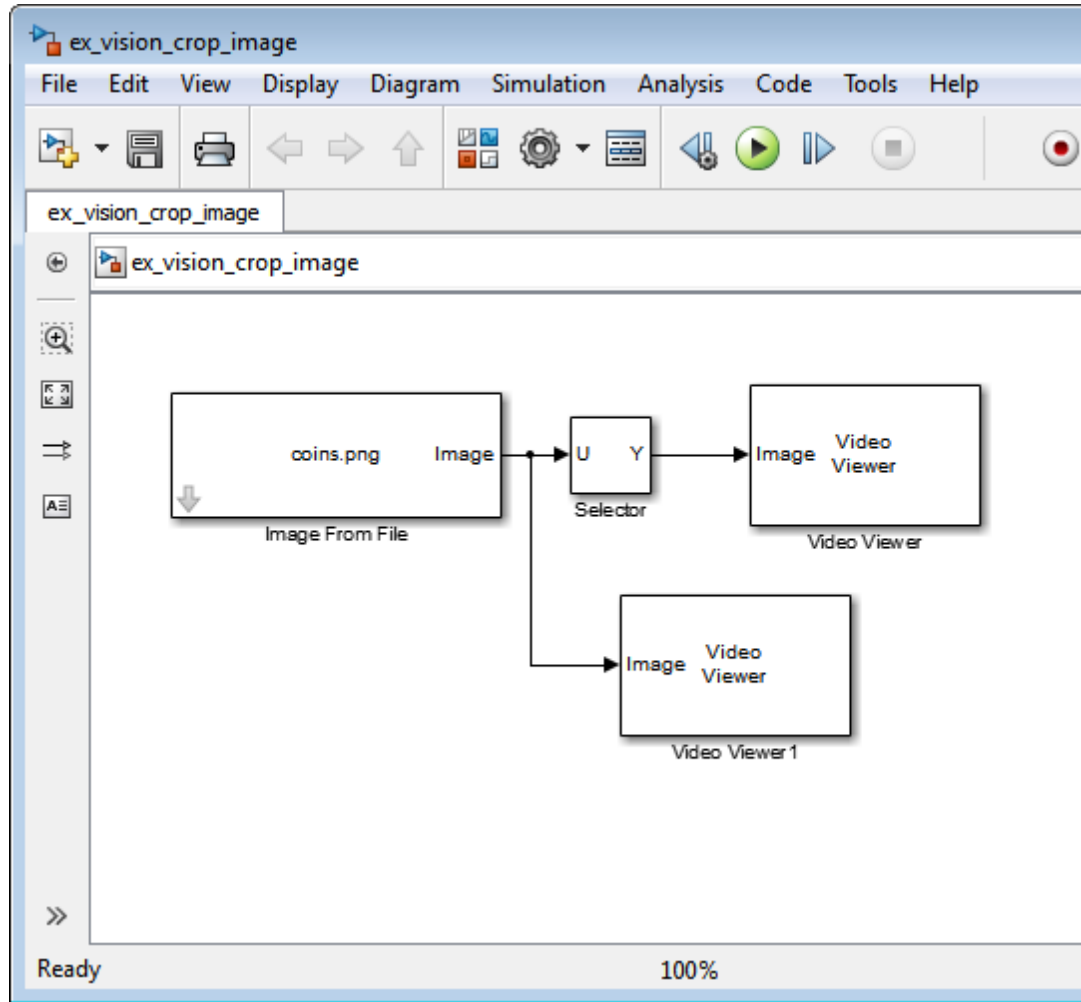
Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Video Viewer	Computer Vision System Toolbox > Sinks	2
Selector	Simulink > Signal Routing	1

- 2 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.
- 3 Use the Video Viewer block to display the original image. Accept the default parameters. This block automatically displays the original image in the Video Viewer window when you run the model.
- 4 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.

- 5** Use the Video Viewer1 block to display the cropped image. This block automatically displays the modified image in the Video Viewer window when you run the model.
- 6** Connect the blocks as shown in the following figure.

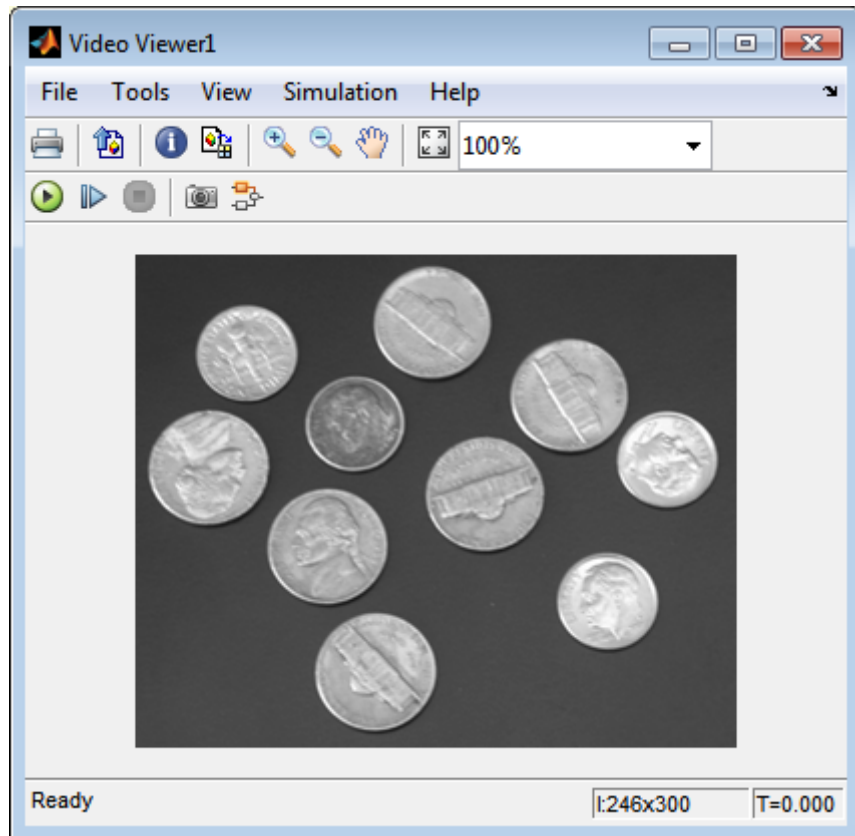


7 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

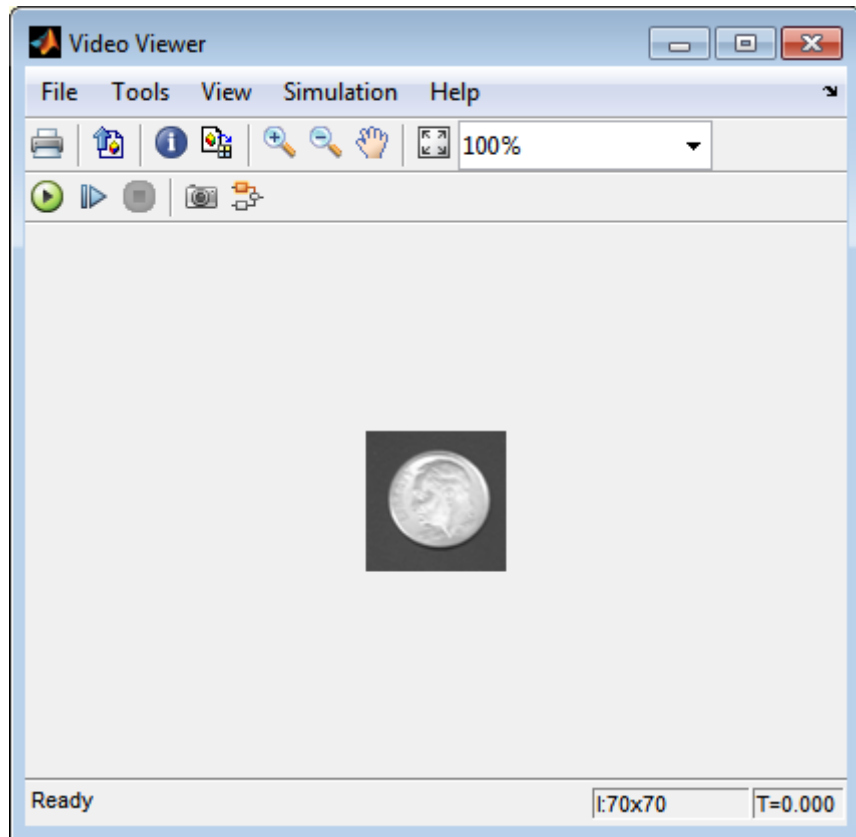
- **Solver** pane, **Stop time** = 0
- **Solver** pane, **Type** = Fixed-step
- **Solver** pane, **Solver** = Discrete (no continuous states)

8 Run the model.

The original image appears in the Video Viewer window.



The cropped image appears in the Video Viewer1 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 5-17

“Bilinear Interpolation” on page 5-18

“Bicubic Interpolation” on page 5-19

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,

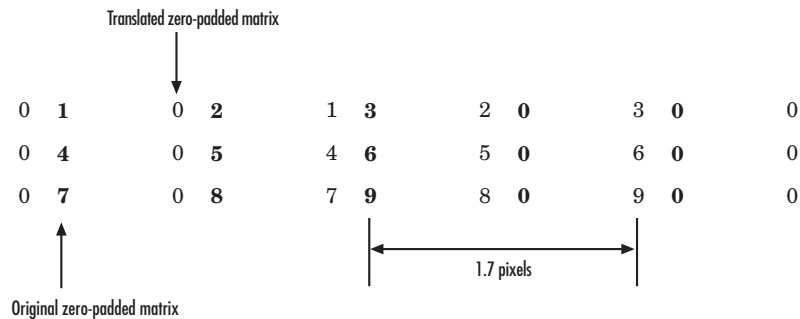
```

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

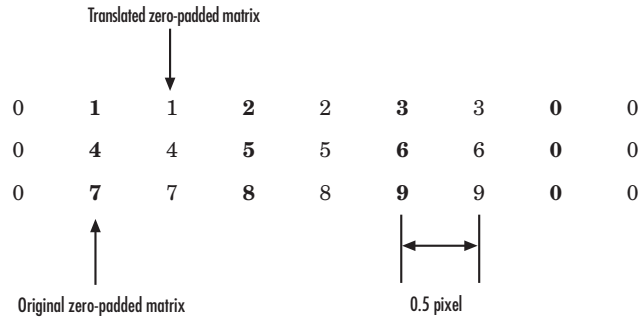
For bilinear interpolation, the block uses the weighted average of two translated pixel values for each output pixel value.

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

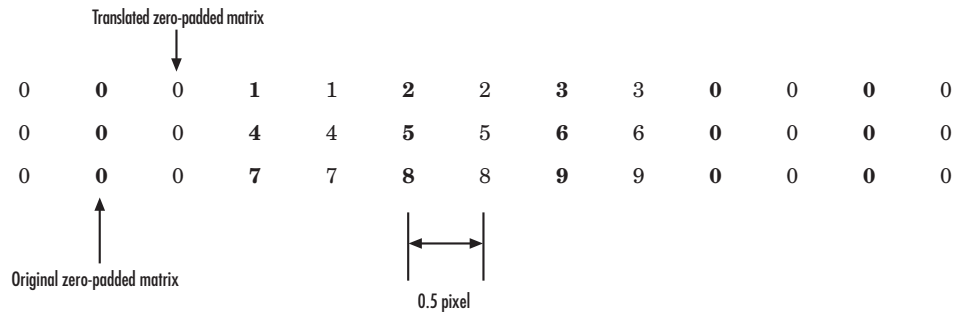
For bicubic interpolation, the block uses the weighted average of four translated pixel values for each output pixel value.

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

Video Stabilization

This example shows how to remove the effect of camera motion from a video stream.

Introduction

In this example we first define the target to track. In this case, it is the back of a car and the license plate. We also establish a dynamic search region, whose position is determined by the last known target location. We then search for the target only within this search region, which reduces the number of computations required to find the target. In each subsequent video frame, we determine how much the target has moved relative to the previous frame. We use this information to remove unwanted translational camera motions and generate a stabilized video.

Initialization

Create a System object to read video from a multimedia file. We set the output to be of intensity only video.

```
% Input video file which needs to be stabilized.
filename = 'shaky_car.avi';

hVideoSource = vision.VideoFileReader(filename, ...
                                     'ImageColorSpace', 'Intensity',...
                                     'VideoOutputDataType', 'double');
```

Create a geometric translator System object used to compensate for the camera movement.

```
hTranslate = vision.GeometricTranslator( ...
                                     'OutputSize', 'Same as input image', ...
                                     'OffsetSource', 'Input port');
```

Create a template matcher System object to compute the location of the best match of the target in the video frame. We use this location to find translation between successive video frames.

```
hTM = vision.TemplateMatcher('ROIInputPort', true, ...
                             'BestMatchNeighborhoodOutputPort', true);
```

Create a System object to highlight the search region and target in the video by drawing rectangles around them.

```
hShapeInserter = vision.ShapeInserter('BorderColor', 'White');
```

Create a System object to show the displacement values on the video.

```
hTextInserter = vision.TextInserter('%+05.1f,%+05.1f', ...  
                                     'Location', [191 215], ...  
                                     'Color', 1, ...  
                                     'FontSize', 16);
```

Create a System object to display the original video and the stabilized video.

```
hVideoOut = vision.VideoPlayer('Name', 'Video Stabilization');  
hVideoOut.Position(1) = round(0.4*hVideoOut.Position(1));  
hVideoOut.Position(2) = round(1.5*(hVideoOut.Position(2)));  
hVideoOut.Position(3:4) = [650 350];
```

Here we initialize some variables used in the processing loop.

```
pos.template_orig = [109 100]; % [x y] upper left corner  
pos.template_size = [22 18]; % [width height]  
pos.search_border = [15 10]; % max horizontal and vertical displacement  
pos.template_center = floor((pos.template_size-1)/2);  
pos.template_center_pos = (pos.template_orig + pos.template_center - 1);  
fileInfo = info(hVideoSource);  
W = fileInfo.VideoSize(1); % Width in pixels  
H = fileInfo.VideoSize(2); % Height in pixels  
BorderCols = [1:pos.search_border(1)+4 W-pos.search_border(1)+4:W];  
BorderRows = [1:pos.search_border(2)+4 H-pos.search_border(2)+4:H];  
sz = fileInfo.VideoSize;  
TargetRowIndices = ...  
    pos.template_orig(2)-1:pos.template_orig(2)+pos.template_size(2)-2;  
TargetColIndices = ...  
    pos.template_orig(1)-1:pos.template_orig(1)+pos.template_size(1)-2;  
SearchRegion = pos.template_orig - pos.search_border - 1;  
Offset = [0 0];  
Target = zeros(18,22);  
firstTime = true;
```

Stream Processing Loop

This is the main processing loop which uses the objects we instantiated above to stabilize the input video.

```

while ~isDone(hVideoSource)
    input = step(hVideoSource);

    % Find location of Target in the input video frame
    if firstTime
        Idx = int32(pos.template_center_pos);
        MotionVector = [0 0];
        firstTime = false;
    else
        IdxPrev = Idx;

        ROI = [SearchRegion, pos.template_size+2*pos.search_border];
        Idx = step(hTM, input, Target, ROI);

        MotionVector = double(Idx-IdxPrev);
    end

    [Offset, SearchRegion] = updatesearch(sz, MotionVector, ...
        SearchRegion, Offset, pos);

    % Translate video frame to offset the camera motion
    Stabilized = step(hTranslate, input, fliplr(Offset));

    Target = Stabilized(TargetRowIndices, TargetColIndices);

    % Add black border for display
    Stabilized(:, BorderCols) = 0;
    Stabilized(BorderRows, :) = 0;

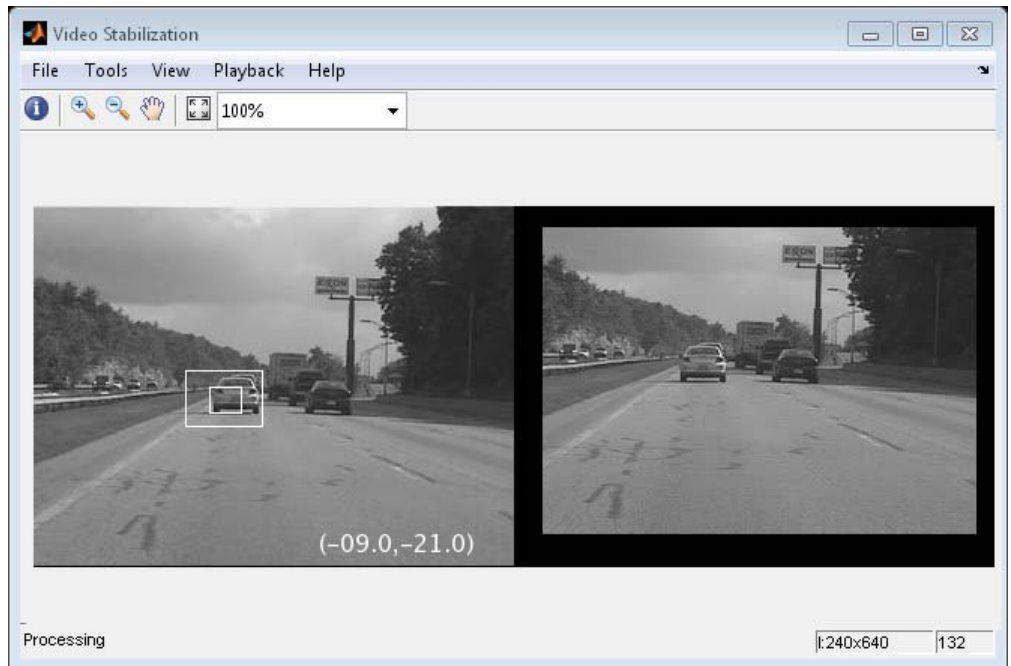
    TargetRect = [pos.template_orig-Offset, pos.template_size];
    SearchRegionRect = [SearchRegion, pos.template_size + 2*pos.search_bord

    % Draw rectangles on input to show target and search region
    input = step(hShapeInserter, input, [TargetRect; SearchRegionRect]);

```

```
% Display the offset values on the input image
input = step(hTextInserter, input, Offset);

% Display video
step(hVideoOut, [input Stabilized]);
end
```



Release

Here you call the release method on the objects to close any open files and devices.

```
release(hVideoSource);
```

Conclusion

Using the Computer Vision System Toolbox™ functionality from MATLAB command line it is easy to implement complex systems like video stabilization.

Appendix

The following helper function is used in this example.

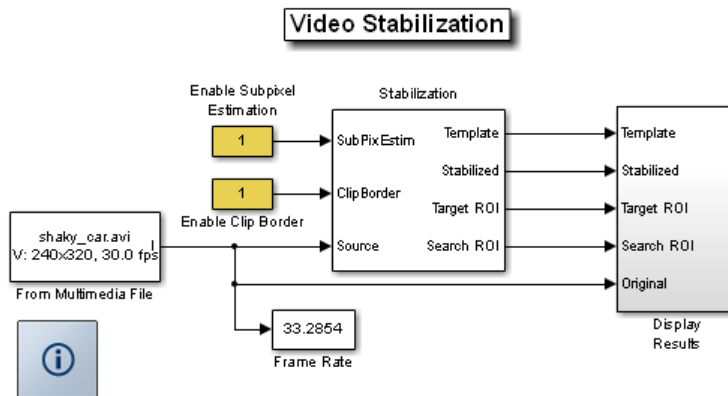
- `updatesearch.m`

Video Stabilization

This example shows how to remove 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, whose position is determined by the last known target location. The model only searches for the target within this search region, which reduces the number of computations required to find the target. In each subsequent video frame, the model determines how much the target has moved relative to the previous frame. It uses this information to remove unwanted translational camera motions and generate a stabilized video.

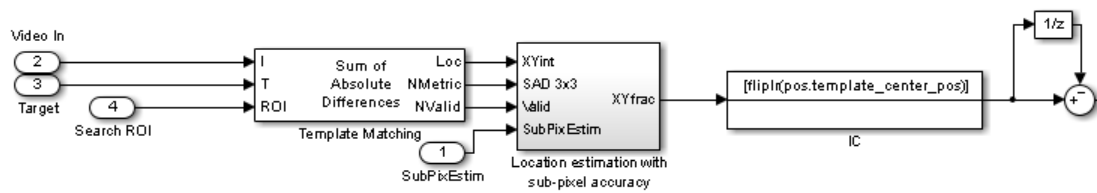
Example Model

The following figure shows the Video Stabilization model:



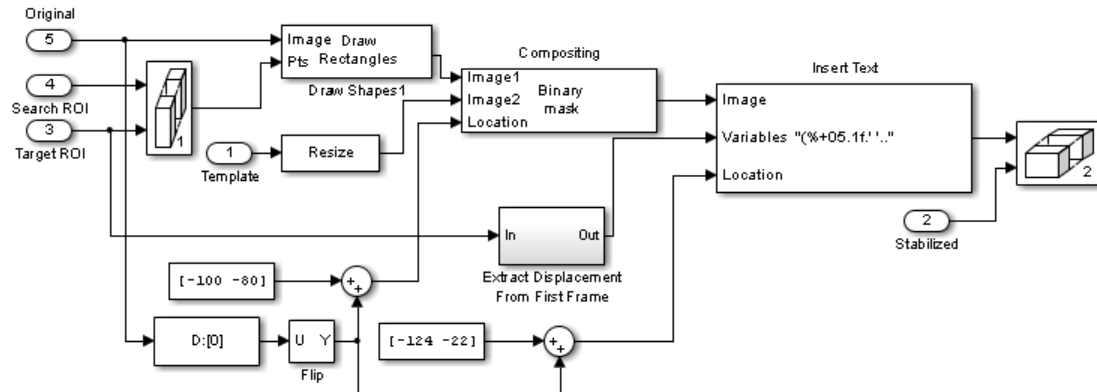
Estimate Motion Subsystem

The model uses the Template Matching block to move the target over the search region and compute the Sum of Absolute Differences (SAD) at each location. The location with the lowest SAD value corresponds to the location of the target in the video frame. Based on the location information, the model computes the displacement vector between the target and its original location. The Translate block in the Stabilization subsystem uses this vector to shift each frame so that the camera motion is removed from the video stream.



Display Results Subsystem

The model uses the Resize, Compositing, and Insert Text blocks to embed the enlarged target and its displacement vector on the original video.



Video Stabilization Results

The figure on the left shows the original video. The figure on the right shows the stabilized video.



Available Example Versions

Floating-point versions of this example:

Windows only: vipstabilize_win.mdl

Platform independent: vipstabilize_all.mdl

Fixed-point versions of this example:

Windows only: vipstabilize_fixpt_win.mdl

Platform independent: vipstabilize_fixpt_all.mdl

Fixed-point versions of this example that simulate row major data organization:

Windows only: vipstabilize_fixpt_rowmajor_win.mdl

Platform independent: vipstabilize_fixpt_rowmajor_all.mdl

Windows-only example models might contain compressed multimedia files or To Video Display blocks, both of which are only supported on Windows platforms. The To Video Display block supports code generation, and its performance is optimized for Windows.

Filters, Transforms, and Enhancements

- “Adjust the Contrast of Intensity Images” on page 6-2
- “Adjust the Contrast of Color Images” on page 6-7
- “Remove Salt and Pepper Noise from Images” on page 6-12
- “Sharpen an Image” on page 6-18

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.

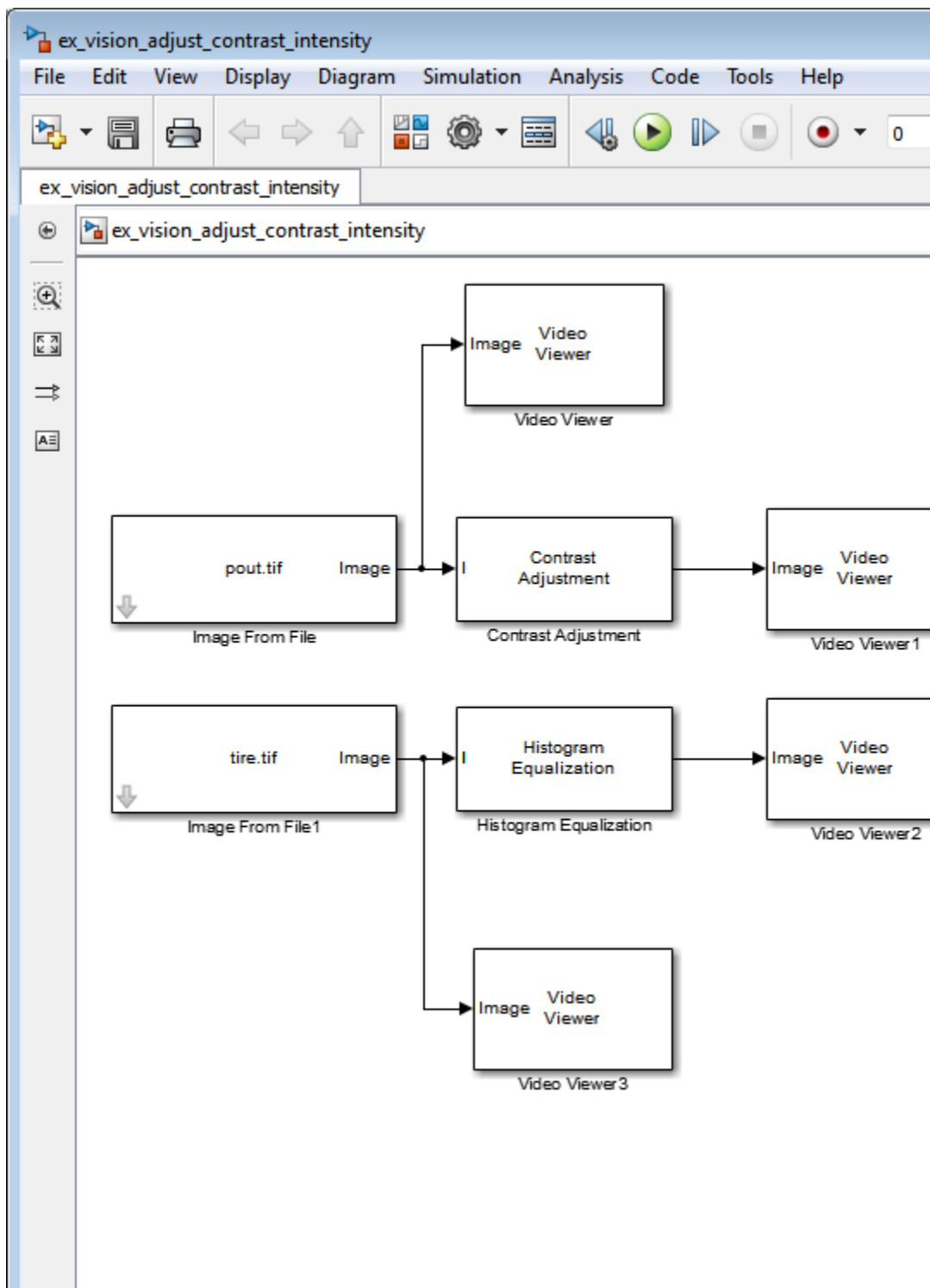
`ex_vision_adjust_contrast_intensity`

- 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 listed in the table above into your new model.
- 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`. 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.

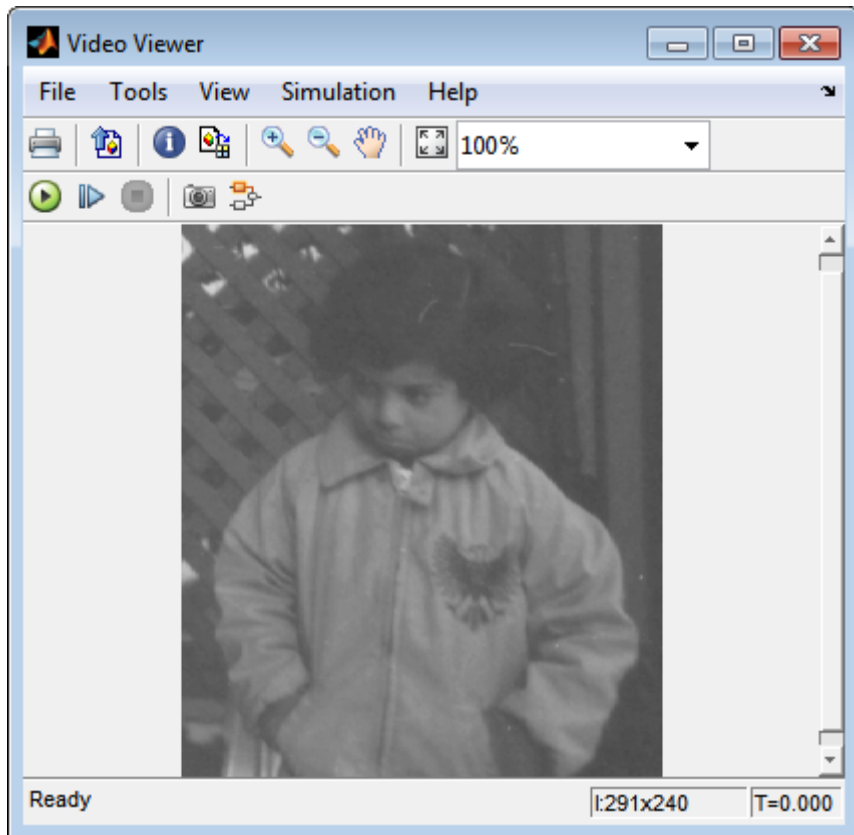


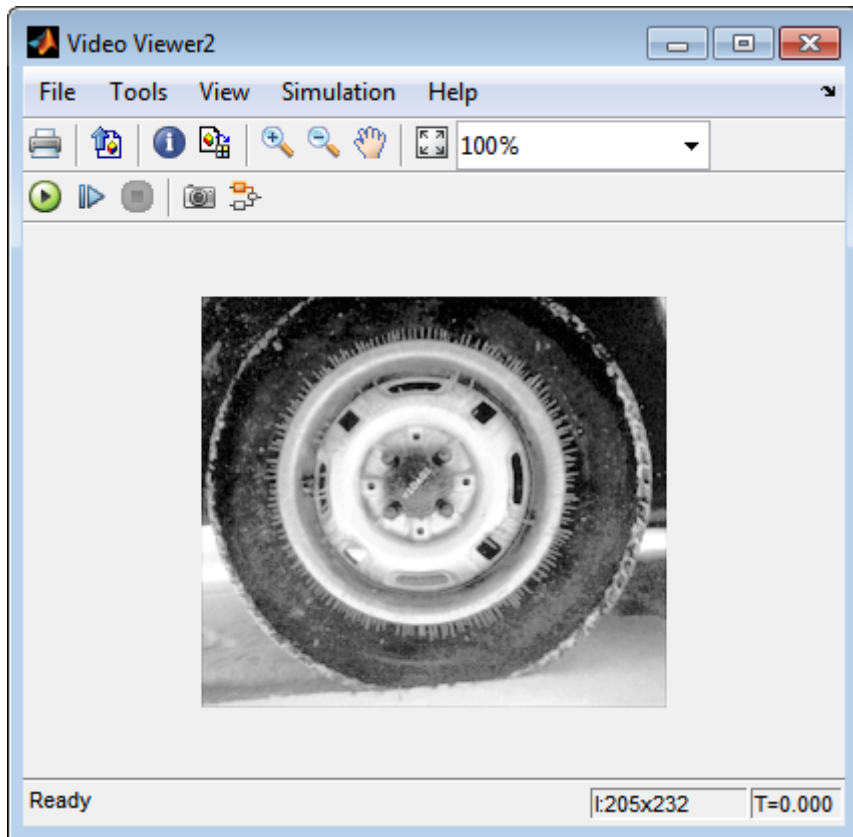
9 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

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

10 Run the model.

The 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.

ex_vision_adjust_contrast_color.mdl

- 1 Use the following code to read in an indexed RGB image, `shadow.tif`, and convert it to an RGB image. The model provided above already includes this code in file > Model Properties > Model Properties > InitFcn, and executes it prior to simulation.

```
[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 listed in the table above into your new model.
- 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. 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. Accept the default parameters.

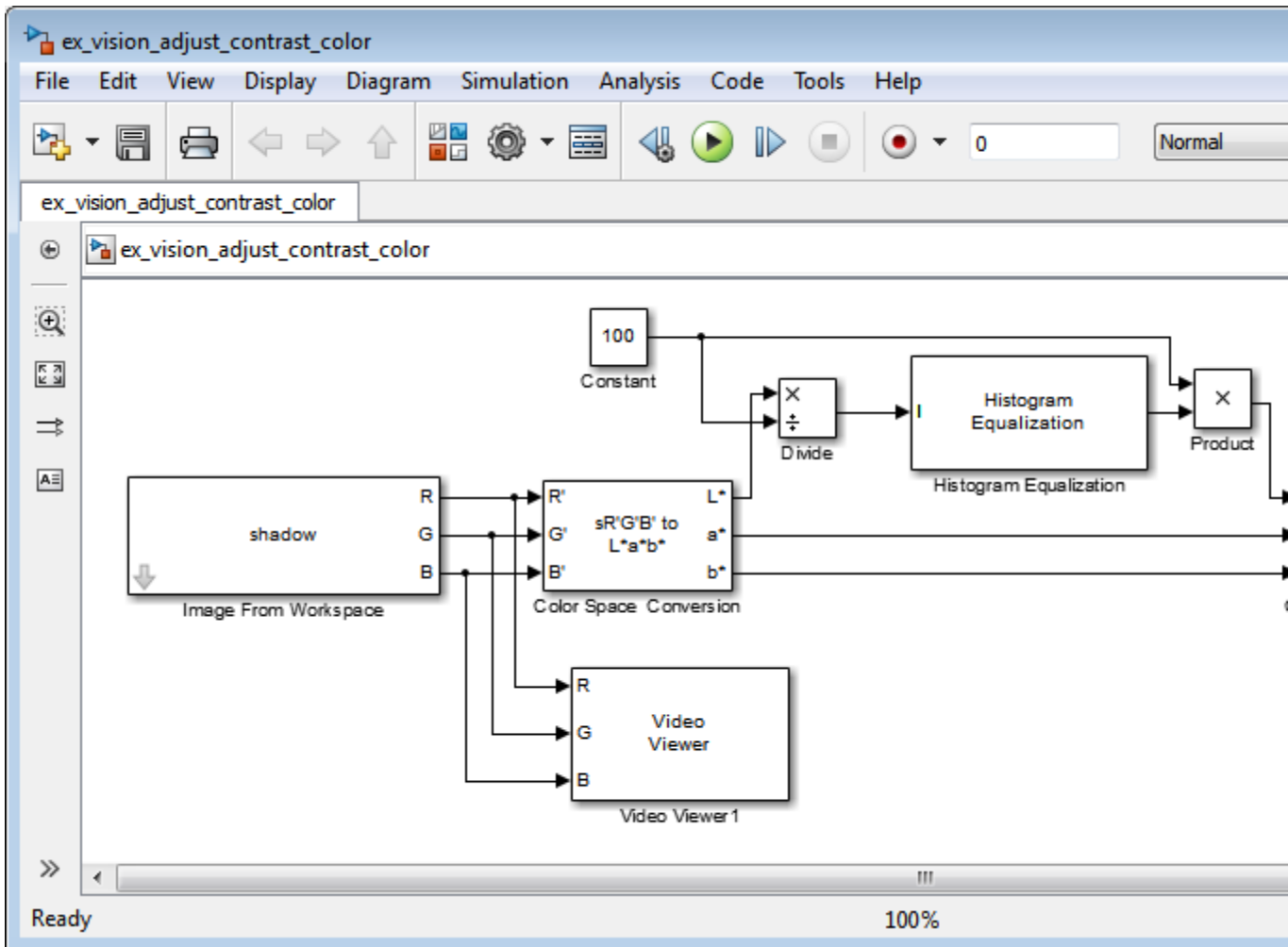
9 Use the Product block to scale the values back to be between the 0 to 100 range. Accept the default parameters.

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.

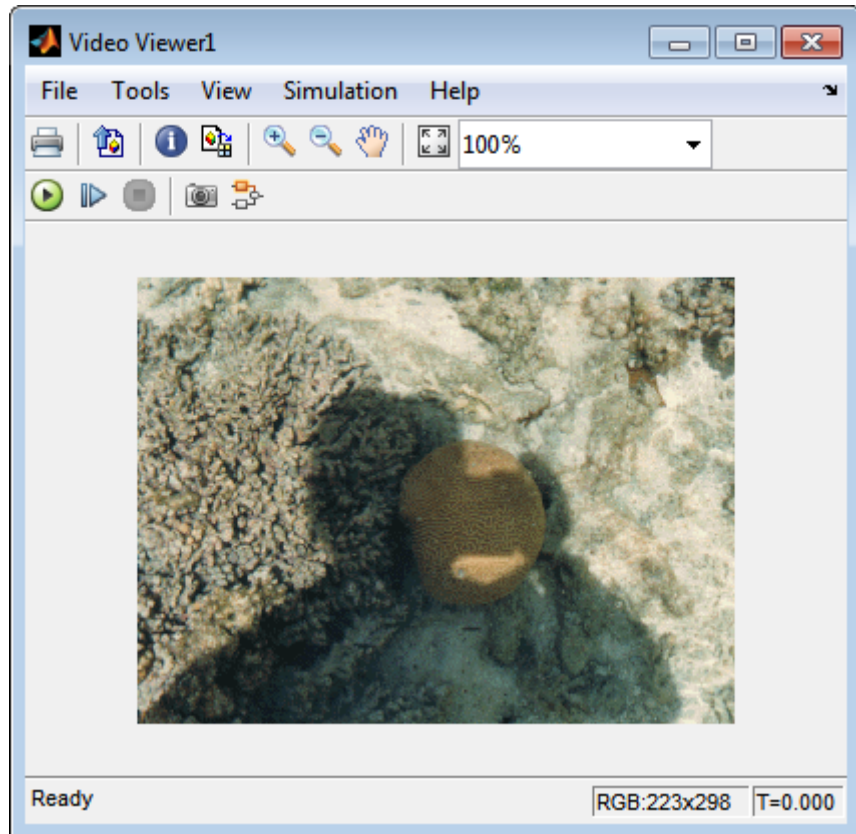


13 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

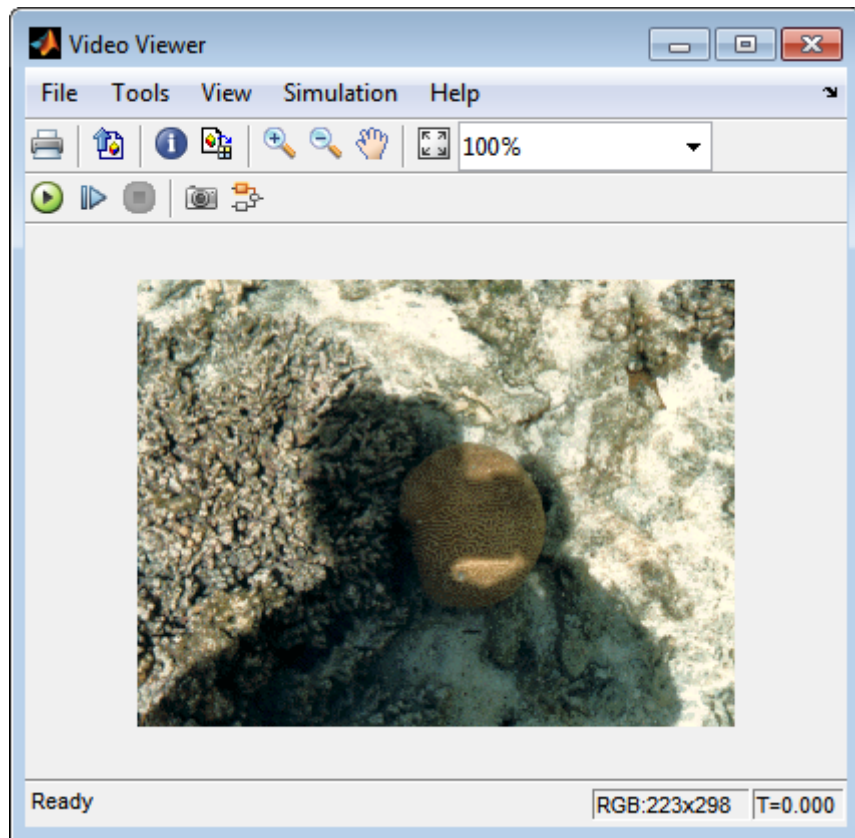
- **Solver** pane, **Stop time** = 0
- **Solver** pane, **Type** = Fixed-step
- **Solver** pane, **Solver** = Discrete (no continuous states)

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 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:

```
ex_vision_remove_noise
```

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

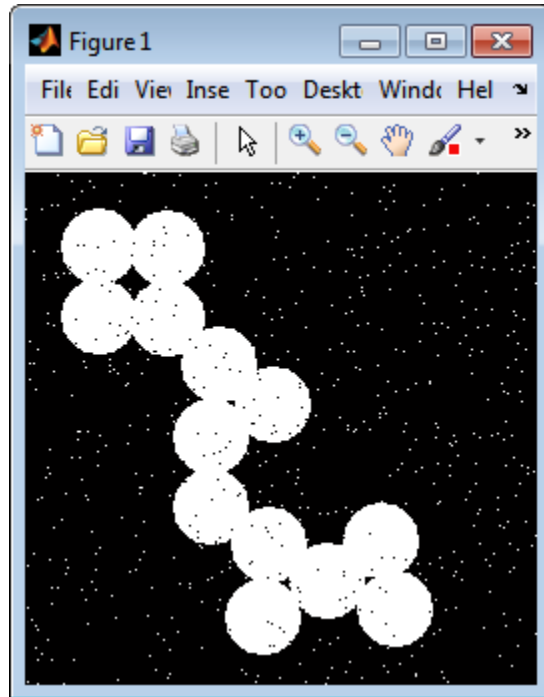
```
I= double(imread('circles.png'));  
I= imnoise(I,'salt & pepper',0.02);
```

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

The model provided with this example already includes this code in `file>Model Properties>Model Properties>InitFcn`, and executes it prior to simulation.

- 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.

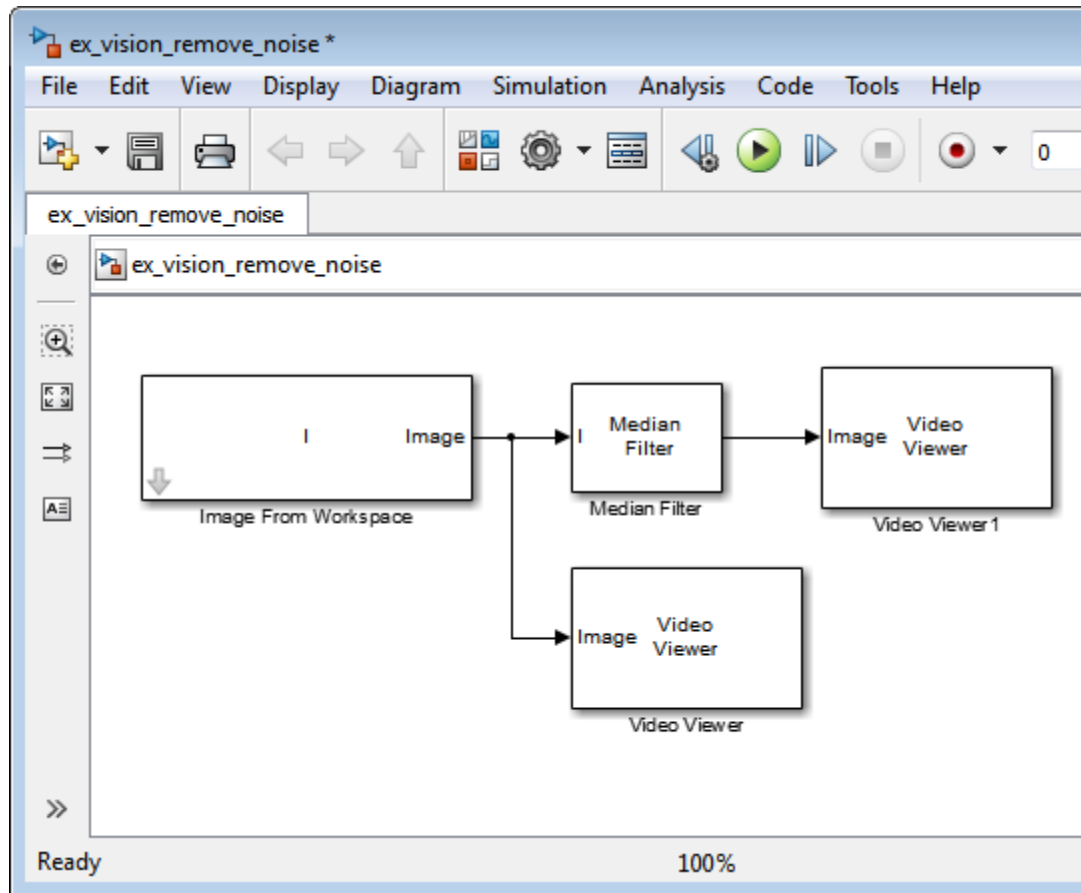
Block	Library	Quantity
Image From Workspace	Computer Vision System Toolbox > Sources	1
Median Filter	Computer Vision System Toolbox > Filtering	1
Video Viewer	Computer Vision System Toolbox > Sinks	2

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

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

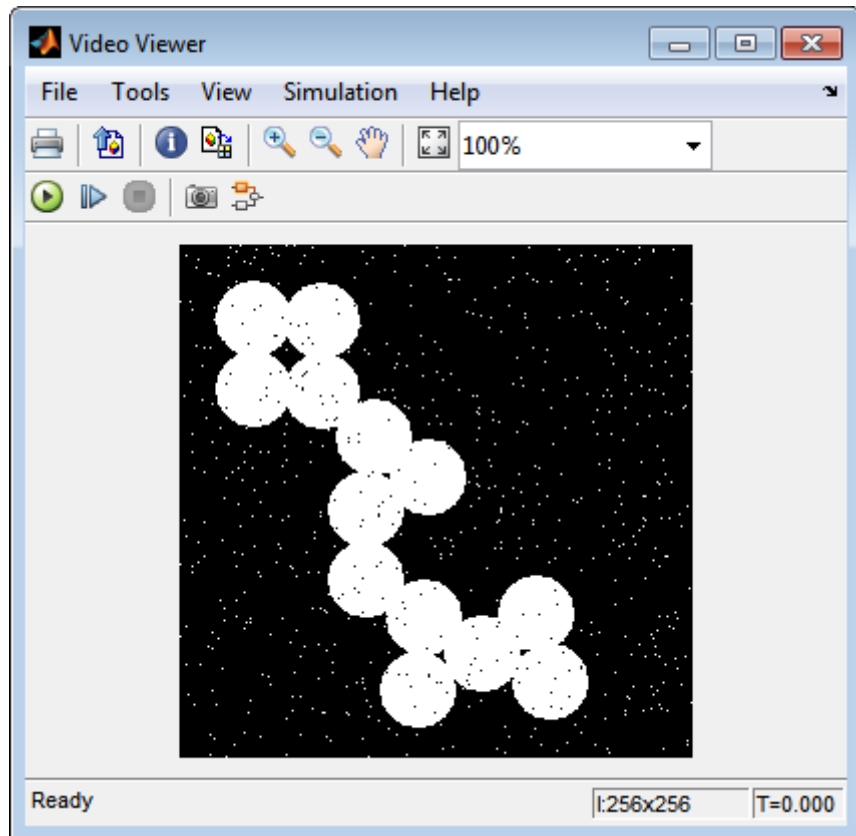
The Median Filter block replaces the central value of the 3-by-3 neighborhood with the median value of the neighborhood. This process removes the noise in the image.

- 6 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.
- 7 Connect the blocks as shown in the following figure.

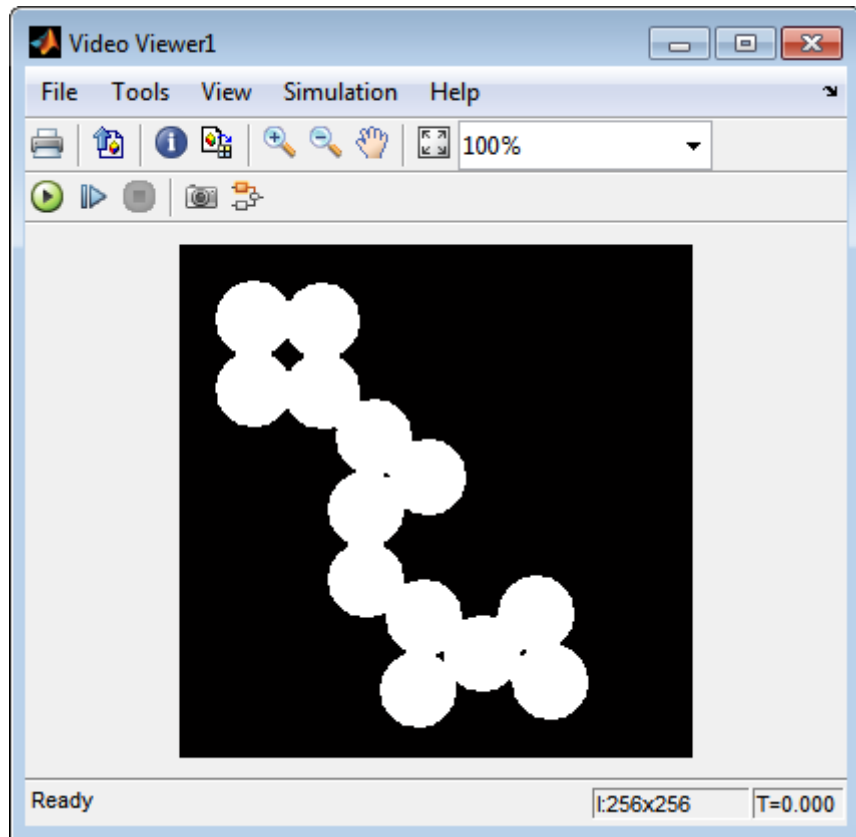


- 8 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:
 - **Solver** pane, **Stop time** = 0
 - **Solver** pane, **Type** = Fixed-step
 - **Solver** pane, **Solver** = Discrete (no continuous states)
- 9 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.

ex_vision_sharpen_image

- 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

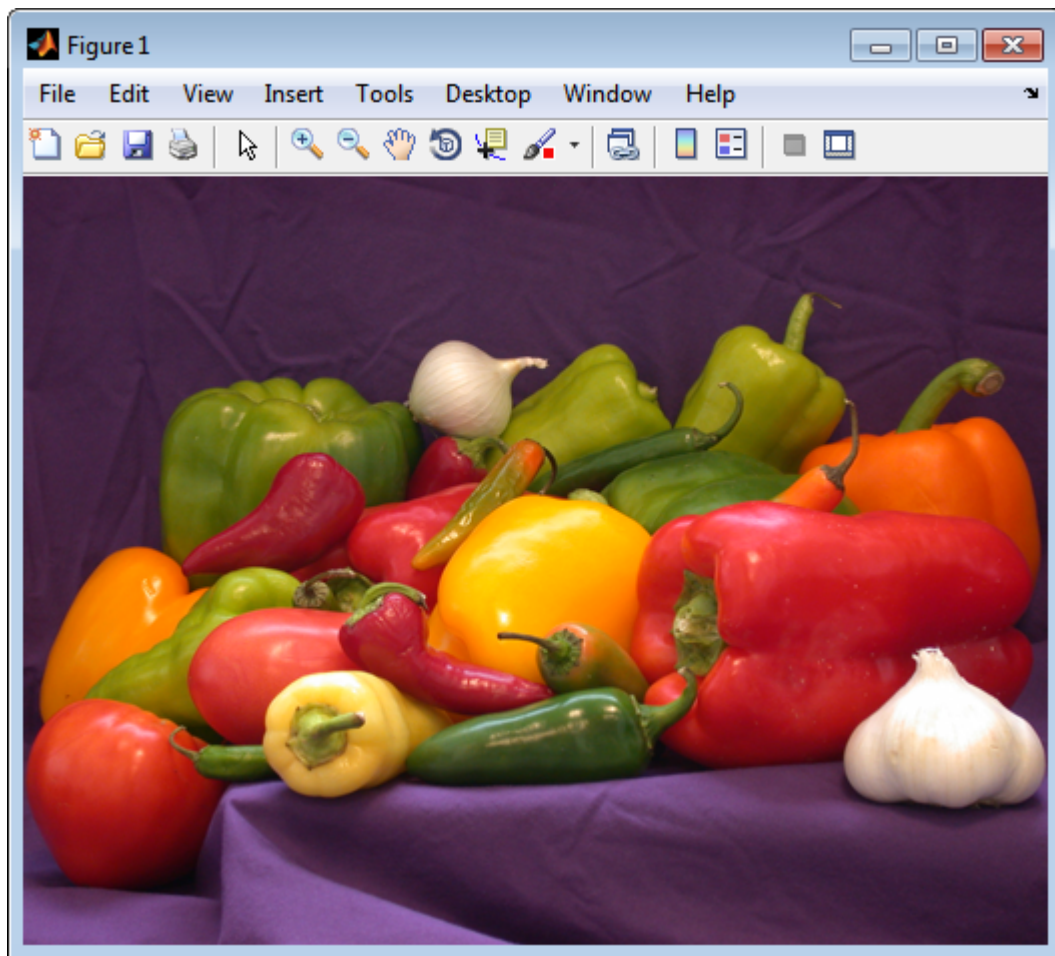
```
I = im2double(imread('peppers.png'));
```

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.

The model provided with this example already includes this code in `file>Model Properties>Model Properties>InitFcn`, and executes it prior to simulation.

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

```
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** 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

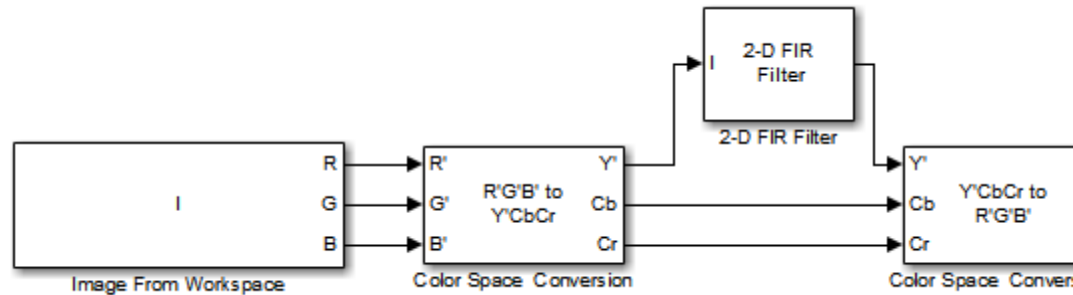
The block outputs the R', G', and B' planes of the I array at the output ports.

- 5** 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
- 6** 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.

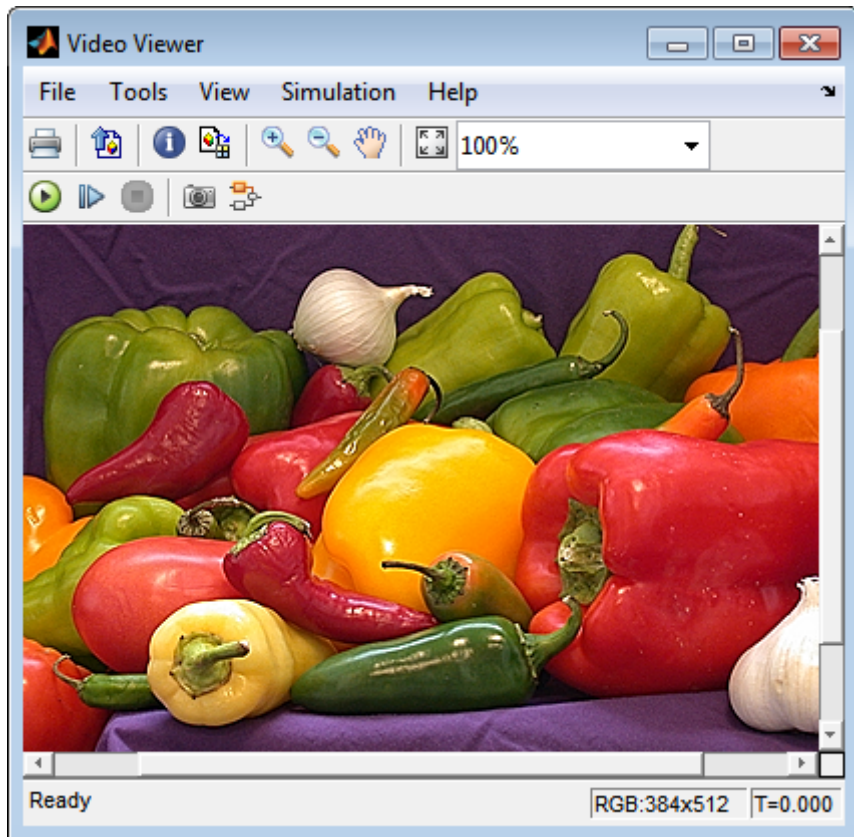
- 7** The second Color Space Conversion block converts the color information from the Y'CbCr color space to the R'G'B' color space. Set the block parameters as follows:
- **Conversion** = Y'CbCr to R'G'B'

- **Image signal** = Separate color signals
- 8 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**.
 - 9 Connect the blocks as shown in the following figure.



- 10 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)
- 11 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.

Statistics and Morphological Operations

- “Find the Histogram of an Image” on page 7-2
- “Correct Nonuniform Illumination” on page 7-8
- “Count Objects in an Image” on page 7-15

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:

Note Running this example requires a DSP System Toolbox license.

You can open the example model by typing

```
ex_vision_find_histogram
```

on the MATLAB command line.

- 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 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

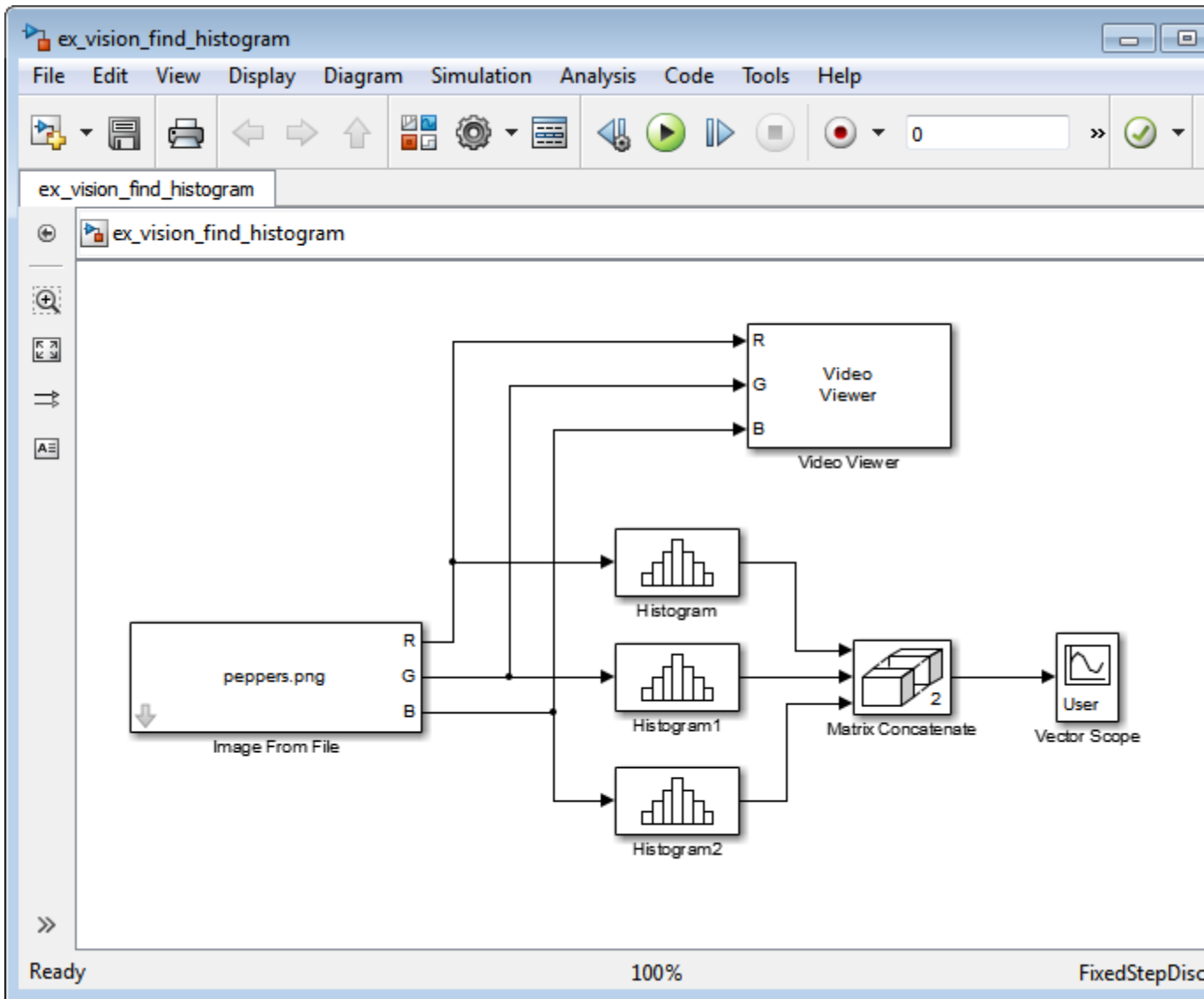
- On the Data Types tab, **Output data type:** = double
- 3** Use the Video Viewer block to automatically display the original image in the viewer window when you run the model. Set the **Image signal** parameter to `Separate color signals` from the File menu.
 - 4** Use the Histogram blocks to calculate 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
 - **Find the histogram over:** = Entire Input

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.

- 5** 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.
- 6** 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]

7 Connect the blocks as shown in the following figure.

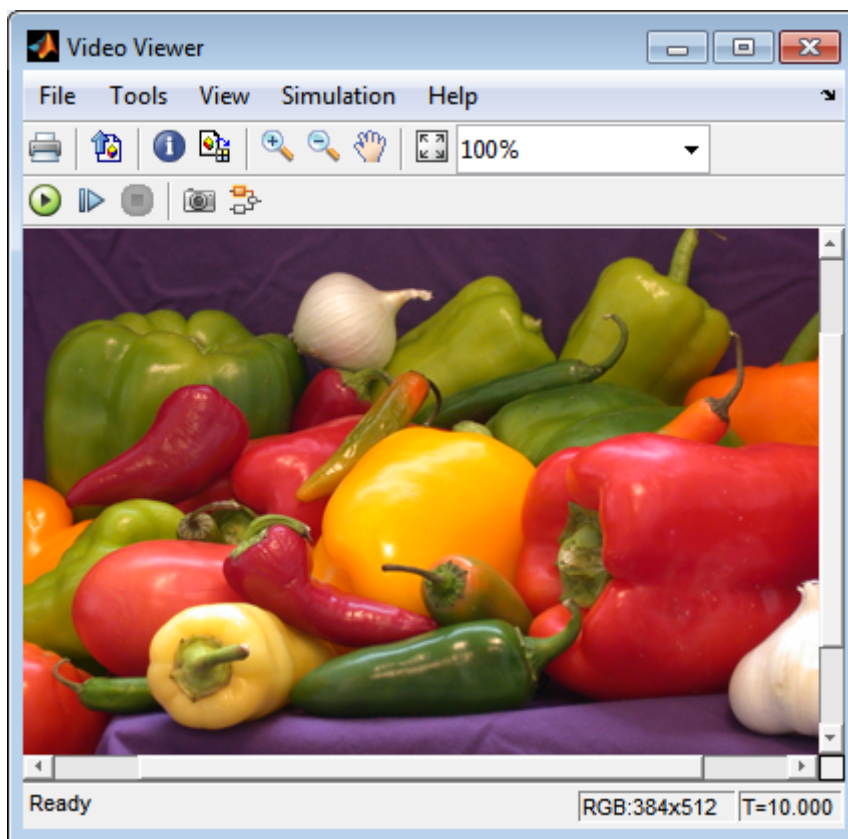


8 Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- Solver pane, **Stop time** = 0

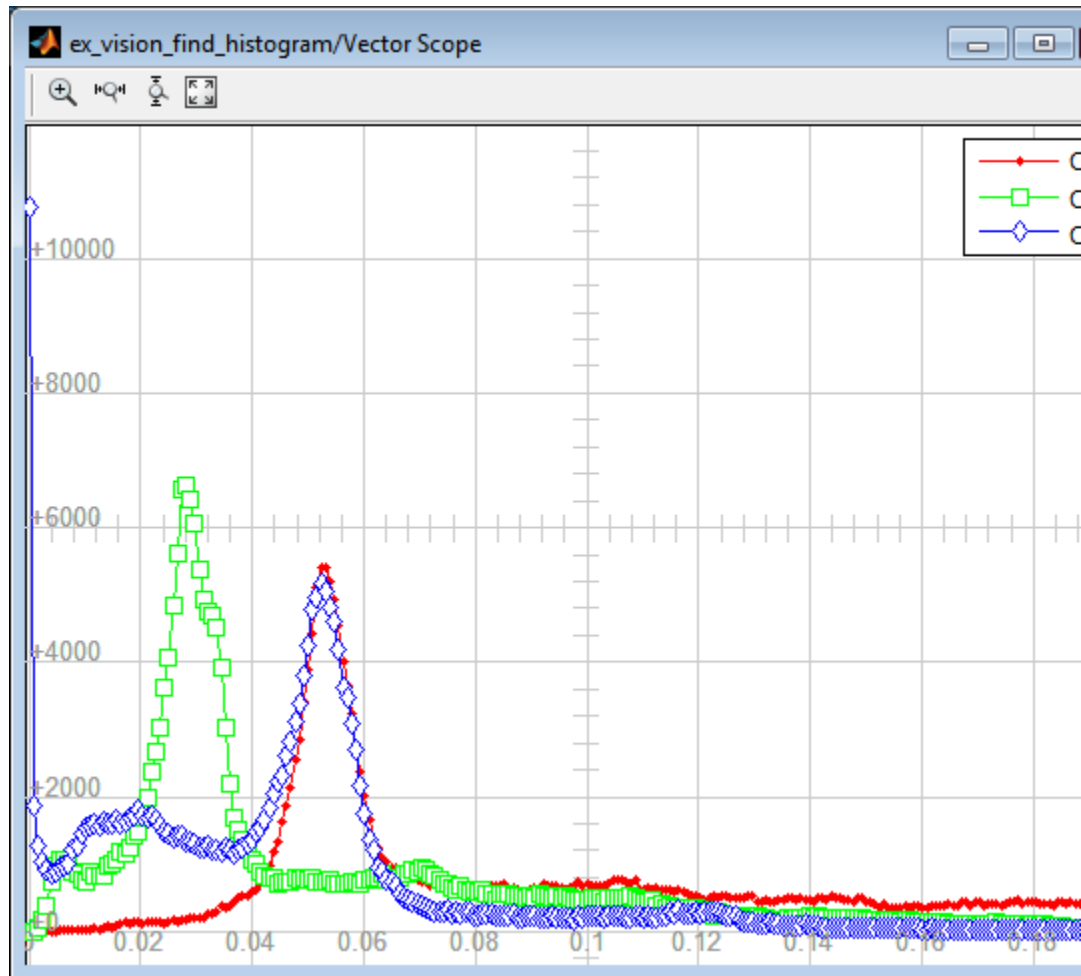
- Solver pane, **Type** = Fixed-step
 - Solver pane, **Solver** = Discrete (no continuous states)
- 9 Run the model using either the simulation button, or by selecting Simulation > Start.

The original image appears in the Video Viewer window.



- 10 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 2-D Histogram block to calculate the histogram of the R, G, and B values in an RGB image. To open a 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:

You can open the example model by typing

```
ex_vision_correct_nonuniform
```

on the MATLAB command line.

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

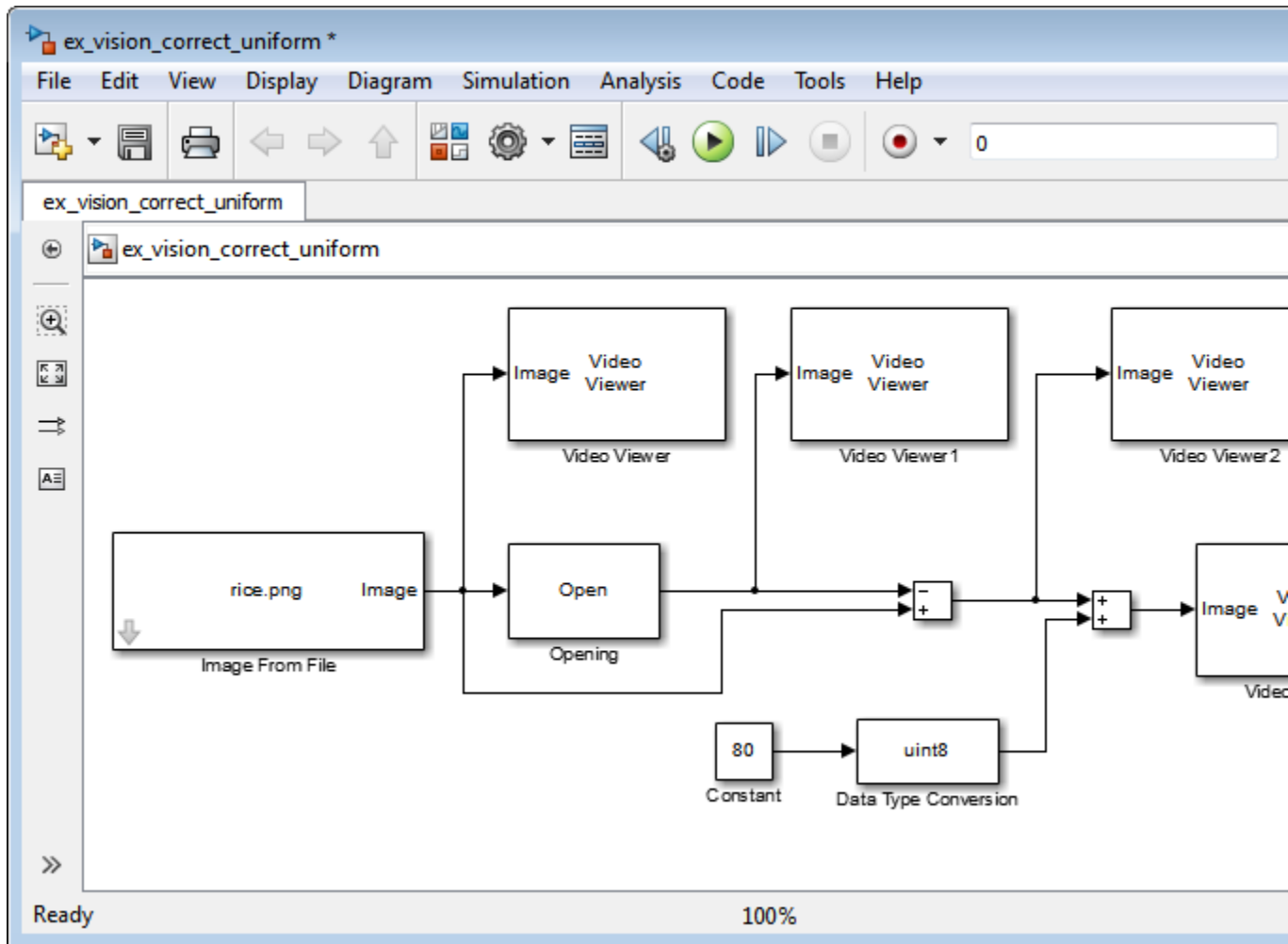
Block	Library	Quantity
Image From File	Computer Vision System Toolbox > Sources	1
Opening	Computer Vision System Toolbox > Morphological Operations	1
Video Viewer	Computer Vision System Toolbox > Sinks	4
Constant	Simulink > Sources	1
Sum	Simulink > Math Operations	2
Data Type Conversion	Simulink > Signal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

- 2 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.
- 3 Use the Video Viewer block to view the original image. Accept the default parameters.

- 4 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.

- 5 Use the Video Viewer1 block to view the background estimated by the Opening block. Accept the default parameters.
- 6 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** = -+
- 7 Use the Video Viewer2 block to view the result of subtracting the background from the original image. Accept the default parameters.
- 8 Use the Constant block to define an offset value. Set the **Constant value** parameter to 80.
- 9 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`.
- 10 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** = ++
- 11 Use the Video Viewer3 block to view the corrected image. Accept the default parameters.
- 12 Connect the blocks as shown in the following figure.

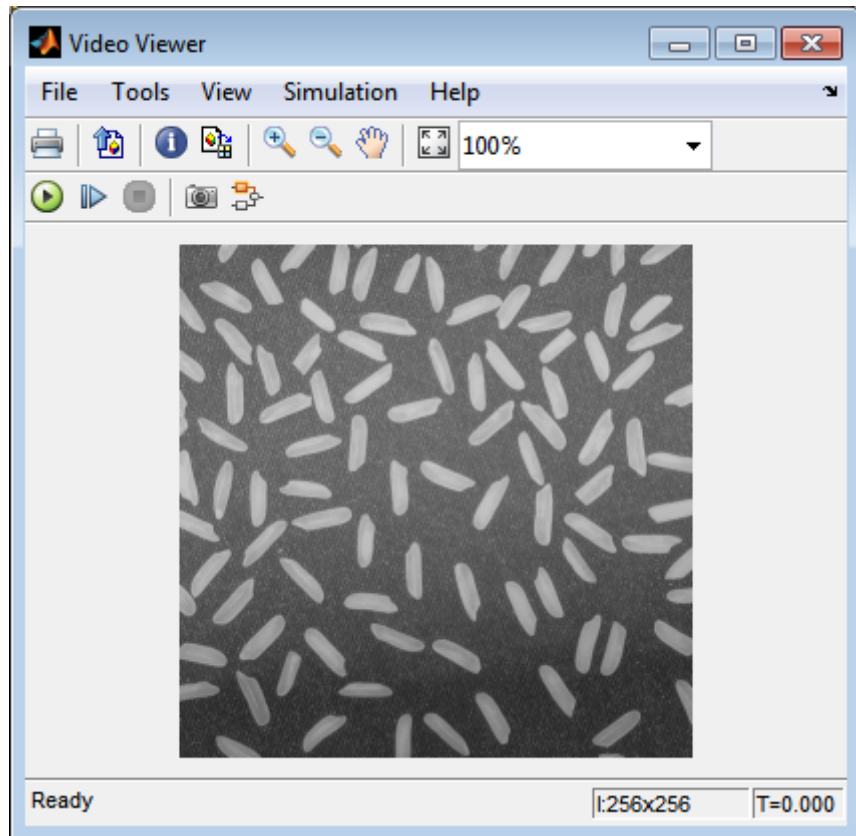


13 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

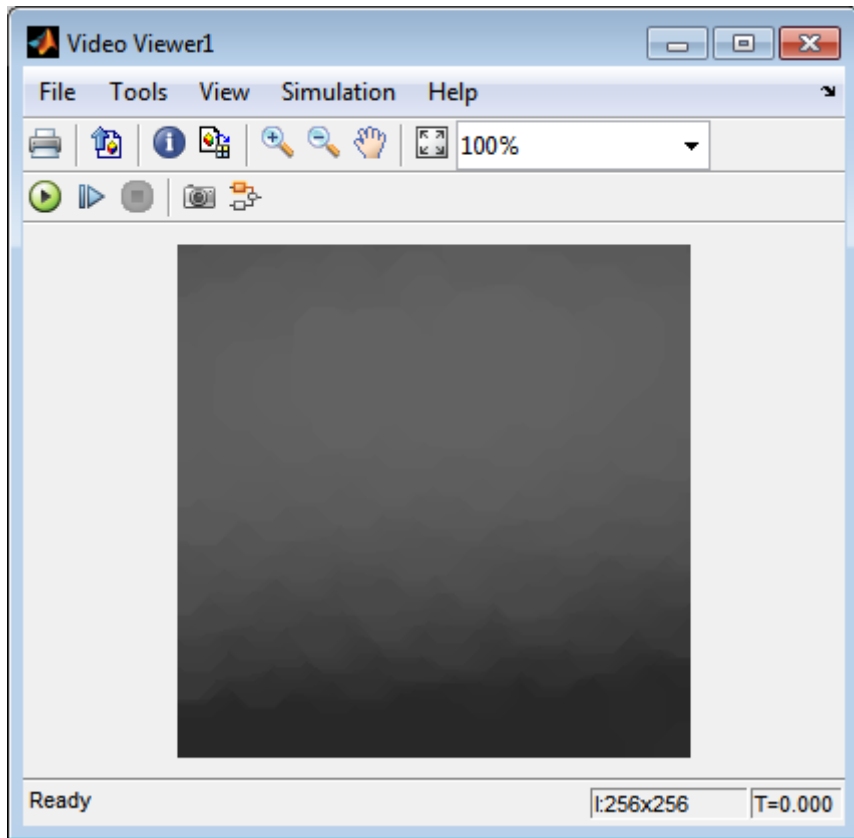
- **Solver** pane, **Stop time** = 0
- **Solver** pane, **Type** = Fixed-step
- **Solver** pane, **Solver** = discrete (no continuous states)

14 Run the model.

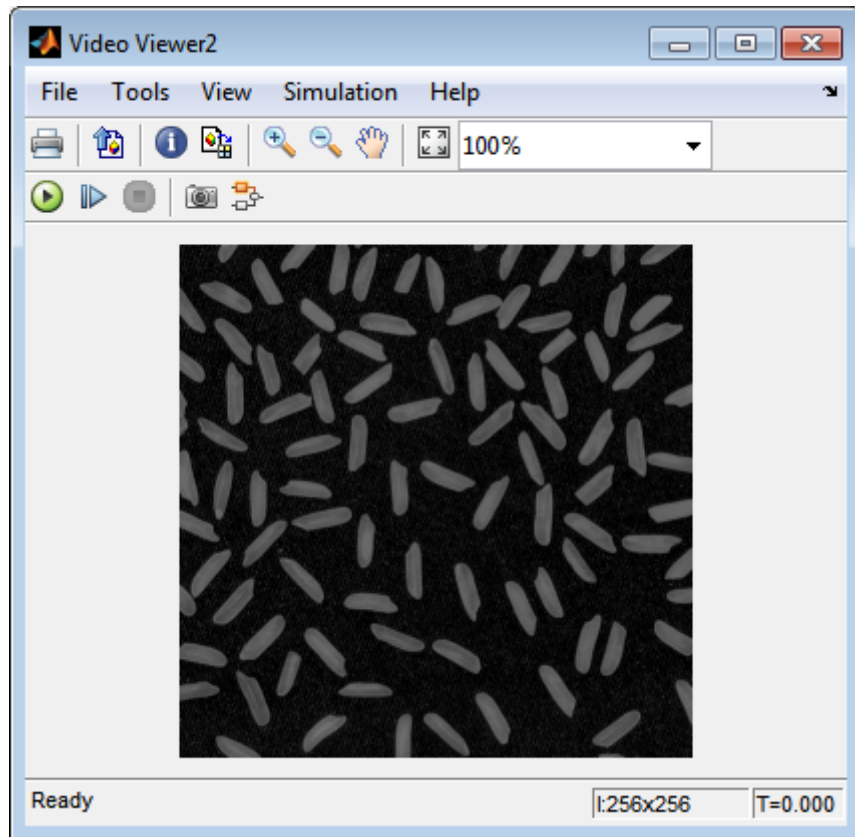
The original image appears in the Video Viewer window.



The estimated background appears in the Video Viewer1 window.

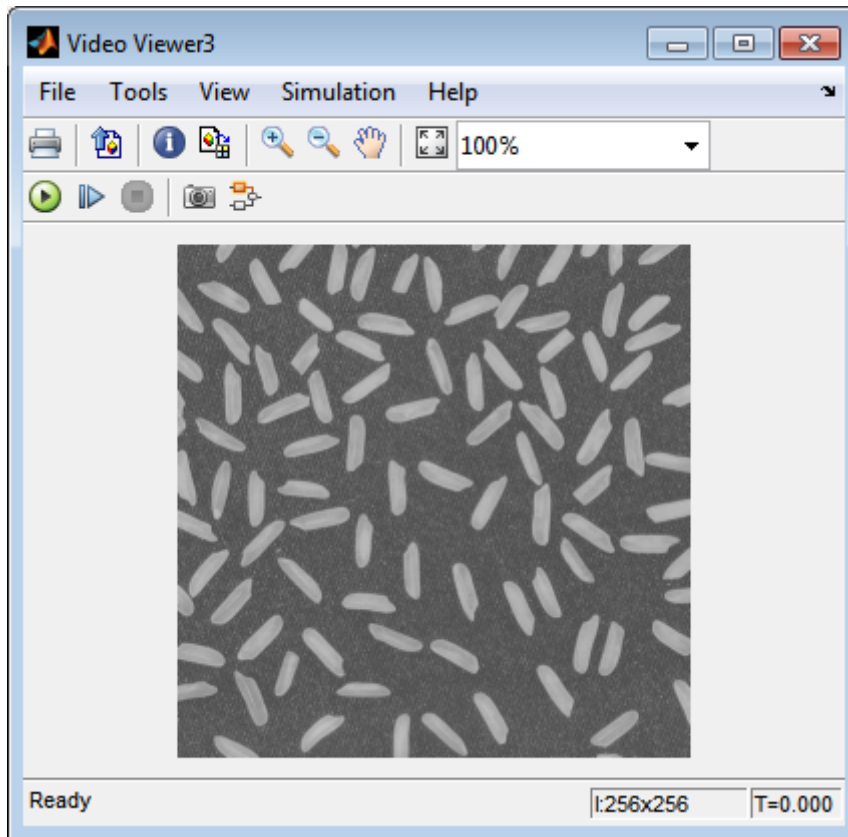


The image without the estimated background appears in the Video Viewer2 window.



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.

Note Running this example requires a DSP System Toolbox license.

You can open the example model by typing

```
ex_vision_count_objects
```

on the MATLAB command line.

- 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

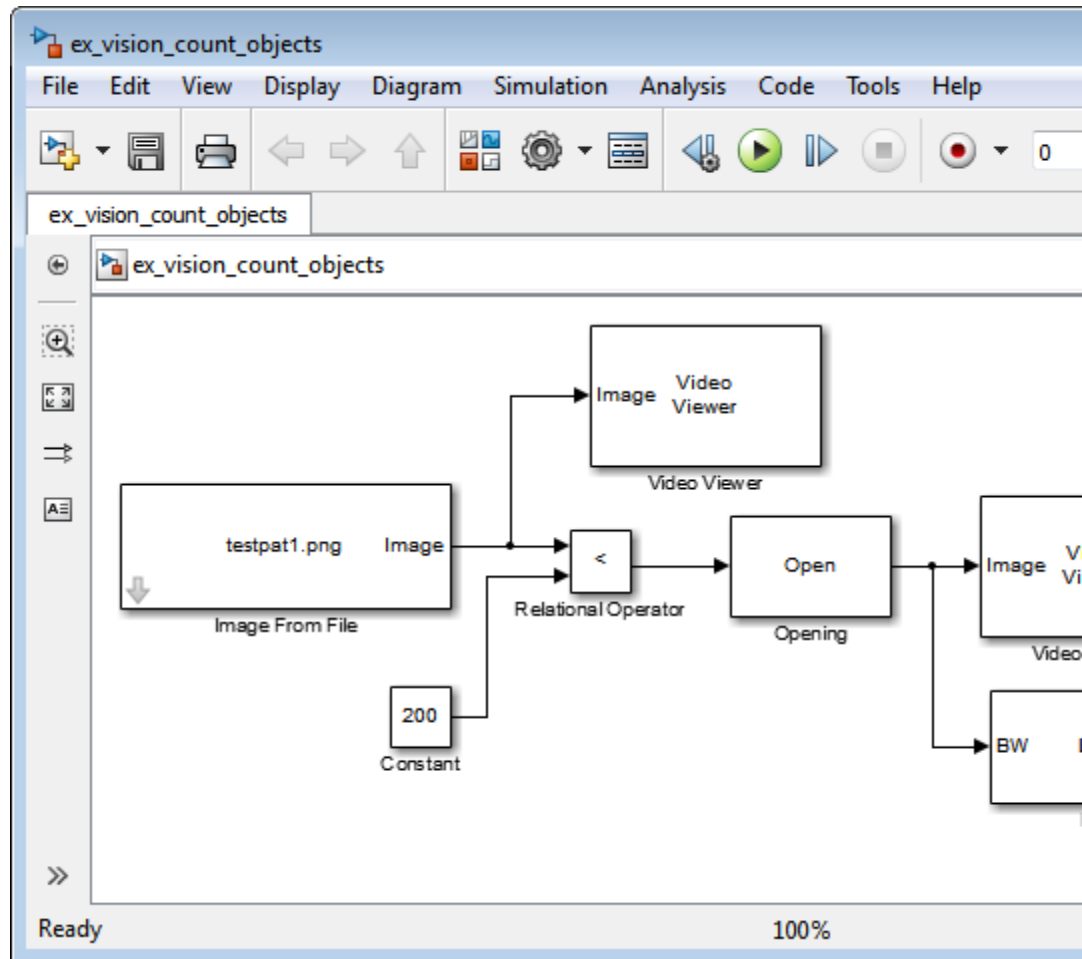
- 2 Use the Image From File block to import your image. Set the **File name** parameter to `testpat1.png`. This is a 256-by-256 matrix image of 8-bit unsigned integers.
- 3 Use the Constant block to define a threshold value for the Relational Operator block. Set the **Constant value** parameter to 200.
- 4 Use the Video Viewer block to view the original image. Accept the default parameters.
- 5 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.

- 6 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.

- 7 Use the Video Viewer1 block to view the opened image. Accept the default parameters.
- 8 Use the Label block to count the number of spokes in the input image. Set the **Output** parameter to `Number of labels`.
- 9 The Display block displays the number of spokes in the input image. Use the default parameters.
- 10 Connect the block as shown in the following figure.

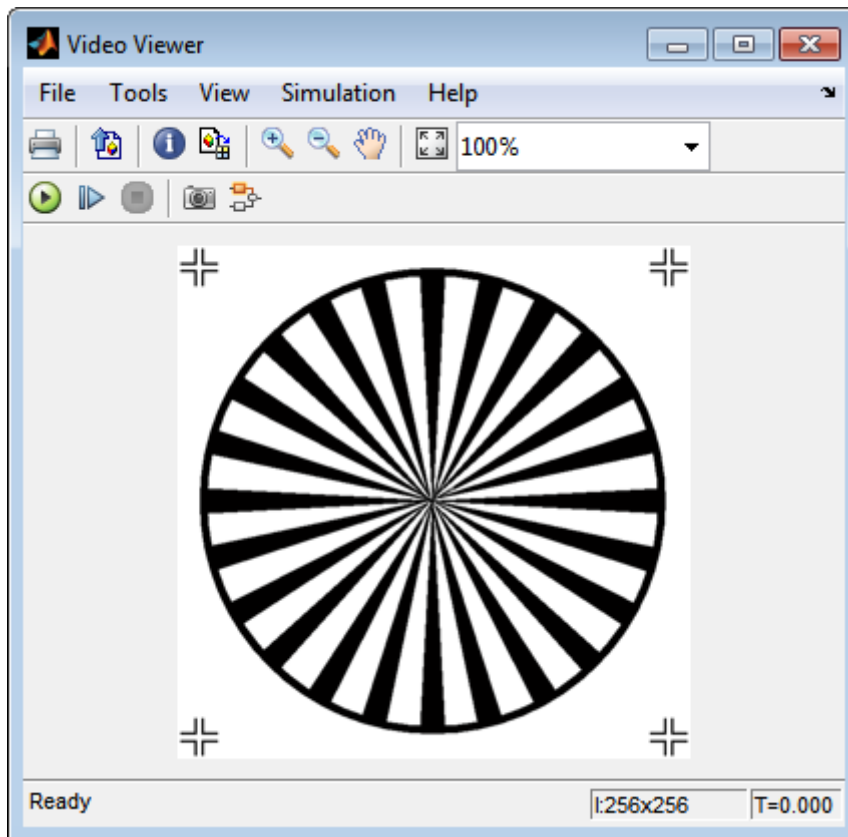


11 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

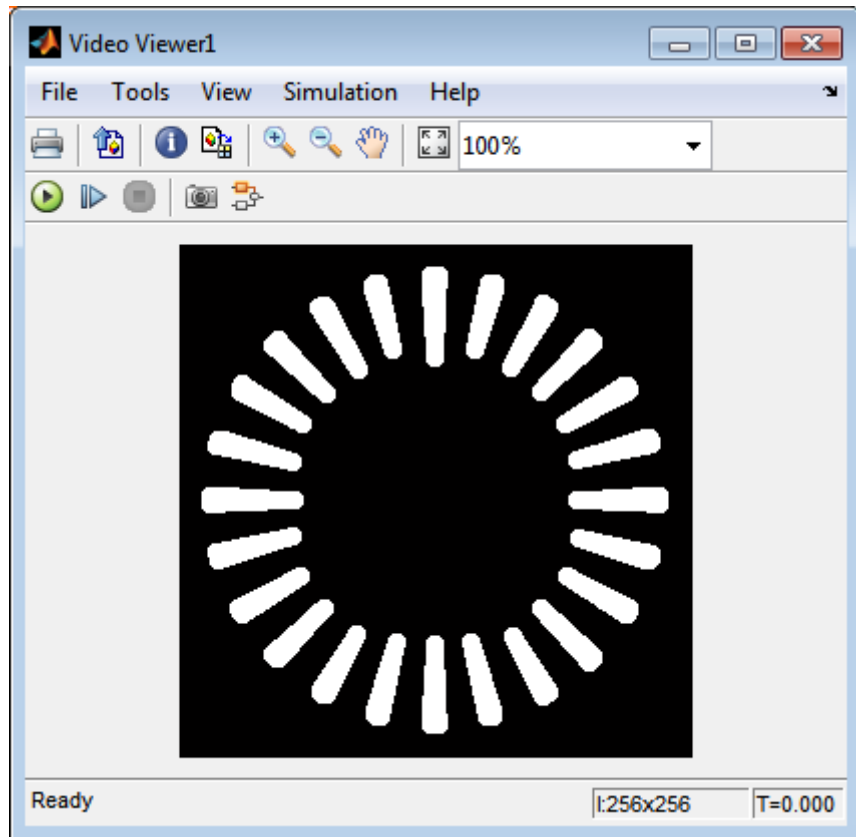
- **Solver** pane, **Stop time** = 0
- **Solver** pane, **Type** = Fixed-step
- **Solver** pane, **Solver** = discrete (no continuous states)

12 Run the model.

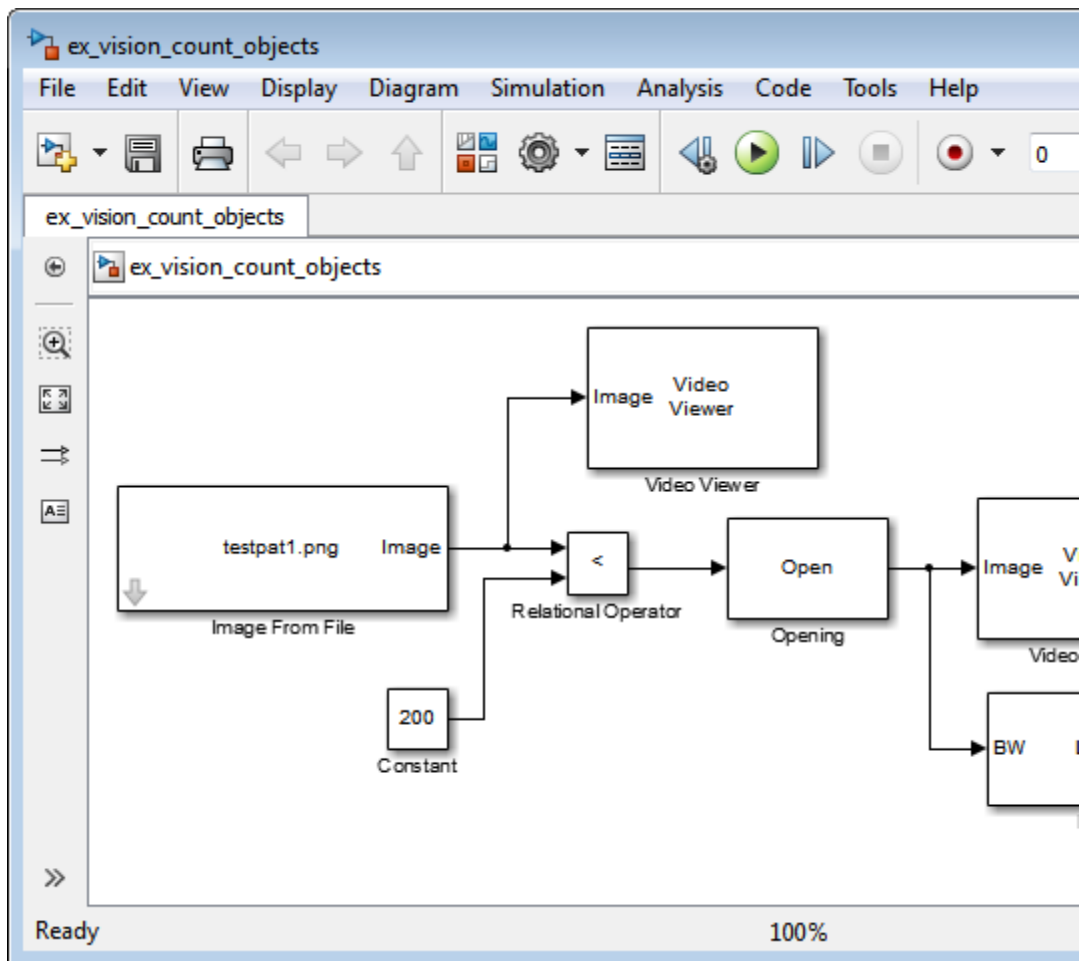
The original image appears in the Video Viewer1 window. To view the image at its true size, right-click the window and select **Set Display To True Size**.



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.

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.

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™” documentation.

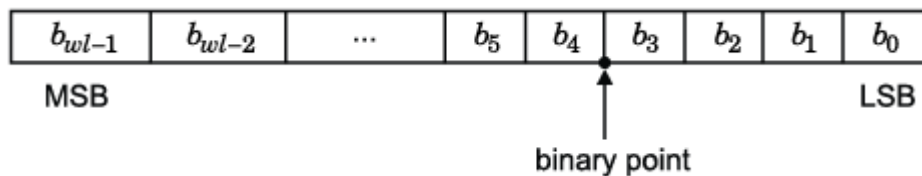
Fixed-Point Data Types

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 i^{th} 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

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

where the slope can be expressed as

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

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:

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

or

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

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.

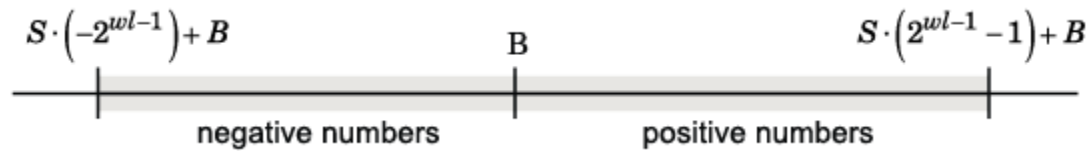
All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

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

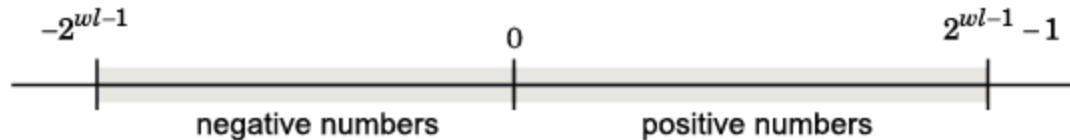
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl , scaling S , and bias B is illustrated below:



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 $2wl^{-1}$:

For slope = 1 and bias = 0:



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

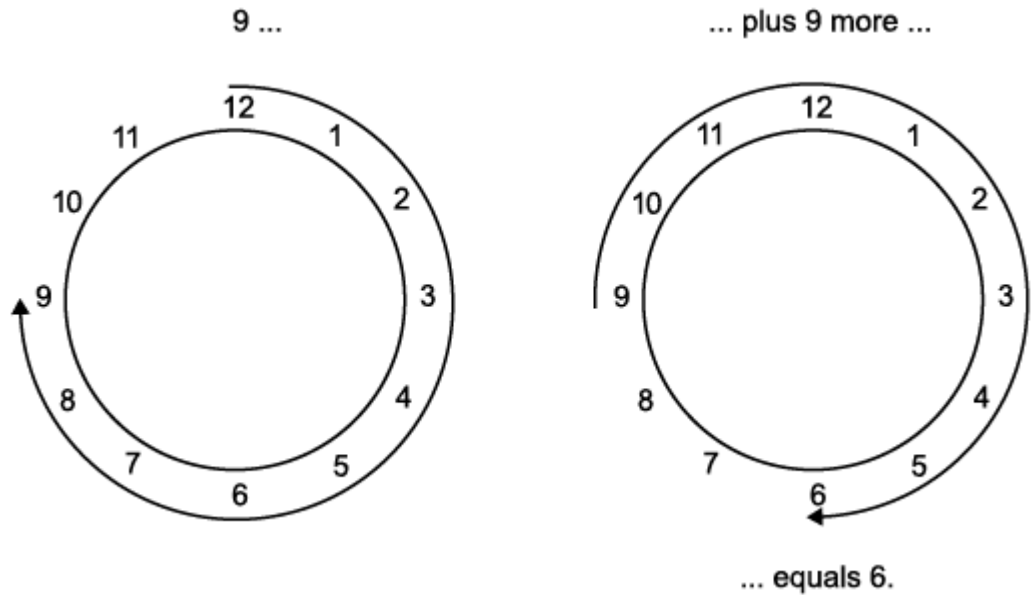
In this section...
“Modulo Arithmetic” on page 8-10
“Two’s Complement” on page 8-11
“Addition and Subtraction” on page 8-12
“Multiplication” on page 8-13
“Casts” on page 8-16

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:

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

Addition and Subtraction

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

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

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

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):

$$\begin{array}{r}
 010010.100 \text{ (18.5)} \\
 - 0110.110 \text{ (6.75)} \\
 \hline
 \end{array}
 \xrightarrow[\text{and sign extension}]{\text{two's complement}}
 \begin{array}{r}
 010010.100 \text{ (18.5)} \\
 +111001.010 \text{ (-6.75)} \\
 \hline
 1001011.110 \text{ (11.75)}
 \end{array}$$

Carry bit is discarded.

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):

$$\begin{array}{r}
 10.11 \text{ (-1.25)} \\
 \quad 011 \text{ (3)} \\
 \hline
 11011 \\
 \quad 1011 \\
 \hline
 1100.01 \text{ (-3.75)}
 \end{array}$$

The extra 1 is the result of necessary sign extension.

The number of fractional bits of the result is the sum of the number of fractional bits of the factors.

Multiplication Data Types

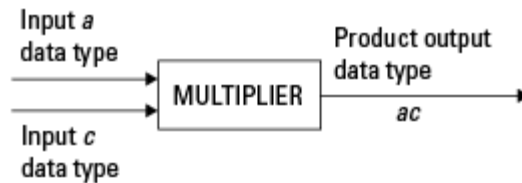
The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex

multiplication. See individual reference pages to determine whether a particular block accepts complex fixed-point inputs.

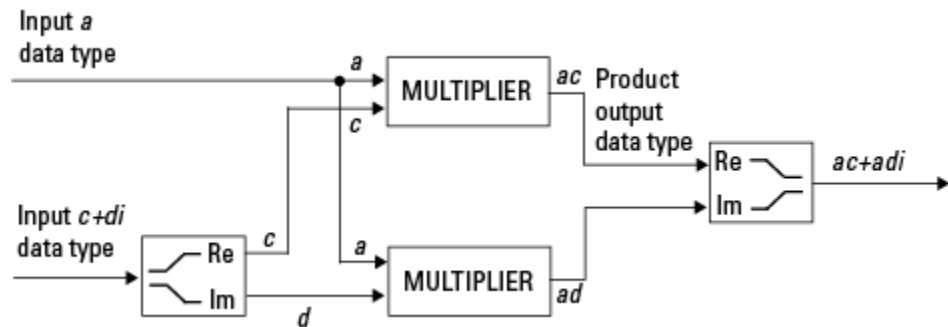
In most cases, you can set the data types used during multiplication in the block mask. See Accumulator Parameters, Intermediate Product Parameters, Product Output Parameters, and Output Parameters. These data types are defined in “Casts” on page 8-16.

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.

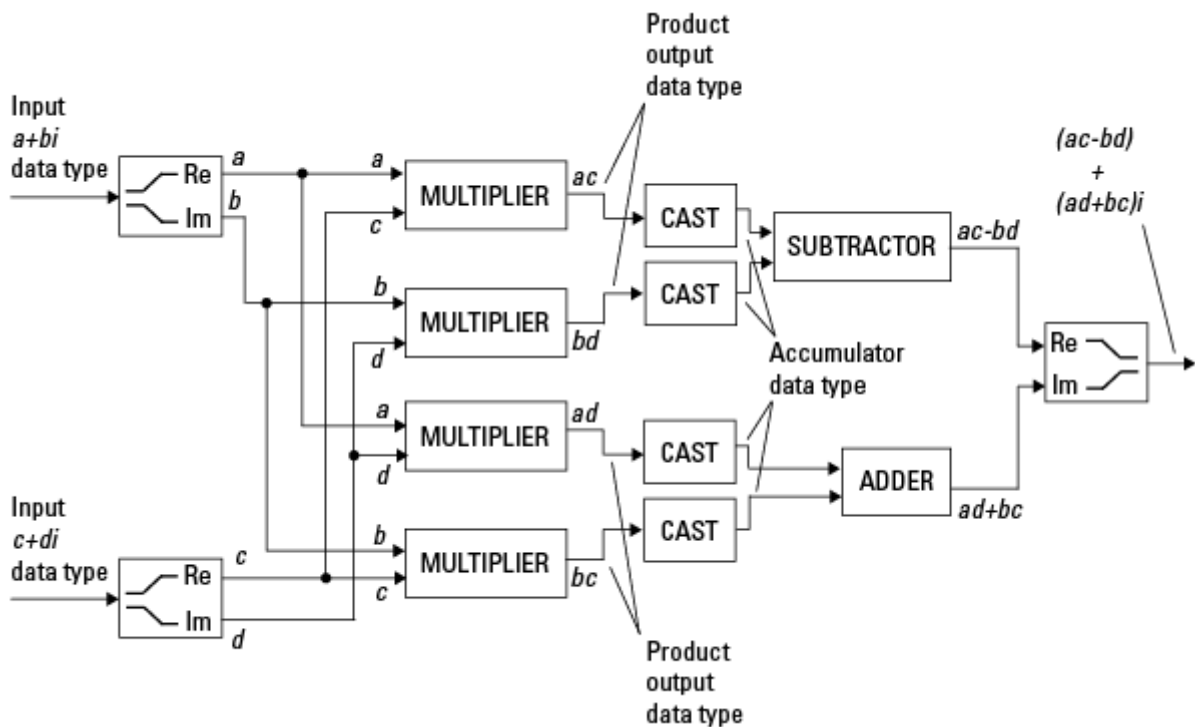
Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in System Toolbox software. The software returns the output of this operation in the product output data type, as the next figure shows.



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

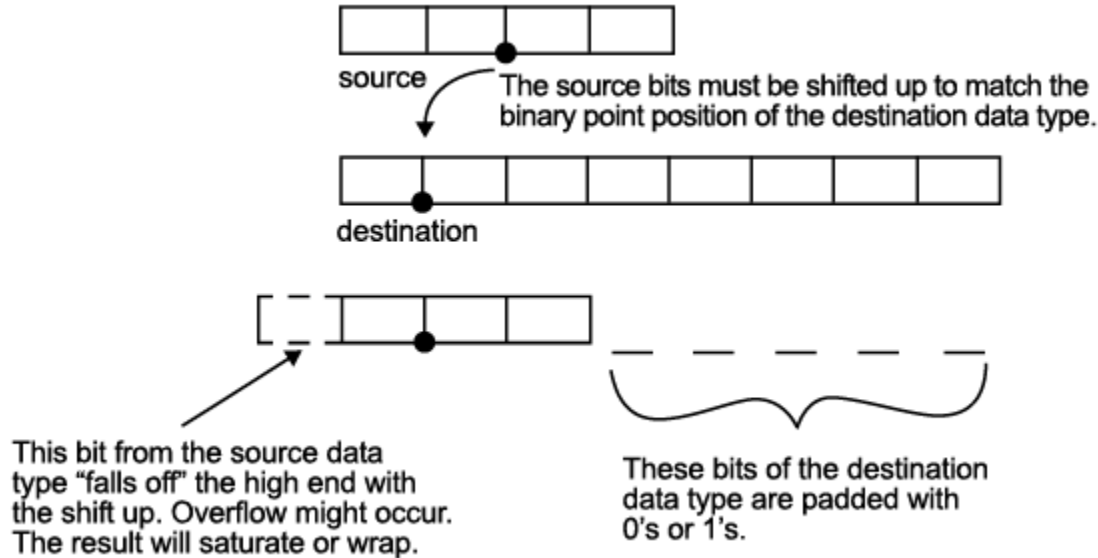
Many fixed-point System Toolbox blocks allow you to specify the data type and scaling of the block output on the mask. Remember that the software does not allow mixed types on the input and output ports of its blocks. Therefore, if you would like to specify a fixed-point output data type and scaling 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:



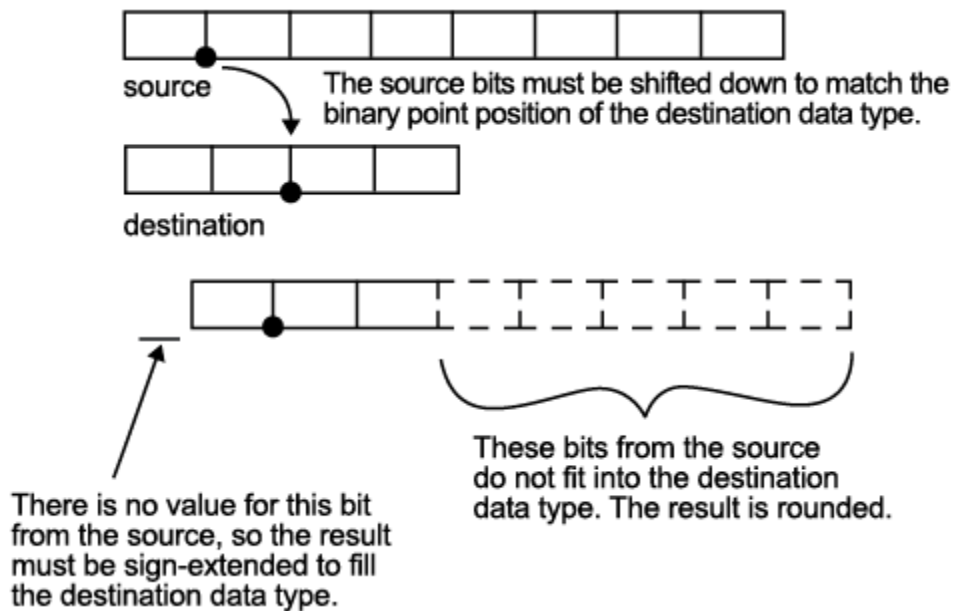
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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 the “Fixed-Point” topic.

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.Autothresher  
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
```

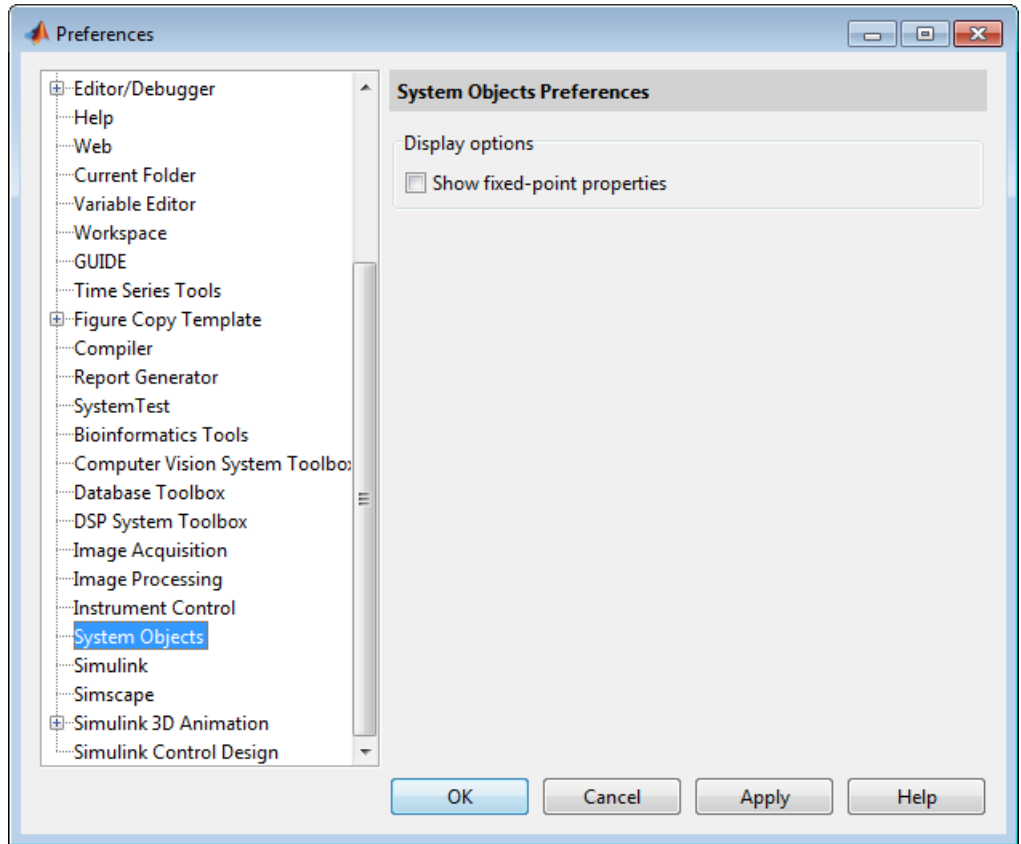
Displaying Fixed-Point Properties

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 the **Preferences** icon from

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.

For any System object provided in the Toolbox, 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

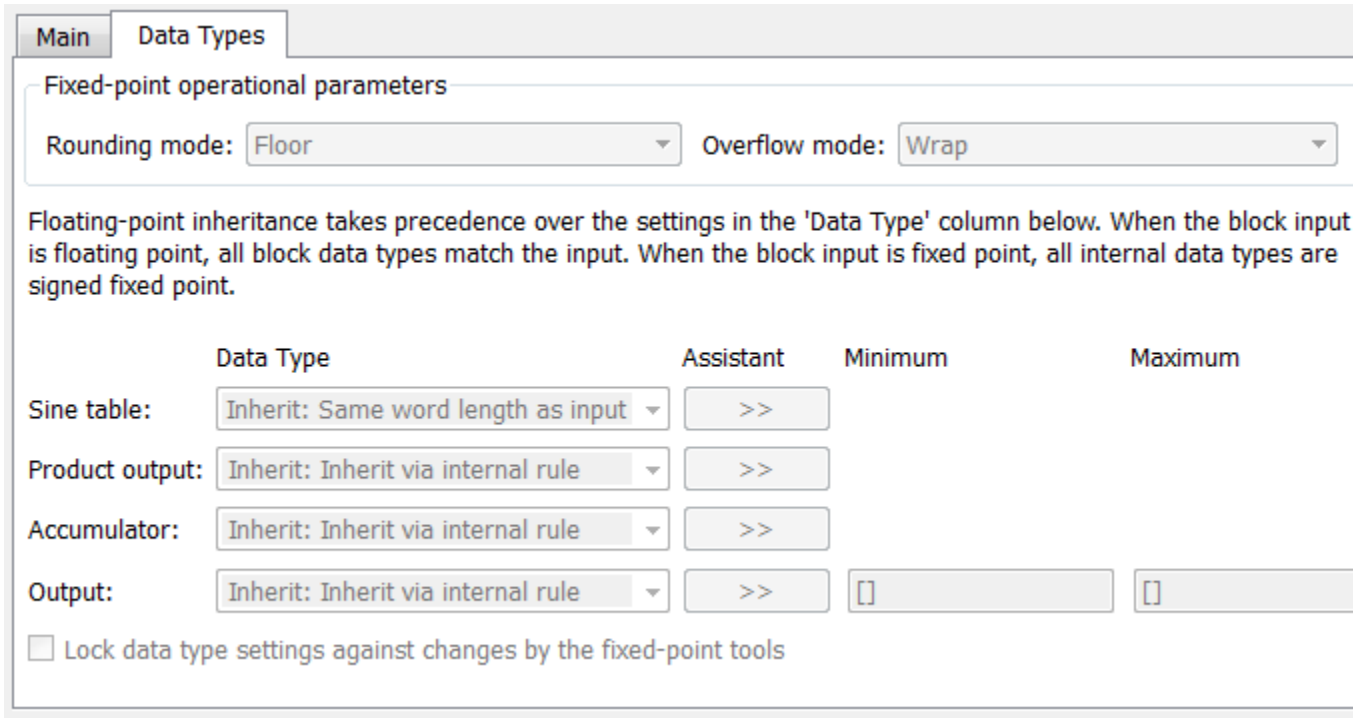
In this section...
“Fixed-Point Block Parameters” on page 8-25
“Specify System-Level Settings” on page 8-28
“Inherit via Internal Rule” on page 8-29
“Specify Data Types for Fixed-Point Blocks” on page 8-40

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.

Fixed-Point Data Type Parameter	Description
	See “Overflow Handling” on page 8-7 for more information on the available options.
Intermediate Product	<p>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.</p> <p>See the reference page of a specific block to learn about the intermediate product data type for that block.</p>
Product Output	<p>Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results.</p> <p>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.</p>
Accumulator	<p>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).</p> <p>See the reference page of a specific block for details on the accumulator data type of that block.</p>
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:

“Specify Data Types Using 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 `fxptd1g` reference page — A reference page on the Fixed-Point Tool in the Simulink documentation
- “Overview of the 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 8-31
- “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 8-31
- “Internal Rule Examples” on page 8-33

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.

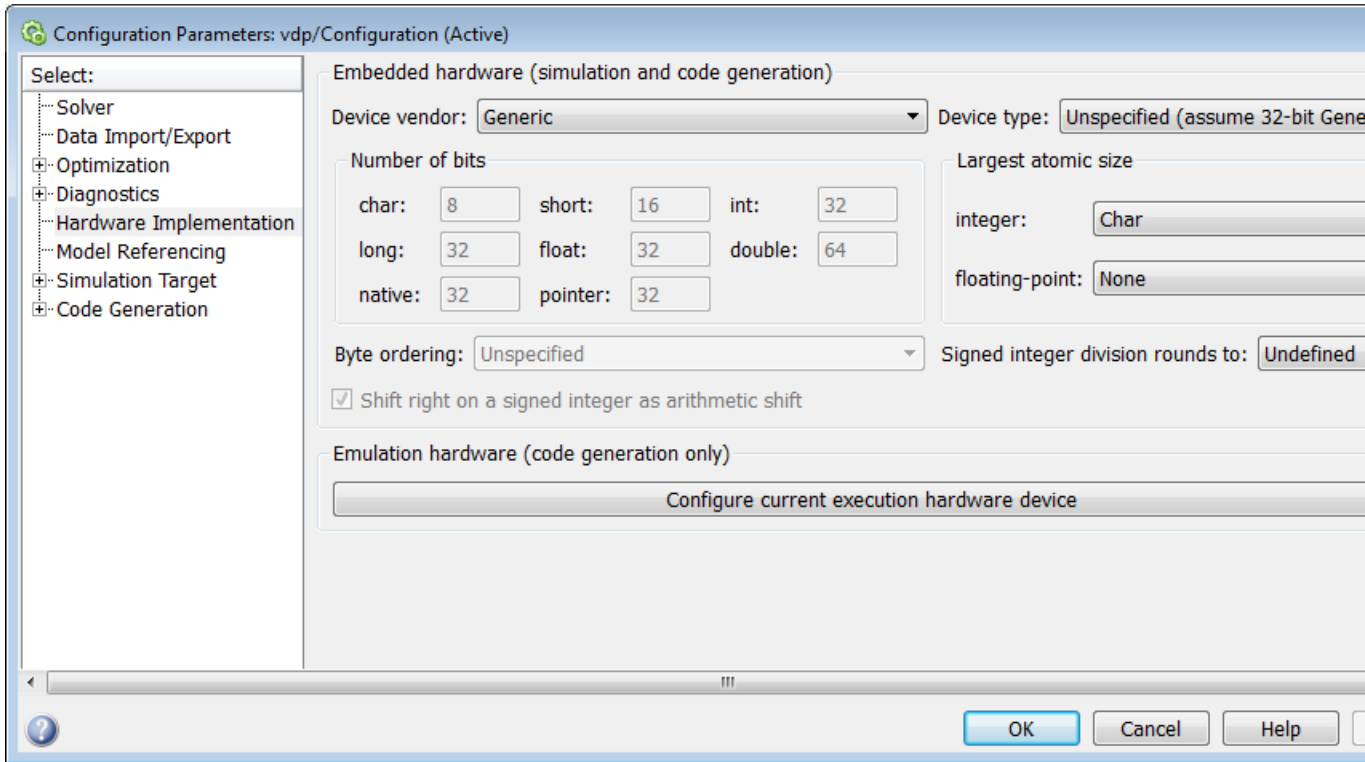
Internal Rule for Output Data Types

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.

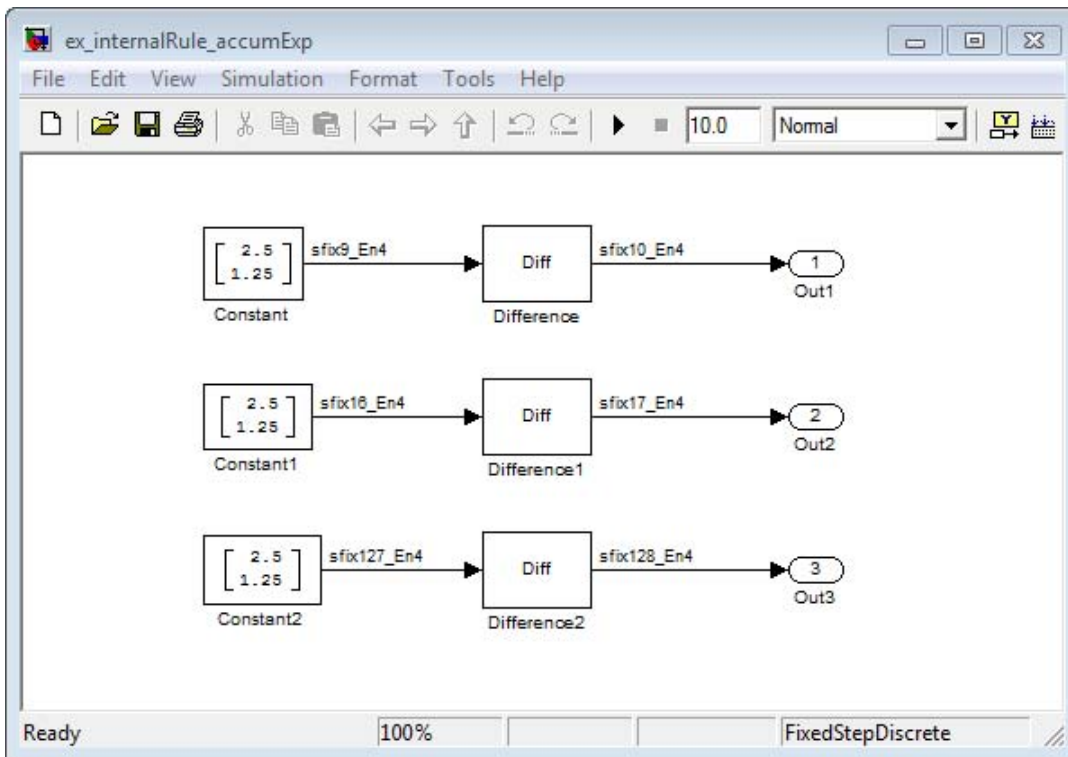
Other targets. For all targets other than ASIC/FPGA, the ideal, full-precision word length calculated by the internal rule is rounded up to the next available word length of the target. The calculated ideal fraction length is used, keeping the least-significant bits.

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:

$$WL_{ideal\ accumulator} = WL_{input\ to\ accumulator} + \text{floor}(\log_2(\text{number of accumulations})) + 1$$

$$WL_{ideal\ accumulator} = 9 + \text{floor}(\log_2(1)) + 1$$

$$WL_{ideal\ accumulator} = 9 + 0 + 1 = 10$$

$$WL_{ideal\ accumulator1} = WL_{input\ to\ accumulator1} + \text{floor}(\log_2(\text{number of accumulations})) + 1$$

$$WL_{ideal\ accumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$

$$WL_{ideal\ accumulator1} = 16 + 0 + 1 = 17$$

$$WL_{ideal\ accumulator2} = WL_{input\ to\ accumulator2} + \text{floor}(\log_2(\text{number of accumulations})) + 1$$

$$WL_{ideal\ accumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$

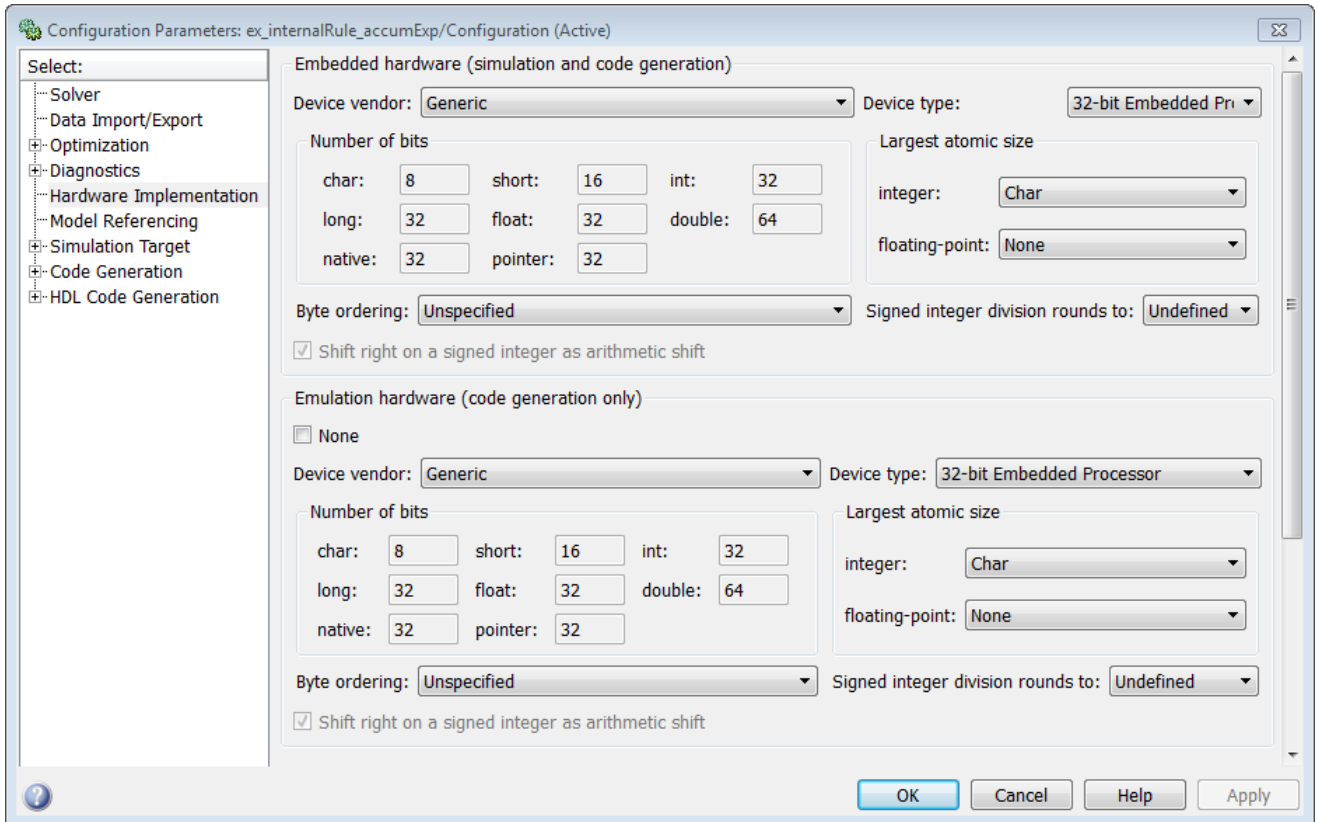
$$WL_{ideal\ accumulator2} = 127 + 0 + 1 = 128$$

Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

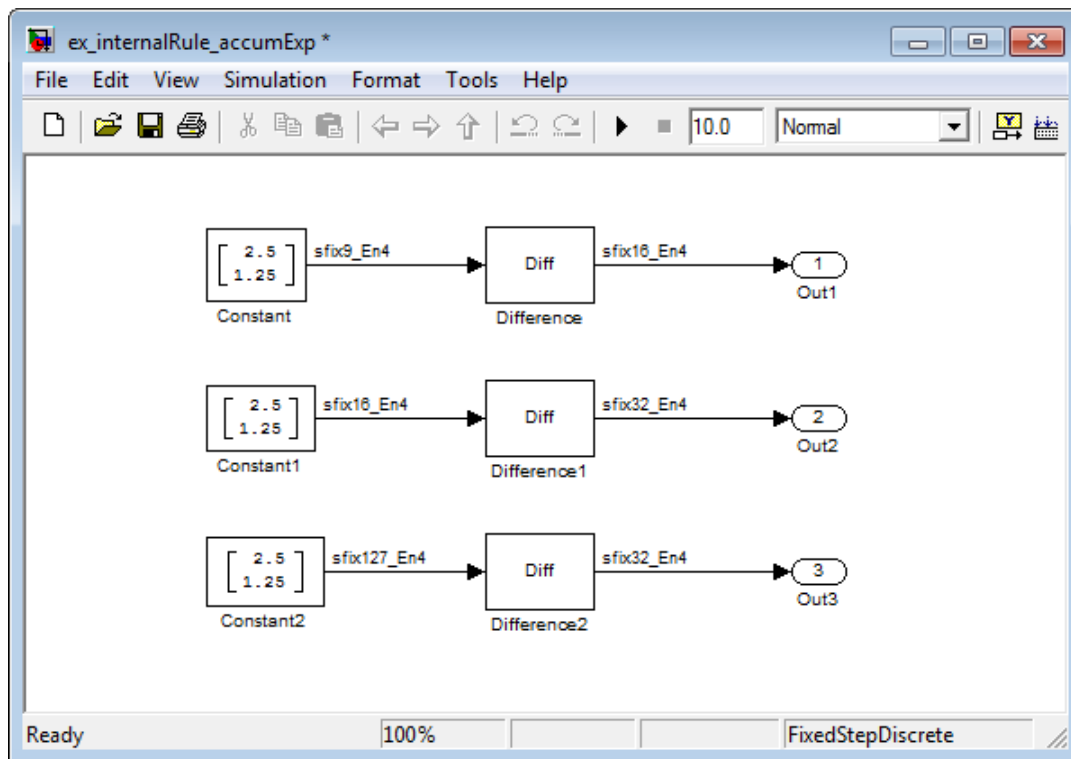
$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$

$$FL_{ideal\ accumulator} = 4$$

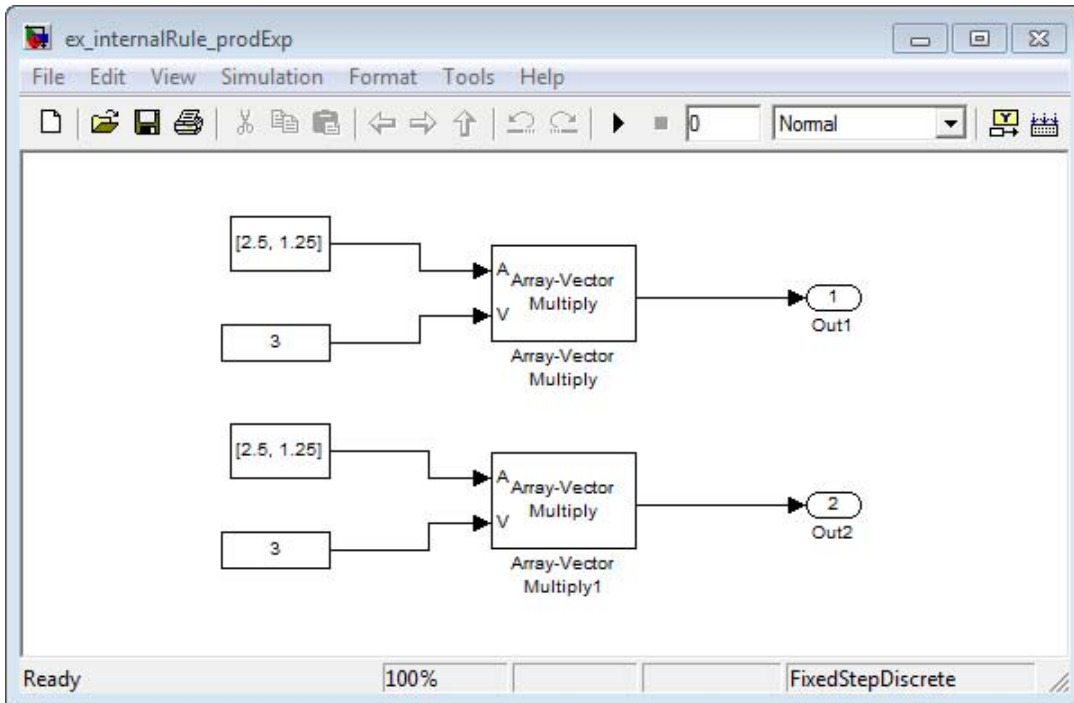
Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to 32 bit Embedded Processor, by changing the parameters as shown in the following figure.



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.



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:

$$WL_{ideal\ product} = WL_{input\ a} + WL_{input\ b}$$

$$WL_{ideal\ product} = 7 + 5 = 12$$

$$WL_{ideal\ product1} = WL_{input\ a} + WL_{input\ b}$$

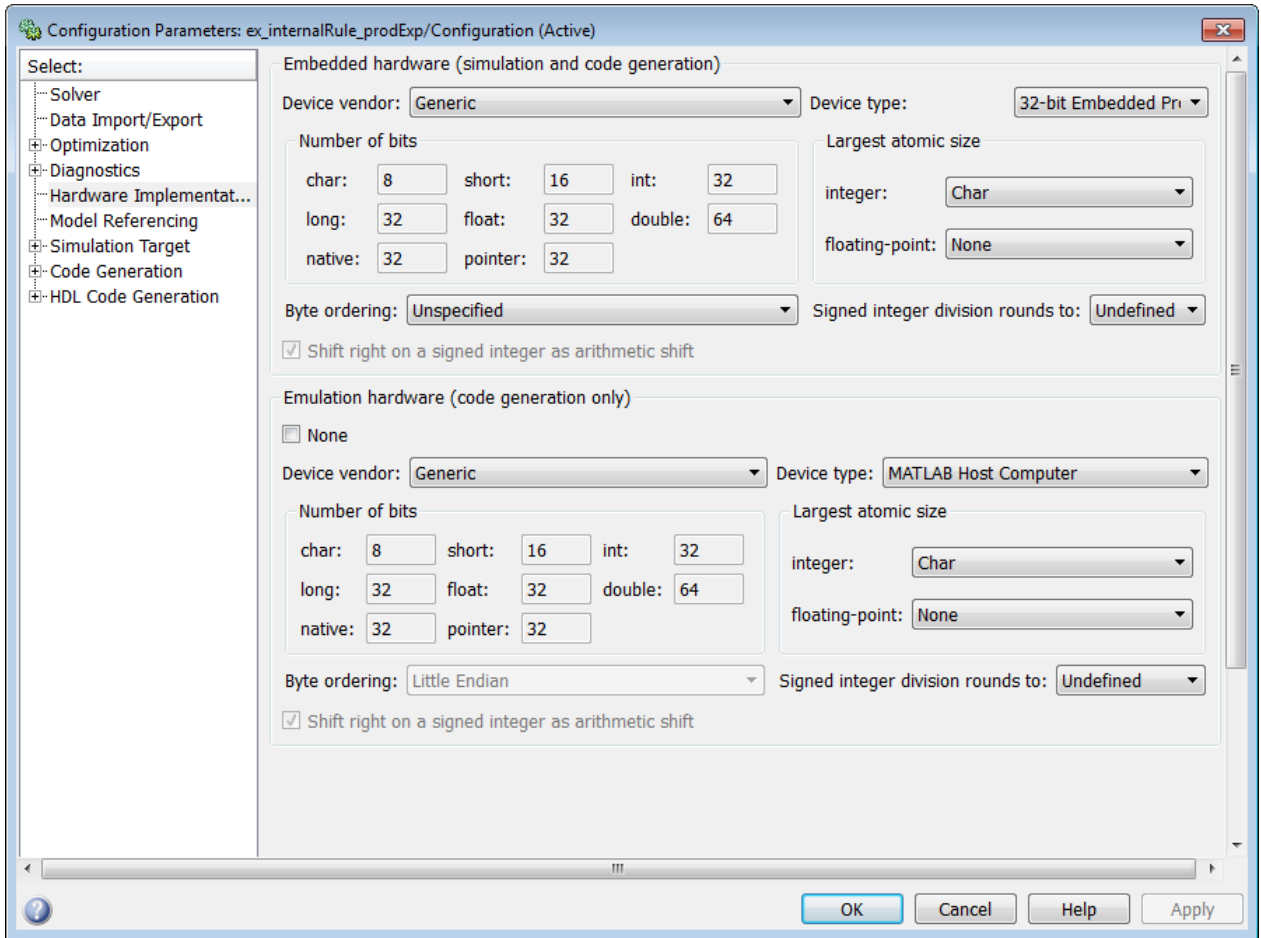
$$WL_{ideal\ product1} = 16 + 15 = 31$$

Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

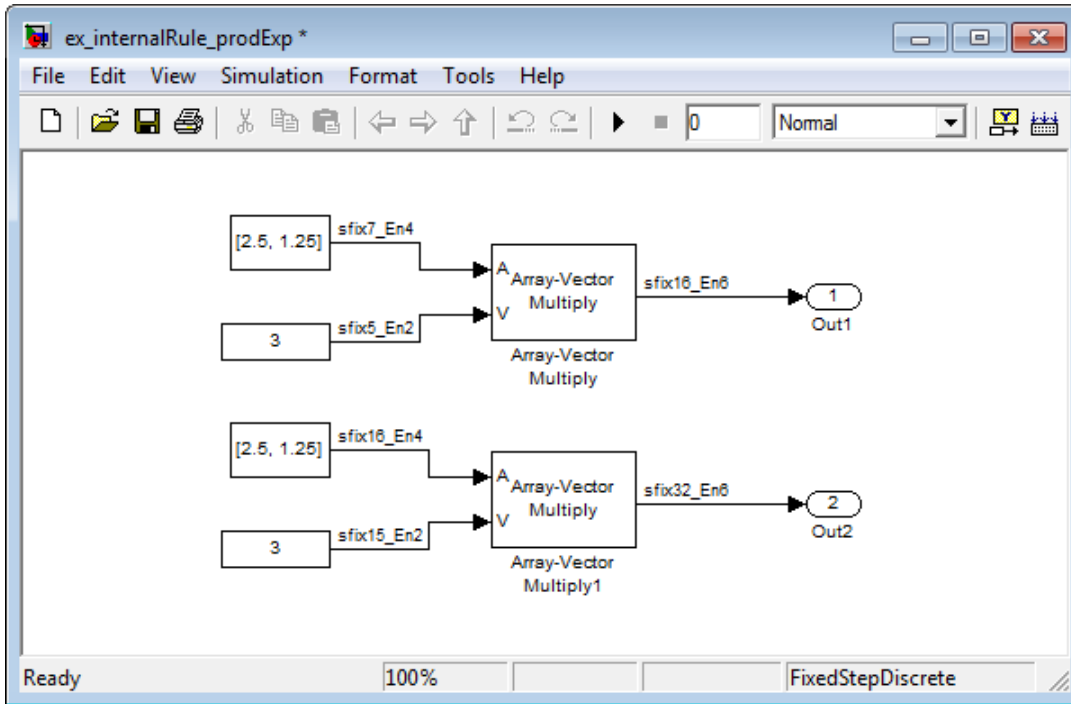
$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$

$$FL_{ideal\ accumulator} = 4$$

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.



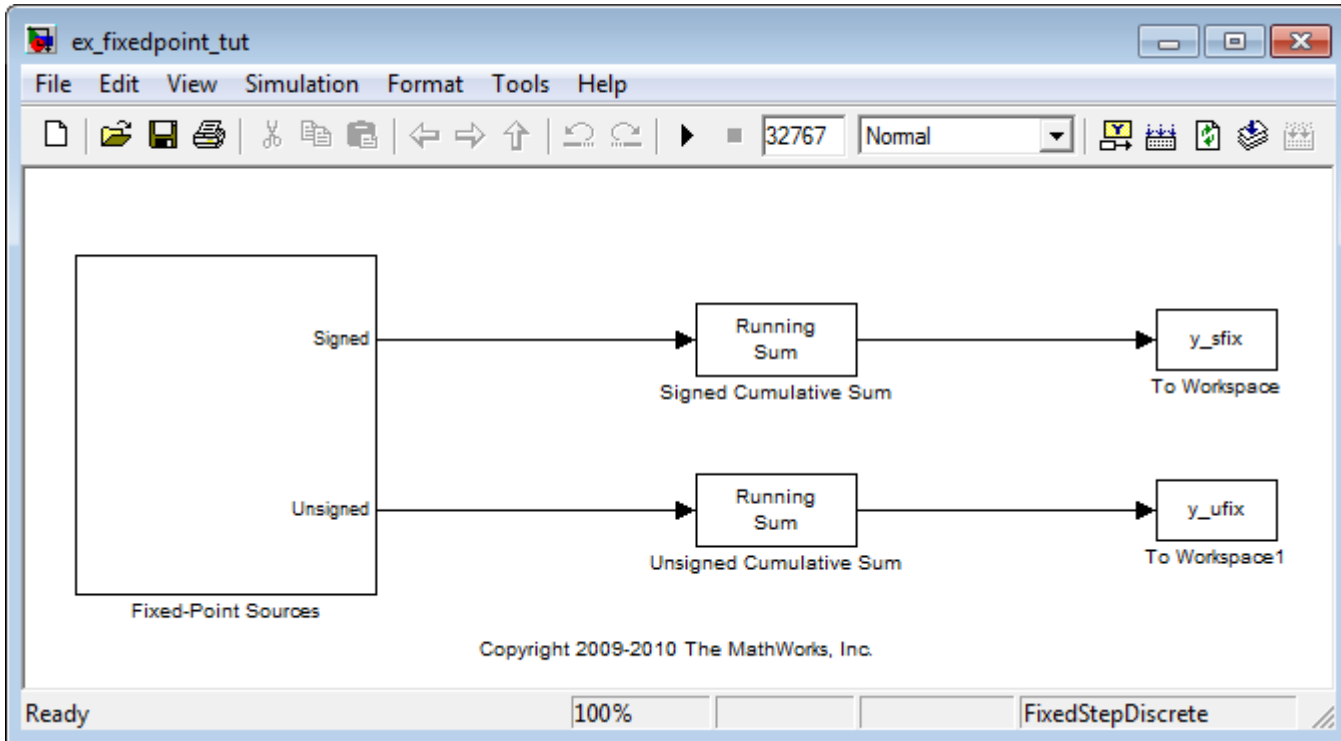
Specify Data Types for Fixed-Point Blocks

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 8-46
- “Use the Fixed-Point Tool to Propose Fraction Lengths” on page 8-47
- “Examine the Results and Accept the Proposed Scaling” on page 8-47

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.

- 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**.

The screenshot displays the Fixed-Point Tool interface. The title bar reads "Fixed-Point Tool". The menu bar includes "File", "Collect", "Autoscaling", "Results", "Run", "View", "Tools", and "Help". The toolbar contains icons for file operations, simulation control, and data viewing, along with a "Show:" dropdown set to "All results".

The "Model Hierarchy" pane on the left shows a tree structure:

- Fixed-Point Tool Root
 - ex_fixedpoint_tut
 - Fixed-Point Sources

The main workspace, titled "Contents of: ex_fixedpoint_tut", shows a "Column View" set to "Simulation View" with a "Show Details" link. Below this is a table with the following columns:

Name	Run	CompiledDT	SpecifiedDT	SimMin
------	-----	------------	-------------	--------

The right-hand side of the interface features a vertical toolbar with various icons and controls, including a "Fixed-Point" section with a "Use" button, a "Data" section with a "Use" button, and an "Autom" section with "DT" and "DT" buttons.

- **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.

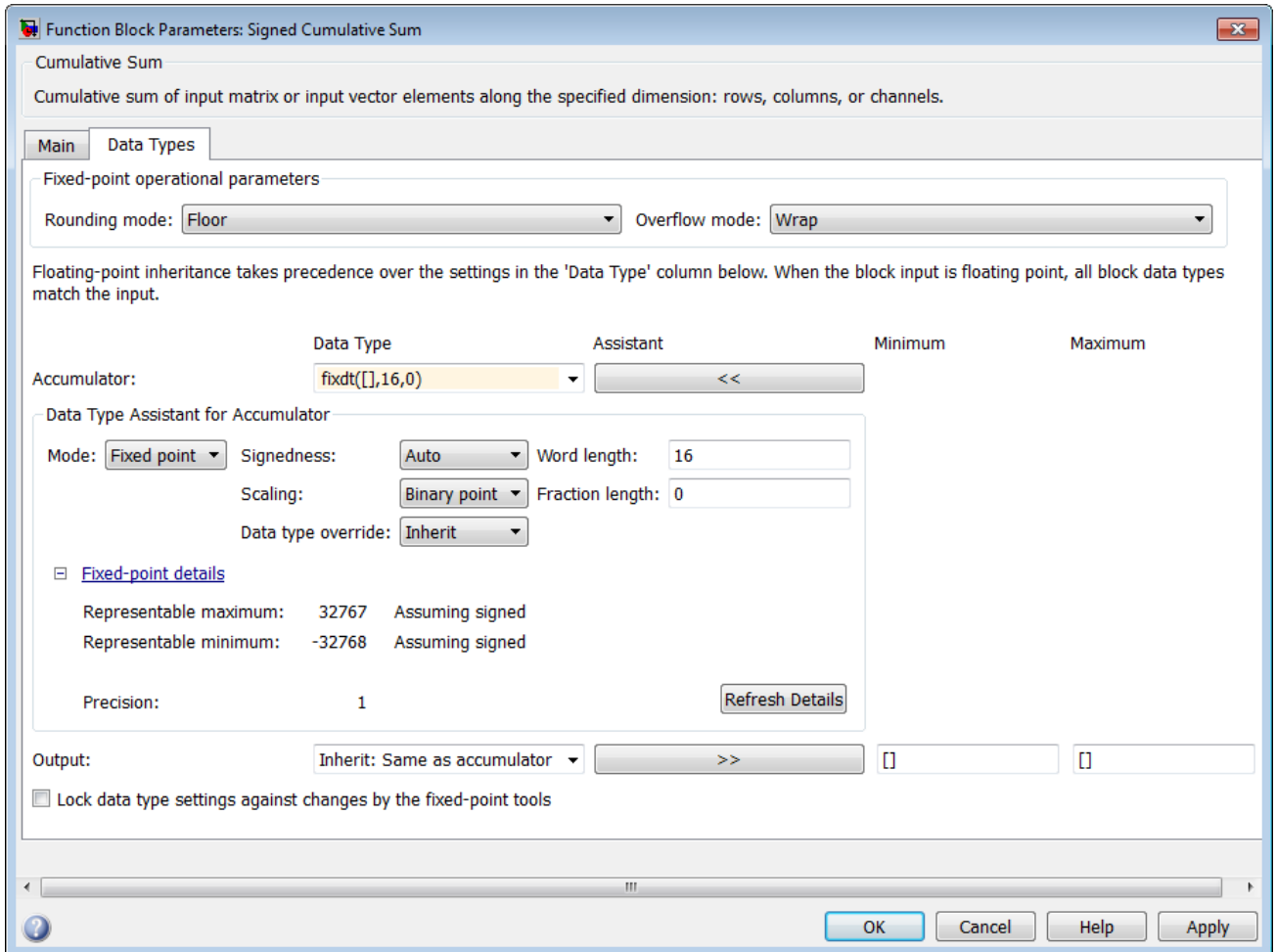
You can also see that the **SimMin** and **SimMax** values for the Accumulator data types range from 0 to .9997. The logged results indicate that 8,192 overflows wrapped during simulation in the Accumulator data type of the Signed Cumulative Sum block. Similarly, the Accumulator data type of the Unsigned 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.

- 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.


- 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.
- 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.

- 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:

- 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)`).

Code Generation

- “Code Generation with System Objects” on page 9-2
- “Functions and Classes that Generate Code” on page 9-7
- “Shared Library Dependencies” on page 9-9
- “Accelerating Simulink Models” on page 9-10

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 in MATLAB Code Generation” for more information.

Supported Computer Vision System Toolbox System Objects

Object	Description
Analysis & Enhancement	
<code>vision.BoundaryTracer</code>	Trace object boundaries in binary images
<code>vision.ContrastAdjuster</code>	Adjust image contrast by linear scaling
<code>vision.Deinterlacer</code>	Remove motion artifacts by deinterlacing input video signal
<code>vision.EdgeDetector</code>	Find edges of objects in images
<code>vision.ForegroundDetector</code>	Detect foreground using Gaussian Mixture Models. This object supports tunable properties in code generation.
<code>vision.HistogramEqualizer</code>	Enhance contrast of images using histogram equalization
<code>vision.TemplateMatcher</code>	Perform template matching by shifting template over image
Conversions	
<code>vision.Autothresher</code>	Convert intensity image to binary image
<code>vision.ChromaResampler</code>	Downsample or upsample chrominance components of images
<code>vision.ColorSpaceConverter</code>	Convert color information between color spaces
<code>vision.DemosaicInterpolator</code>	Demosaic Bayer’s format images
<code>vision.GammaCorrector</code>	Apply or remove gamma correction from images or video streams

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.ImageComplementer</code>	Compute complement of pixel values in binary, intensity, or RGB images
<code>vision.ImageDataTypeConverter</code>	Convert and scale input image to specified output data type
Feature Detection, Extraction, and Matching	
<code>vision.CornerDetector</code>	Corner metric matrix and corner detector. This object supports tunable properties in code generation.
Filtering	
<code>vision.Convolver</code>	Compute 2-D discrete convolution of two input matrices
<code>vision.ImageFilter</code>	Perform 2-D FIR filtering of input matrix
<code>vision.MedianFilter</code>	2D median filtering
Geometric Transformations	
<code>vision.GeometricRotator</code>	Rotate image by specified angle
<code>vision.GeometricScaler</code>	Enlarge or shrink image size
<code>vision.GeometricShearer</code>	Shift rows or columns of image by linearly varying offset
<code>vision.GeometricTransformer</code>	Apply projective or affine transformation to an image
<code>vision.GeometricTransformEstimator</code>	Estimate geometric transformation from matching point pairs
<code>vision.GeometricTranslator</code>	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

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.Histogram</code>	Generate histogram of each input matrix. This object has no tunable properties.
<code>vision.LocalMaximaFinder</code>	Find local maxima in matrices
<code>vision.Maximum</code>	Find maximum values in input or sequence of inputs
<code>vision.Mean</code>	Find mean value of input or sequence of inputs
<code>vision.Median</code>	Find median values in an input
<code>vision.Minimum</code>	Find minimum values in input or sequence of inputs
<code>vision.PSNR</code>	Compute peak signal-to-noise ratio (PSNR) between images
<code>vision.StandardDeviation</code>	Find standard deviation of input or sequence of inputs
<code>vision.Variance</code>	Find variance values in an input or sequence of inputs
Text & Graphics	
<code>vision.AlphaBlender</code>	Combine images, overlay images, or highlight selected pixels
<code>vision.MarkerInserter</code>	Draw markers on output image
<code>vision.ShapeInserter</code>	Draw rectangles, lines, polygons, or circles on images
<code>vision.TextInserter</code>	Draw text on image or video stream
Transforms	
<code>vision.DCT</code>	Compute 2-D discrete cosine transform
<code>vision.FFT</code>	Two-dimensional discrete Fourier transform
<code>vision.HoughLines</code>	Find Cartesian coordinates of lines that are described by rho and theta pairs
<code>vision.HoughTransform</code>	Find lines in images via Hough transform

Supported Computer Vision System Toolbox System Objects (Continued)

Object	Description
<code>vision.IDCT</code>	Compute 2-D inverse discrete cosine transform
<code>vision.IFFT</code>	Two-dimensional inverse discrete Fourier transform
<code>vision.Pyramid</code>	Perform Gaussian pyramid decomposition
Utilities	
<code>vision.ImagePadder</code>	Pad or crop input image along its rows, columns, or both

Functions and Classes that Generate Code

The following Computer Vision System Toolbox functions and classes support code generation in MATLAB. See “About MATLAB Coder” for more information.

Function	Description
<code>epipolarLine</code>	Compute epipolar lines for stereo images
<code>estimateFundamentalMatrix</code>	Estimate fundamental matrix from corresponding points in stereo image
<code>estimateUncalibratedRectification</code>	Uncalibrated stereo rectification
<code>extractFeatures</code>	Extract interest point descriptors
<code>integralImage</code>	Compute integral image
<code>isEpipoleInImage</code>	Determine whether image contains epipole
<code>vision.KalmanFilter</code>	Kalman filter for object tracking

Function	Description
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:

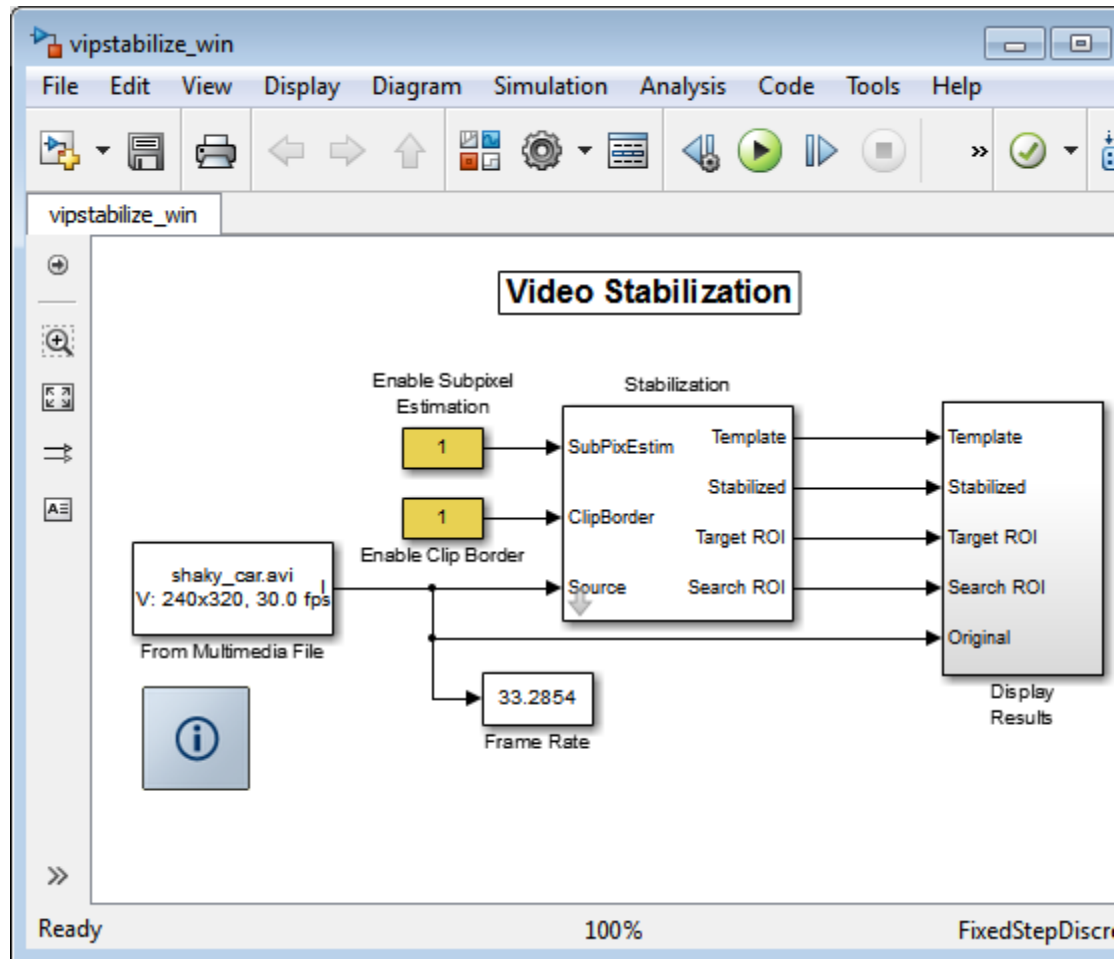
Host computer only. Excludes Real-Time Windows (RTWIN) target.

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 “Choosing a Simulation Mode”.

Define New System Objects

- “Define Basic System Objects” on page 10-2
- “Change Number of Step Inputs or Outputs” on page 10-4
- “Validate Property and Input Values” on page 10-6
- “Initialize Properties and Setup One-Time Calculations” on page 10-9
- “Set Property Values at Construction Time” on page 10-12
- “Reset Algorithm State” on page 10-15
- “Define Property Attributes” on page 10-17
- “Hide Inactive Properties” on page 10-21
- “Limit Property Values to Finite String Set” 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
- “Save System Object” on page 10-36
- “Load System Object” on page 10-39
- “Methods Timing” on page 10-42

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 Inputs or Outputs” on page 10-4

Concepts

- “Process Data Using System Objects”

Change Number of Step 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 10-6
- “Define Basic System Objects” on page 10-2

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
```

```
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 10-6

Concepts

- “Methods Timing” on page 10-42
- “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
    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 10-17

Concepts

- “Methods Timing” on page 10-42

Set Property Values at Construction Time

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
    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 10-17
- “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 10-42 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` |

Concepts

- “Methods Timing” on page 10-42

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

Use the *nontunable* attribute for a property when the algorithm depends on the value being constant once data processing starts. Defining a property as nontunable may improve the efficiency of your algorithm by removing the need to check for or react to values that change. For code generation, defining a property as nontunable allows the memory associated with that property to be optimized. You should define all properties that affect the number of input or output ports 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 or any value that can be converted to a logical. The value, however, displays as `true` or `false`. 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, but not a scaled double `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 initial 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  
end
```

Concepts

- “Class Attributes”
- “Tuning System object Properties in MATLAB”
- “Methods Timing” on page 10-42

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 Finite String Set

This example shows how to limit a property to accept only a finite set of string values.

Specify a Set of Valid String Values

String 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` |

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;
        end
    end
end
```

```
        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
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
    end
end
```

```
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 10-9

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
end
```

```
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

end
```



```
        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

end
```

See Also `matlab.system.mixin.FiniteSource` |

Concepts

- “Subclassing Multiple Classes”

Save System Object

This example shows how to save a System object.

Save System Object and Child Object

Define a `saveObjectImpl` method to specify that more than just public properties should be saved when the user saves a System object. Within this method, use the default `saveObjectImpl@matlab.System` to save public properties to the struct, `s`. Use the `saveObject` method to save child objects. Save protected and dependent properties, and finally, if the object is locked, save the object's state.

```
methods(Access=protected)
    function s = saveObjectImpl(obj)
        s = saveObjectImpl@matlab.System(obj);
        s.child = matlab.System.saveObject(obj.child);
        s.protected = obj.protected;
        s.pdependentprop = obj.pdependentprop;
        if isLocked(obj)
            s.state = obj.state;
        end
    end
end
```

Complete Class Definition File with Save and Load

```
classdef MySaveLoader < matlab.System

    properties (Access=private)
        child
        pdependentprop
    end

    properties (Access=protected)
        protected = rand;
    end

    properties (DiscreteState=true)
        state
    end
end
```

```
end

properties (Dependent)
    dependentprop
end

methods
    function obj = MySaveLoader(varargin)
        obj@matlab.System();
        setProperties(obj, nargin, varargin{:});
    end
end

methods(Access = protected)
    function setupImpl(obj, varargin)
        obj.state = 42;
    end

    function out = stepImpl(obj, in)
        obj.state = in;
        out = obj.state;
    end
end

% Serialization
methods(Access=protected)
    function s = saveObjectImpl(obj)
        % Call the base class method
        s = saveObjectImpl@matlab.System(obj);

        % Save the child System objects
        s.child = matlab.System.saveObject(obj.child);

        % Save the protected & private properties
        s.protected = obj.protected;
        s.pdependentprop = obj.pdependentprop;

        % Save the state only if object locked
        if isLocked(obj)
```

```
        s.state = obj.state;
    end
end

function loadObjectImpl(obj,s,wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
    obj.protected = s.protected;
    obj.pdependentprop = s.pdependentprop;

    % Load the state only if object locked
    if wasLocked
        obj.state = s.state;
    end

    % Call base class method to load public properties
    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
end
end
```

See Also `loadObjectImpl` |

Related Examples

- “Load System Object” on page 10-39

Load System Object

This example shows how to load a System object.

Load System Object and Child Object

Define a `loadObjectImpl` method to load a previously saved System object. Within this method, use the `matlab.System.loadObject` to assign the child object struct data to the associated object property. Assign protected and dependent property data to the associated object properties. If the object was locked when it was saved, assign the object's state to the associated property. Load the saved public properties with the `loadObjectImpl` method.

```
methods(Access=protected)
    function loadObjectImpl(obj,s,wasLocked)
        obj.child = matlab.System.loadObject(s.child);
        obj.protected = s.protected;
        obj.pdependentprop = s.pdependentprop;
        if wasLocked
            obj.state = s.state;
        end
        loadObjectImpl@matlab.System(obj,s,wasLocked);
    end
end
end
```

Complete Class Definition File with Save and Load

```
classdef MySaveLoader < matlab.System

    properties (Access=private)
        child
        pdependentprop
    end

    properties (Access=protected)
        protected = rand;
    end

    properties (DiscreteState=true)
```

```
        state
    end

    properties (Dependent)
        dependentprop
    end

    methods
        function obj = MySaveLoader(varargin)
            obj@matlab.System();
            setProperties(obj, nargin, varargin{:});
        end
    end

    methods(Access = protected)
        function setupImpl(obj,varargin)
            obj.state = 42;
        end

        function out = stepImpl(obj,in)
            obj.state = in;
            out = obj.state;
        end
    end

    % Serialization
    methods(Access=protected)
        function s = saveObjectImpl(obj)
            % Call the base class method
            s = saveObjectImpl@matlab.System(obj);

            % Save the child System objects
            s.child = matlab.System.saveObject(obj.child);

            % Save the protected & private properties
            s.protected = obj.protected;
            s.pdependentprop = obj.pdependentprop;

            % Save the state only if object locked
```



```
        if isLocked(obj)
            s.state = obj.state;
        end
    end

function loadObjectImpl(obj,s,wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Load protected and private properties
    obj.protected = s.protected;
    obj.pdependentprop = s.pdependentprop;

    % Load the state only if object locked
    if wasLocked
        obj.state = s.state;
    end

    % Call base class method to load public properties
    loadObjectImpl@matlab.System(obj,s,wasLocked);
end
end
end
```

See Also `saveObjectImpl` |

Related Examples

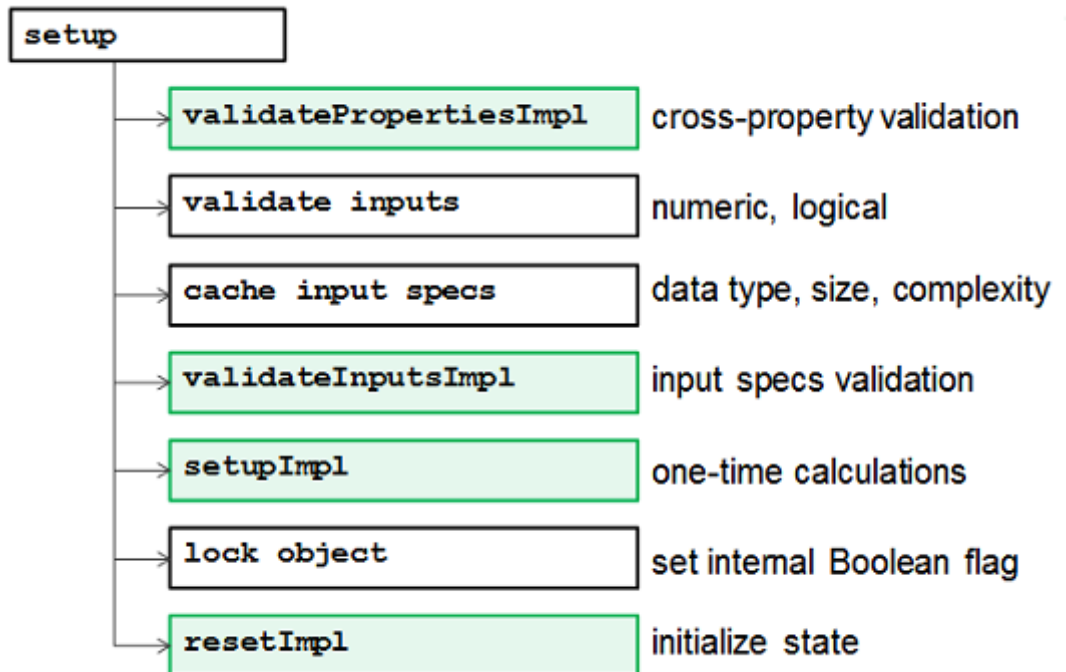
- “Save System Object” on page 10-36

Methods Timing

In this section...
“Setup Method Call Sequence” on page 10-42
“Step Method Call Sequence” on page 10-43
“Reset Method Call Sequence” on page 10-43
“Release Method Call Sequence” on page 10-44

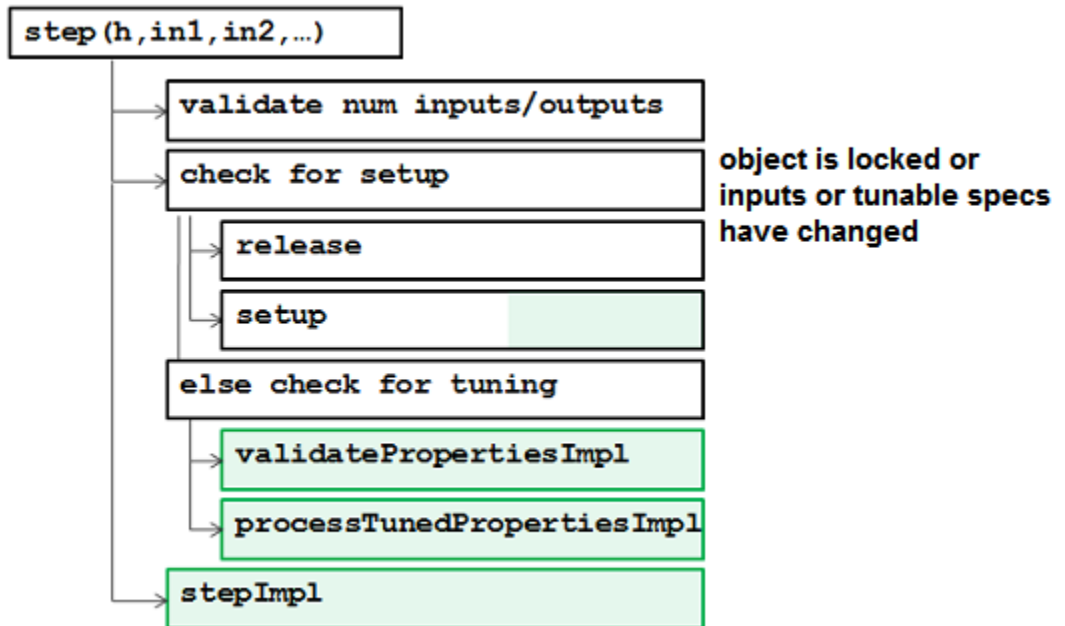
Setup Method Call Sequence

This hierarchy shows the actions performed when you call the setup method.



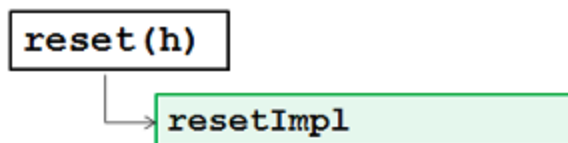
Step Method Call Sequence

This hierarchy shows the actions performed when you call the step method.



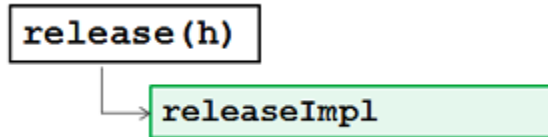
Reset Method Call Sequence

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 10-15
- “Set Property Values at Construction Time” on page 10-12
- “Define Basic System Objects” on page 10-2

Concepts

- “What are System object Methods?”
- “The Step Method”
- “Common Methods”

A

- Accelerator mode 9-10
- adjusting
 - intensity image contrast 6-2
 - RGB image contrast 6-7
- algorithms
 - bicubic interpolation 5-19
 - bilinear interpolation 5-18
 - nearest neighbor interpolation 5-17
- angles
 - rotation 5-2
- annotating
 - AVI files 2-35
- arithmetic operations
 - fixed-point 8-10
- audio
 - exporting to video file 1-18
- Autothreshold block
 - to perform thresholding 1-34
- AVI files
 - annotating 2-35
 - cropping 1-12
 - saving to multiple files 1-12
 - splitting 1-12

B

- background
 - estimation 7-8
 - pixels 3-3
- batch processing 1-7
- bicubic interpolation 5-19
- bilinear interpolation 5-18
- binary
 - conversion from intensity 1-29
 - images 1-53
- block parameters
 - fixed-point 8-25
- blurring images 6-18
- Boolean matrices 1-53

- boundaries
 - of objects 3-3
- brightening images 7-8

C

- casts
 - fixed-point 8-16
- changing
 - image size 5-8
 - intensity image contrast 6-2
 - RGB image contrast 6-7
- chroma components
 - of images 1-24
- chroma resampling 1-24
- chrominance resampling 1-24
- code generation
 - computer vision objects 9-2
 - fixed-point 8-3
- color
 - definition of 1-51
- colormaps 1-53
- column-major format 1-52
- complex multiplication
 - fixed-point 8-13
- continuous rotation 5-2
- contrast
 - increasing 1-2
- conventions
 - column-major format 1-52
- conversion
 - intensity to binary 1-29
 - R'G'B' to intensity 1-41
- correction
 - of uneven lighting 7-8
- correlation
 - used in object tracking 4-19
- counting objects 7-15
- cropping
 - AVI files 1-12

images 5-12

D

data types 1-51

definition of

intensity and color 1-51

detection of

edges 3-3

lines 3-11

downsampling

chroma components 1-24

dynamic range 1-51

E

edge

pixels 3-3

thinning 3-3

edge detection 3-3

estimation

of image background 7-8

exporting

video files 1-2

F

feature extraction

finding angles between lines 3-33

finding edges 3-3

finding lines 3-11

filtering

median 6-12

finding

angles between lines 3-33

edges of objects 3-3

histograms of images 7-2

lines in images 3-11

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 6-18

G

gradient components

of images 3-3

H

histograms
 of images 7-2

I

image data
 storage of 1-52
image rotation 5-2
image sequence processing 1-7
image types 1-52
images
 binary 1-53
 brightening 7-8
 correcting for uneven lighting 7-8
 counting objects in 7-15
 cropping 5-12
 finding angles between lines 3-33
 finding edges in 3-3
 finding histograms of 7-2
 finding lines in 3-11
 gradient components 3-3
 intensity 1-53
 intensity to binary conversion 1-29
 labeling objects in 7-15
 lightening 7-8
 noisy 6-12
 removing salt and pepper noise 6-12
 resizing of 5-8
 RGB 1-53
 rotation of 5-2
 sharpening and blurring 6-18
 true-color 1-53
 types of 1-52
importing
 multimedia files 1-4
improvement
 of performance 9-10
increasing video contrast 1-2
inherit via internal rule 8-29

intensity
 conversion from R'G'B' 1-41
 conversion to binary 1-29
 definition of 1-51
 images 1-53
intensity images
 adjusting the contrast of 6-2
interpolation
 bicubic 5-19
 bilinear 5-18
 examples 5-17
 nearest neighbor 5-17
 overview 5-17
irregular illumination 7-8

L

labeling objects 7-15
lightening images 7-8
location of
 lines 3-11
 object edges 3-3
 objects in an image 4-19
logging
 fixed-point data types 8-28
luma components
 applying highpass filter 6-18
 applying lowpass filter 6-18
 of images 1-24
luminance 1-24

M

median filtering 6-12
methods
 interpolation 5-17
 thresholding 7-8
modes
 Normal and Accelerator 9-10
modular arithmetic 8-10

- morphology
 - opening 7-15
 - STREL object 7-15
- multimedia files
 - exporting 1-2
 - importing 1-4
 - viewing 1-4
- multiplication
 - fixed-point 8-13

N

- nearest neighbor interpolation 5-17
- noise removal
 - salt and pepper 6-12
- nonuniform illumination
 - correcting for 7-8
- Normal mode 9-10

O

- object boundaries 3-3
- object tracking
 - using correlation 4-19
- objects
 - delineating 7-8
 - location of 4-19
- opening 7-15
- operations
 - thresholding 1-29
- overflow handling 8-6
- overview of
 - interpolation 5-17

P

- performance
 - improving 9-10
- precision
 - fixed-point data types 8-6
- preferences 8-22

R

- R'B'G'
 - conversion to intensity 1-41
- range
 - fixed-point data types 8-6
- reception
 - of an RGB image 1-24
- reduction
 - of image size 5-8
- region of interest
 - cropping to 5-12
 - visualizing 4-19
- relational operators
 - to perform thresholding 1-29
- removal of
 - salt and pepper noise 6-12
- resampling
 - chroma 1-24
- resizing
 - images 5-8
- RGB images 1-53
 - adjusting the contrast of 6-7
- rotation
 - continual 5-2
 - of an image 5-2
- rounding
 - fixed-point data types 8-7

S

- salt and pepper noise removal 6-12
- saturation 8-6
- saving
 - to multiple AVI files 1-12
- scaling 1-51 8-5
- sectioning
 - AVI files 1-12
- sequence
 - of images 1-7
- sharpening images 6-18

- shrinking
 - image size 5-8
- Sobel kernel 3-3
- splitting
 - AVI files 1-12
- storage of image data 1-52
- STREL object 7-15
- System object
 - fixed point
 - Computer Vision System Toolbox™
objects 8-21
 - preferences 8-22
 - system-level settings
 - fixed-point 8-28

T

- techniques
 - thresholding 7-8
- thresholding operation 1-29
 - with uneven lighting 1-34
- thresholding techniques 7-8
- tracking
 - of an object 4-19
- transmission
 - of an RGB image 1-24
- trimming
 - images 5-12

- true-color images 1-53
- two's complement 8-11
- types of images 1-52

U

- uneven lighting
 - correcting for 7-8

V

- video
 - annotating AVI files with video frame
numbers 2-35
 - exporting from video file 1-2
 - importing from multimedia file 1-4
 - increasing the contrast of 1-2
 - interpretation of 1-51
- video files
 - exporting audio and video 1-18
- viewing
 - multimedia files 1-4
 - video files 1-4

W

- wrapping
 - fixed-point data types 8-6