Computer Vision System Toolbox™

User's Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Computer Vision System Toolbox ™ User's Guide

© COPYRIGHT 2000–2018 by 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

Contents

Featured Examples

1

Point Cloud Processing

2

Using the Installer for Computer Vision System Toolbox Product

3

Input, Output, and Conversions

4

Display and Graphics

5

Registration and Stereo Vision

6

Object Detection

7

Motion Estimation and Tracking

8

Geometric Transformations

9

Filters, Transforms, and Enhancements

10

Statistics and Morphological Operations

11

Fixed-Point Design

12

Code Generation

13

Featured Examples

Read and Play a Video File

Load the video using a video reader object.

```
videoFReader = vision.VideoFileReader('ecolicells.avi');
```

Create a video player object to play the video file.

```
videoPlayer = vision.VideoPlayer;
```

Use a while loop to read and play the video frames. Pause for 0.1 seconds after displaying each frame.

```
while ~isDone(videoFReader)
  videoFrame = videoFReader();
  videoPlayer(videoFrame);
  pause(0.1)
end
```
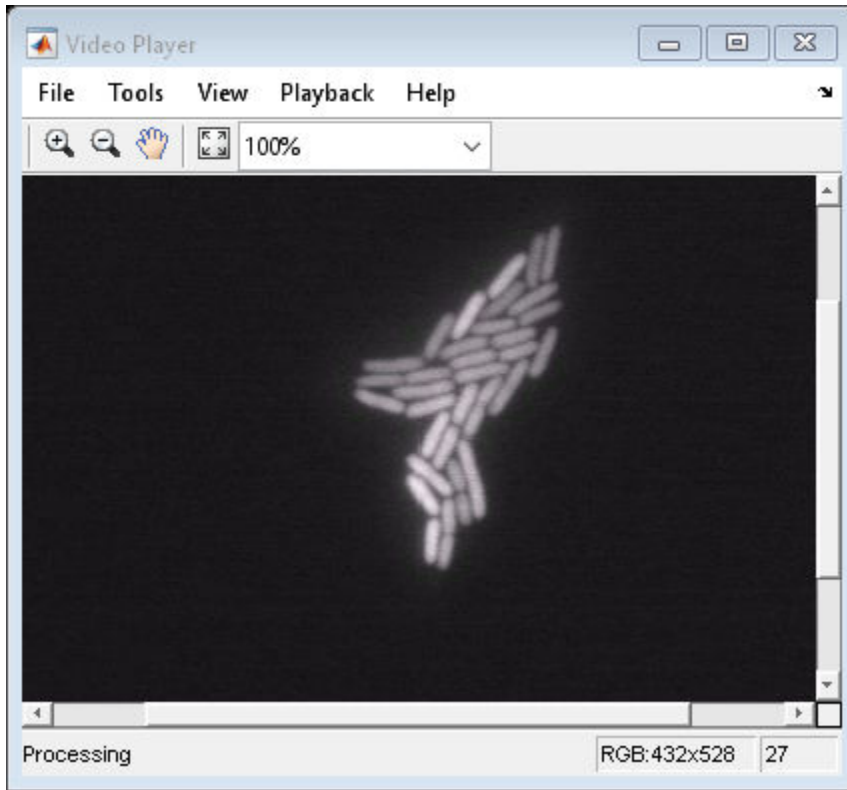
Release the objects.

```
release(videoPlayer);
```
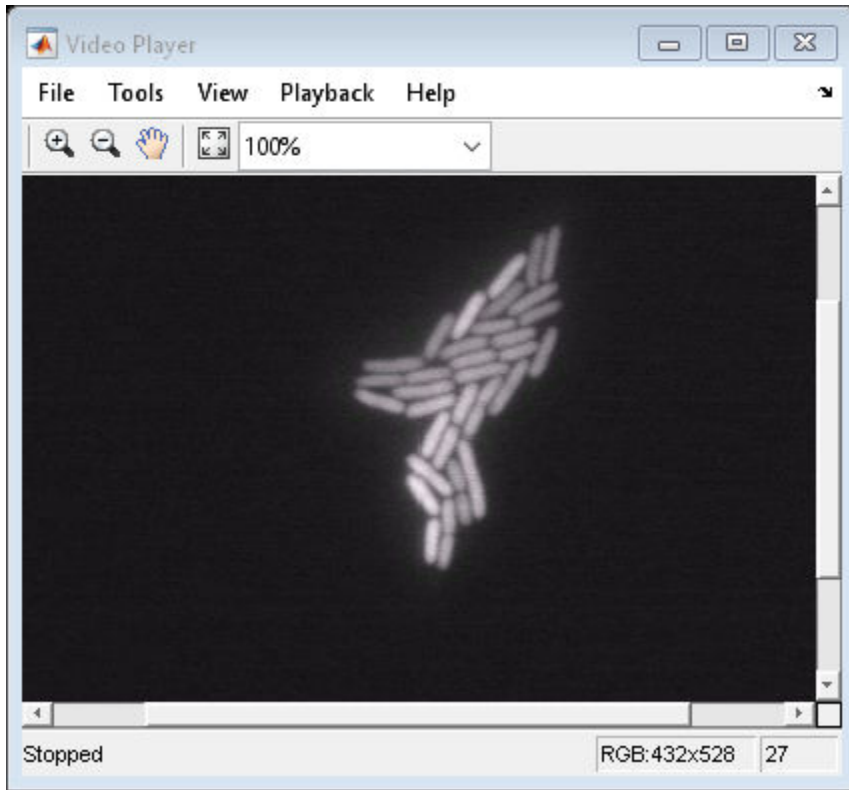
```
release(videoFReader);
```

Find Vertical and Horizontal Edges in Image

Construct Haar-like wavelet filters to find vertical and horizontal edges in an image.

Read the input image and compute the integral image.

```
I = imread('pout.tif');
intImage = integralImage(I);
```

Construct Haar-like wavelet filters. Use the dot notation to find the vertical filter from the horizontal filter.

```
horiH = integralKernel([1 1 4 3; 1 4 4 3],[-1, 1]);
vertH = horiH.'

vertH =
  integralKernel with properties:

    BoundingBoxes: [2x4 double]
          Weights: [-1 1]
     Coefficients: [4x6 double]
           Center: [2 3]
             Size: [4 6]
      Orientation: 'upright'
```

Display the horizontal filter.

```
imtool(horiH.Coefficients, 'InitialMagnification','fit');
```

Compute the filter responses.

```
horiResponse = integralFilter(intImage,horiH);
vertResponse = integralFilter(intImage,vertH);
```

Display the results.

```
figure;
imshow(horiResponse,[]);
title('Horizontal edge responses');
```


Horizontal edge responses

```
figure;
imshow(vertResponse,[]);
title('Vertical edge responses');
```

Vertical edge responses

Blur an Image Using an Average Filter

Read and display the input image.

```
I = imread('pout.tif');
imshow(I);
```


Compute the integral image.

```
intImage = integralImage(I);
```

Apply a 7-by-7 average filter.

```
avgH = integralKernel([1 1 7 7], 1/49);
J = integralFilter(intImage, avgH);
```

Cast the result back to the same class as the input image.

```
J = uint8(J);
figure
imshow(J);
```

Define a Filter to Approximate a Gaussian Second Order Partial Derivative in Y Direction

```
ydH = integralKernel([1,1,5,9;1,4,5,3], [1, -3]);
```

You can also define this filter as integralKernel([1,1,5,3;1,4,5,3;1,7,5,3], [1, -2, 1]);|. This filter definition is less efficient because it requires three bounding boxes.

Visualize the filter.

```
ydH.Coefficients
```

ans = *9×5*

```
    1     1     1     1     1
    1     1     1     1     1
    1     1     1     1     1
   -2    -2    -2    -2    -2
   -2    -2    -2    -2    -2
   -2    -2    -2    -2    -2
    1     1     1     1     1
    1     1     1     1     1
    1     1     1     1     1
```

Find Corresponding Interest Points Between Pair of Images

Find corresponding interest points between a pair of images using local neighbhorhoods and the Harris algorithm.

Read the stereo images.

```
I1 = rgb2gray(imread('viprectification_deskLeft.png'));
I2 = rgb2gray(imread('viprectification_deskRight.png'));
```

Find the corners.

```
points1 = detectHarrisFeatures(I1);
points2 = detectHarrisFeatures(I2);
```

Extract the neighborhood features.

```
[features1,valid_points1] = extractFeatures(I1,points1);
[features2,valid_points2] = extractFeatures(I2,points2);
```

Match the features.

```
indexPairs = matchFeatures(features1,features2);
```

Retrieve the locations of the corresponding points for each image.

```
matchedPoints1 = valid_points1(indexPairs(:,1),:);
matchedPoints2 = valid_points2(indexPairs(:,2),:);
```

Visualize the corresponding points. You can see the effect of translation between the two images despite several erroneous matches.

```
figure; showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2);
```

Find Corresponding Points Using SURF Features

Use the SURF local feature detector function to find the corresponding points between two images that are rotated and scaled with respect to each other.

Read the two images.

```
I1 = imread('cameraman.tif');
I2 = imresize(imrotate(I1,-20),1.2);
```

Find the SURF features.

```
points1 = detectSURFFeatures(I1);
points2 = detectSURFFeatures(I2);
```

Extract the features.

```
[f1,vpts1] = extractFeatures(I1,points1);
[f2,vpts2] = extractFeatures(I2,points2);
```

Retrieve the locations of matched points.

```
indexPairs = matchFeatures(f1,f2) ;
matchedPoints1 = vpts1(indexPairs(:,1));
matchedPoints2 = vpts2(indexPairs(:,2));
```

Display the matching points. The data still includes several outliers, but you can see the effects of rotation and scaling on the display of matched features.

```
figure; showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2);
legend('matched points 1','matched points 2');
```

Detect SURF Interest Points in a Grayscale Image

Read image and detect interest points.

```
I = imread('cameraman.tif');
points = detectSURFFeatures(I);
```

Display locations of interest in image.

```
imshow(I); hold on;
plot(points.selectStrongest(10));
```

Using LBP Features to Differentiate Images by Texture

Read images that contain different textures.

```
brickWall = imread('bricks.jpg');
rotatedBrickWall = imread('bricksRotated.jpg');
carpet = imread('carpet.jpg');
```

Display the images.

```
figure
imshow(brickWall)
title('Bricks')
```


Bricks
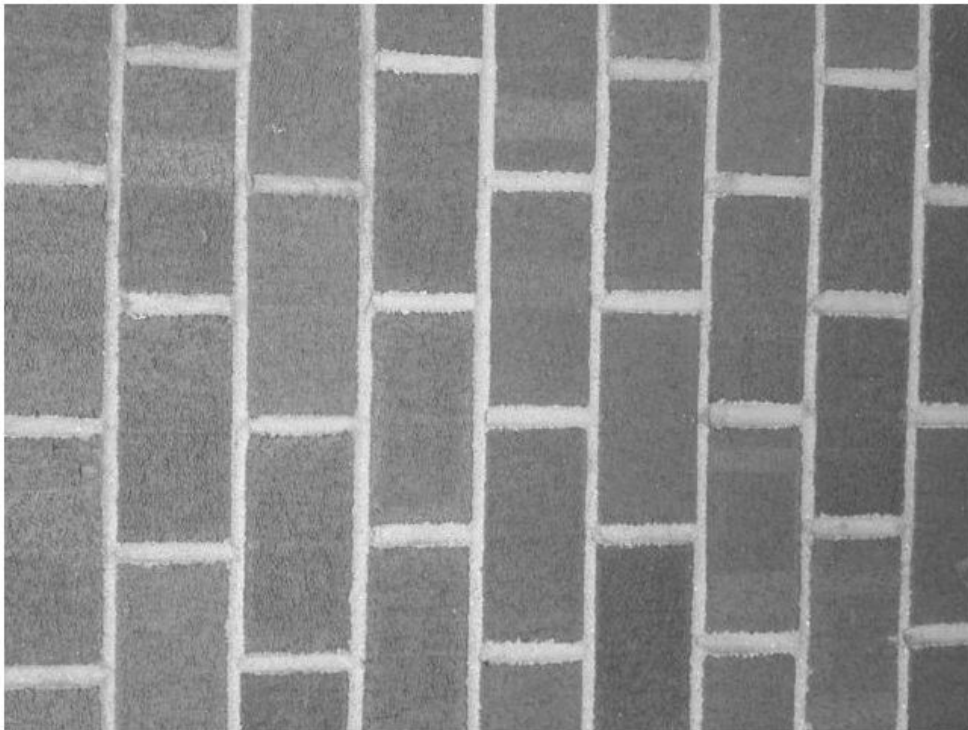
```
figure
imshow(rotatedBrickWall)
title('Rotated Bricks')
```

Rotated Bricks
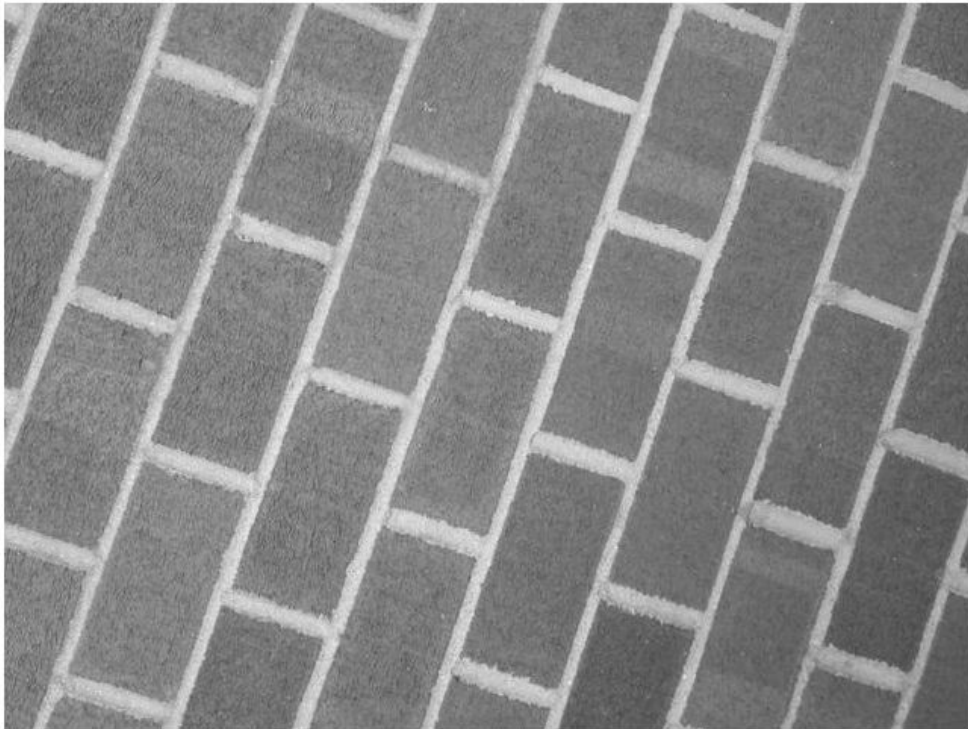

```
figure
imshow(carpet)
title('Carpet')
```

Carpet
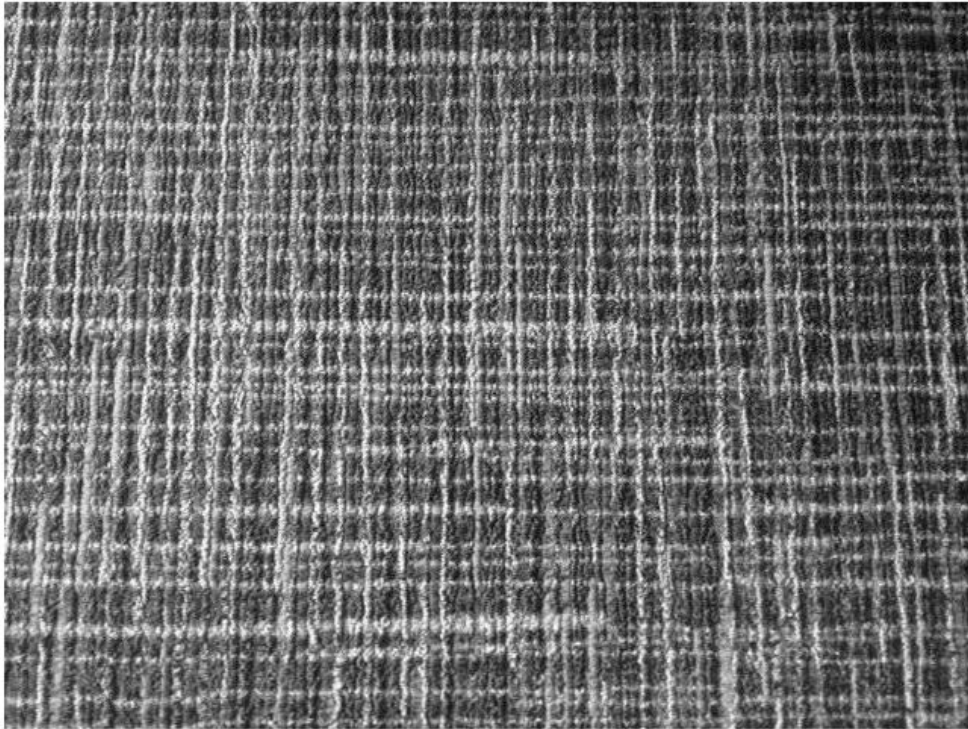
Extract LBP features from the images to encode their texture information.

```
lbpBricks1 = extractLBPFeatures(brickWall,'Upright',false);
lbpBricks2 = extractLBPFeatures(rotatedBrickWall,'Upright',false);
lbpCarpet = extractLBPFeatures(carpet,'Upright',false);
```

Gauge the similarity between the LBP features by computing the squared error between them.

```
brickVsBrick = (lbpBricks1 - lbpBricks2).^2;
brickVsCarpet = (lbpBricks1 - lbpCarpet).^2;
```

Visualize the squared error to compare bricks versus bricks and bricks versus carpet. The squared error is smaller when images have similar texture.

```
figure
bar([brickVsBrick; brickVsCarpet]','grouped')
title('Squared Error of LBP Histograms')
xlabel('LBP Histogram Bins')
legend('Bricks vs Rotated Bricks','Bricks vs Carpet')
```
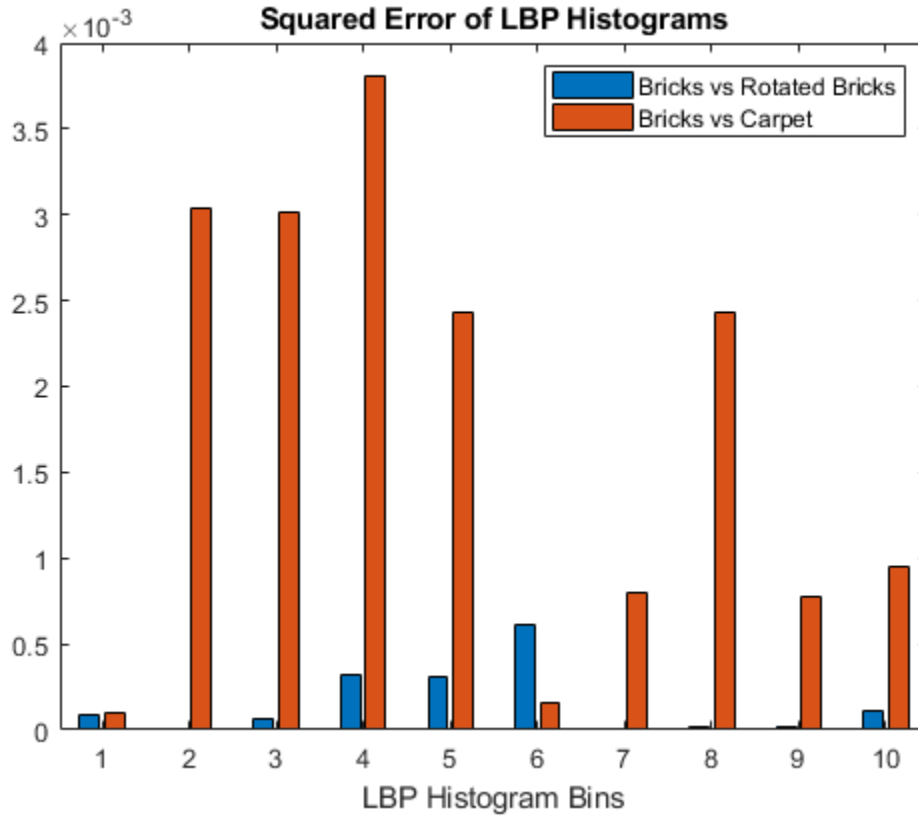
Extract and Plot HOG Features

Read the image of interest.

```
img = imread('cameraman.tif');
```

Extract HOG features.

```
[featureVector,hogVisualization] = extractHOGFeatures(img);
```

Plot HOG features over the original image.

```
figure;
imshow(img);
hold on;
plot(hogVisualization);
```

Find Corresponding Interest Points Between Pair of Images

Find corresponding interest points between a pair of images using local neighbhorhoods and the Harris algorithm.

Read the stereo images.

```
I1 = rgb2gray(imread('viprectification_deskLeft.png'));
I2 = rgb2gray(imread('viprectification_deskRight.png'));
```

Find the corners.

```
points1 = detectHarrisFeatures(I1);
points2 = detectHarrisFeatures(I2);
```

Extract the neighborhood features.

```
[features1,valid_points1] = extractFeatures(I1,points1);
[features2,valid_points2] = extractFeatures(I2,points2);
```

Match the features.

```
indexPairs = matchFeatures(features1,features2);
```

Retrieve the locations of the corresponding points for each image.

```
matchedPoints1 = valid_points1(indexPairs(:,1),:);
matchedPoints2 = valid_points2(indexPairs(:,2),:);
```

Visualize the corresponding points. You can see the effect of translation between the two images despite several erroneous matches.

```
figure; showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2);
```

Recognize Text Within an Image

```
businessCard  = imread('businessCard.png');
ocrResults    = ocr(businessCard)

ocrResults =
  ocrText with properties:

                   Text: '' MathWorks®...'
    CharacterBoundingBoxes: [103x4 double]
      CharacterConfidences: [103x1 single]
                   Words: {16x1 cell}
        WordBoundingBoxes: [16x4 double]
          WordConfidences: [16x1 single]


recognizedText = ocrResults.Text;
figure;
imshow(businessCard);
text(600, 150, recognizedText, 'BackgroundColor', [1 1 1]);
```

1-25

Run Nonmaximal Suppression on Bounding Boxes Using People Detector

Load the pretrained people detector and disable bounding box merging.

```
peopleDetector = vision.PeopleDetector('ClassificationThreshold',...
        0,'MergeDetections',false);
```

Read an image, run the people detector, and then insert bounding boxes with confidence scores.

```
I = imread('visionteam1.jpg');
[bbox,score] = step(peopleDetector,I);
I1 = insertObjectAnnotation(I,'rectangle',bbox,...
        cellstr(num2str(score)),'Color','r');
```

Run nonmaximal suppression on the bounding boxes.

```
[selectedBbox,selectedScore] = selectStrongestBbox(bbox,score);
I2 = insertObjectAnnotation(I,'rectangle',selectedBbox,...
        cellstr(num2str(selectedScore)),'Color','r');
```

Display detection before and after suppression.

```
figure, imshow(I1); ...
title('Detected people and detection scores before suppression');
```

Detected people and detection scores before suppression

```
figure, imshow(I2); ...
title('Detected people and detection scores after suppression');
```

Detected people and detection scores after suppression

Train Stop Sign Detector

Load the positive samples data from a MAT file. The file contains a table specifying bounding boxes for several object categories. The table was exported from the `Training Image Labeler` app.

Load positive samples.

```
load('stopSignsAndCars.mat');
```

Select the bounding boxes for stop signs from the table.

```
positiveInstances = stopSignsAndCars(:,1:2);
```

Add the image folder to the MATLAB path.

```
imDir = fullfile(matlabroot,'toolbox','vision','visiondata',...
    'stopSignImages');
addpath(imDir);
```

Specify the folder for negative images.

```
negativeFolder = fullfile(matlabroot,'toolbox','vision','visiondata',...
    'nonStopSigns');
```

Create an `imageDatastore` object containing negative images.

```
negativeImages = imageDatastore(negativeFolder);
```

Train a cascade object detector called 'stopSignDetector.xml' using HOG features. NOTE: The command can take several minutes to run.

```
trainCascadeObjectDetector('stopSignDetector.xml',positiveInstances, ...
    negativeFolder,'FalseAlarmRate',0.1,'NumCascadeStages',5);
```

```
Automatically setting ObjectTrainingSize to [35, 32]
Using at most 42 of 42 positive samples per stage
Using at most 84 negative samples per stage

--cascadeParams--
Training stage 1 of 5
[..................................................................]
Used 42 positive and 84 negative samples
Time to train stage 1: 1 seconds
```

```
Training stage 2 of 5
[....................................................................]
Used 42 positive and 84 negative samples
Time to train stage 2: 0 seconds

Training stage 3 of 5
[....................................................................]
Used 42 positive and 84 negative samples
Time to train stage 3: 3 seconds

Training stage 4 of 5
[....................................................................]
Used 42 positive and 84 negative samples
Time to train stage 4: 9 seconds

Training stage 5 of 5
[....................................................................]
Used 42 positive and 17 negative samples
Time to train stage 5: 14 seconds

Training complete
```

Use the newly trained classifier to detect a stop sign in an image.

```
detector = vision.CascadeObjectDetector('stopSignDetector.xml');
```

Read the test image.

```
img = imread('stopSignTest.jpg');
```

Detect a stop sign.

```
bbox = step(detector,img);
```

Insert bounding box rectangles and return the marked image.

```
 detectedImg = insertObjectAnnotation(img,'rectangle',bbox,'stop sign');
```

Display the detected stop sign.

```
figure; imshow(detectedImg);
```

Remove the image directory from the path.

```
rmpath(imDir);
```

Track an Occluded Object

Detect and track a ball using Kalman filtering, foreground detection, and blob analysis.

Create System objects to read the video frames, detect foreground physical objects, and display results.

```
videoReader = vision.VideoFileReader('singleball.mp4');
videoPlayer = vision.VideoPlayer('Position',[100,100,500,400]);
foregroundDetector = vision.ForegroundDetector('NumTrainingFrames',10,...
                'InitialVariance',0.05);
blobAnalyzer = vision.BlobAnalysis('AreaOutputPort',false,...
                'MinimumBlobArea',70);
```

Process each video frame to detect and track the ball. After reading the current video frame, the example searches for the ball by using background subtraction and blob analysis. When the ball is first detected, the example creates a Kalman filter. The Kalman filter determines the ball's location, whether it is detected or not. If the ball is detected, the Kalman filter first predicts its state at the current video frame. The filter then uses the newly detected location to correct the state, producing a filtered location. If the ball is missing, the Kalman filter solely relies on its previous state to predict the ball's current location.

```
  kalmanFilter = []; isTrackInitialized = false;
   while ~isDone(videoReader)
     colorImage  = step(videoReader);

     foregroundMask = step(foregroundDetector, rgb2gray(colorImage));
     detectedLocation = step(blobAnalyzer,foregroundMask);
     isObjectDetected = size(detectedLocation, 1) > 0;

     if ~isTrackInitialized
       if isObjectDetected
         kalmanFilter = configureKalmanFilter('ConstantAcceleration',...
                 detectedLocation(1,:), [1 1 1]*1e5, [25, 10, 10], 25);
         isTrackInitialized = true;
       end
       label = ''; circle = zeros(0,3);
     else
       if isObjectDetected
         predict(kalmanFilter);
         trackedLocation = correct(kalmanFilter, detectedLocation(1,:));
         label = 'Corrected';
```

```
        else
          trackedLocation = predict(kalmanFilter);
          label = 'Predicted';
        end
        circle = [trackedLocation, 5];
    end

    colorImage = insertObjectAnnotation(colorImage,'circle',...
              circle,label,'Color','red');
    step(videoPlayer,colorImage);
end
```
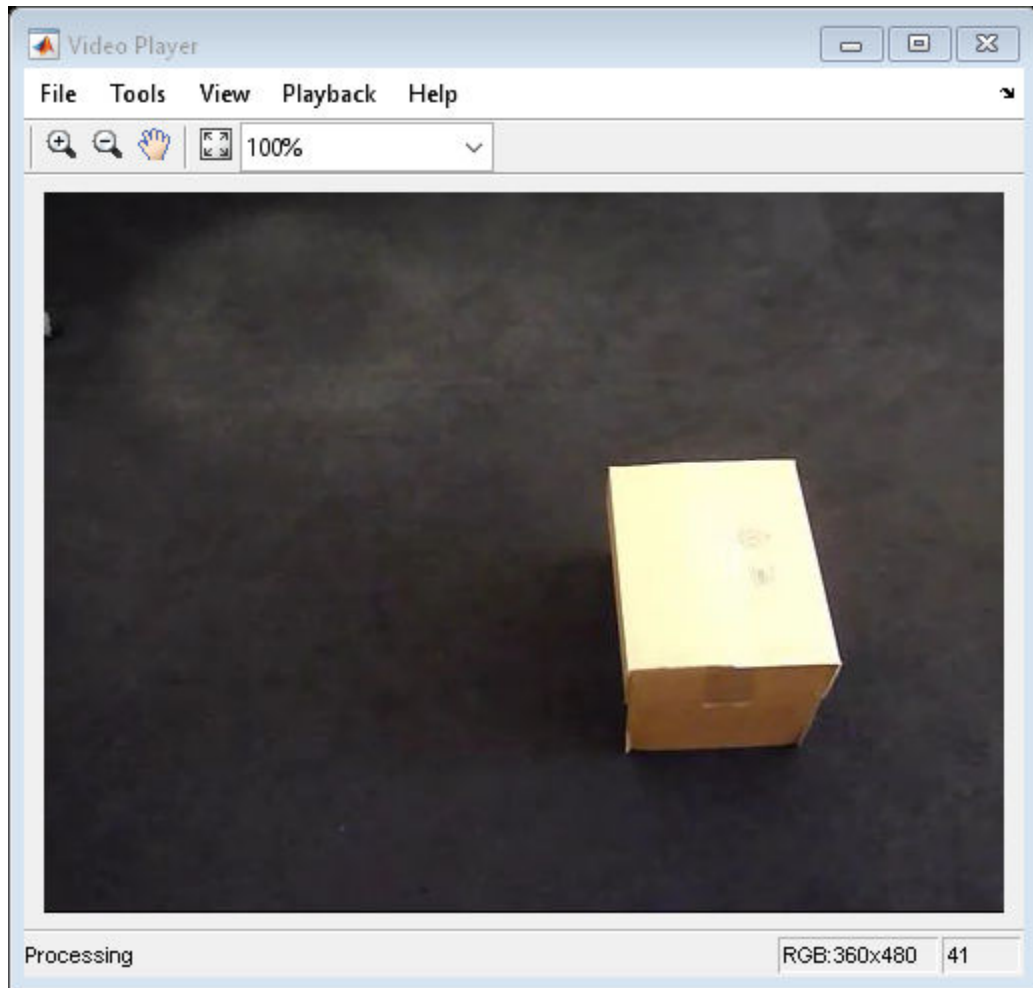
Release resources.

```
release(videoPlayer);
release(videoReader);
```
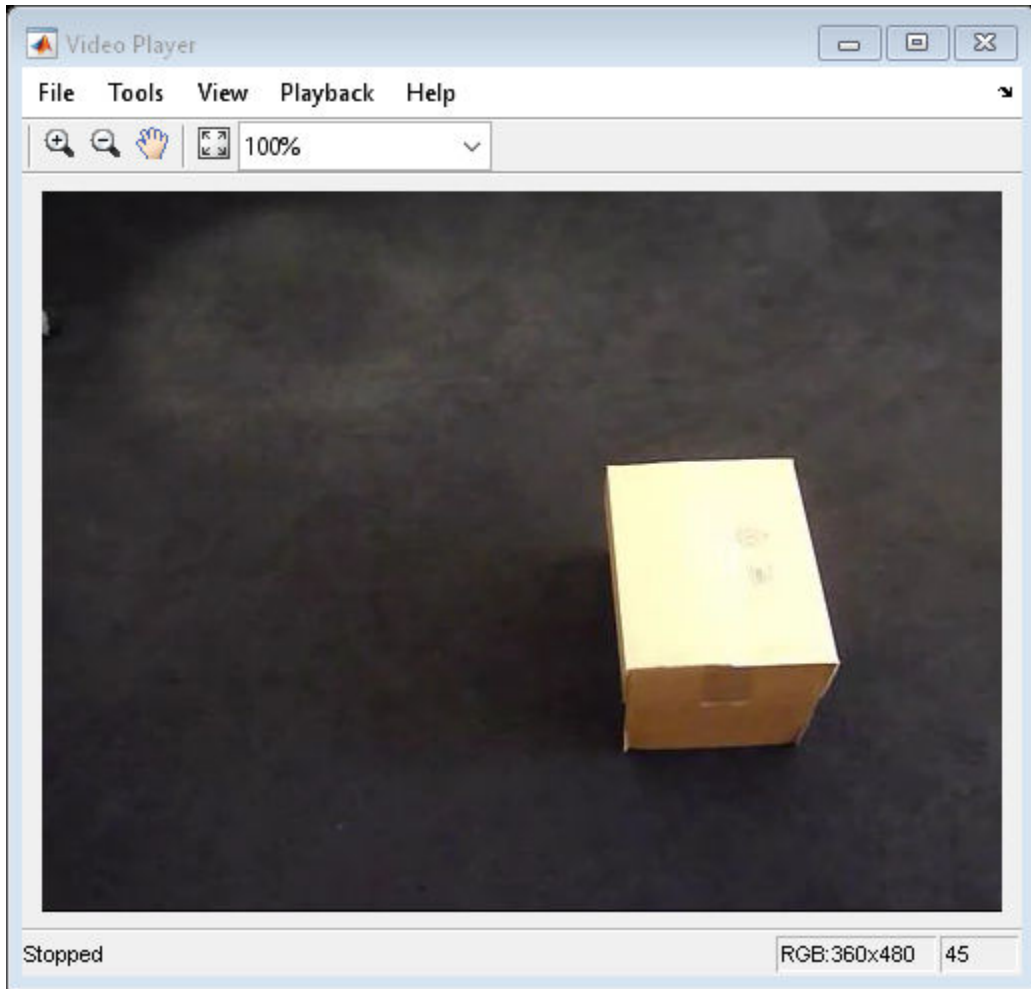
Track a Face in Scene

Create System objects for reading and displaying video and for drawing a bounding box of the object.

```
videoFileReader = vision.VideoFileReader('visionface.avi');
videoPlayer = vision.VideoPlayer('Position',[100,100,680,520]);
```

Read the first video frame, which contains the object, define the region.

```
objectFrame = videoFileReader();
objectRegion = [264,122,93,93];
```

As an alternative, you can use the following commands to select the object region using a mouse. The object must occupy the majority of the region:

figure; imshow(objectFrame);

objectRegion=round(getPosition(imrect))

Show initial frame with a red bounding box.

```
objectImage = insertShape(objectFrame,'Rectangle',objectRegion,'Color','red');
figure;
imshow(objectImage);
title('Red box shows object region');
```

Red box shows object region

Detect interest points in the object region.

```
points = detectMinEigenFeatures(rgb2gray(objectFrame),'ROI',objectRegion);
```

Display the detected points.

```
pointImage = insertMarker(objectFrame,points.Location,'+','Color','white');
figure;
imshow(pointImage);
title('Detected interest points');
```

Detected interest points

Create a tracker object.

```
tracker = vision.PointTracker('MaxBidirectionalError',1);
```

Initialize the tracker.

```
initialize(tracker,points.Location,objectFrame);
```

Read, track, display points, and results in each video frame.

```
while ~isDone(videoFileReader)
    frame = videoFileReader();
    [points,validity] = tracker(frame);
    out = insertMarker(frame,points(validity, :),'+');
```

1-39

```
        videoPlayer(out);
end
```
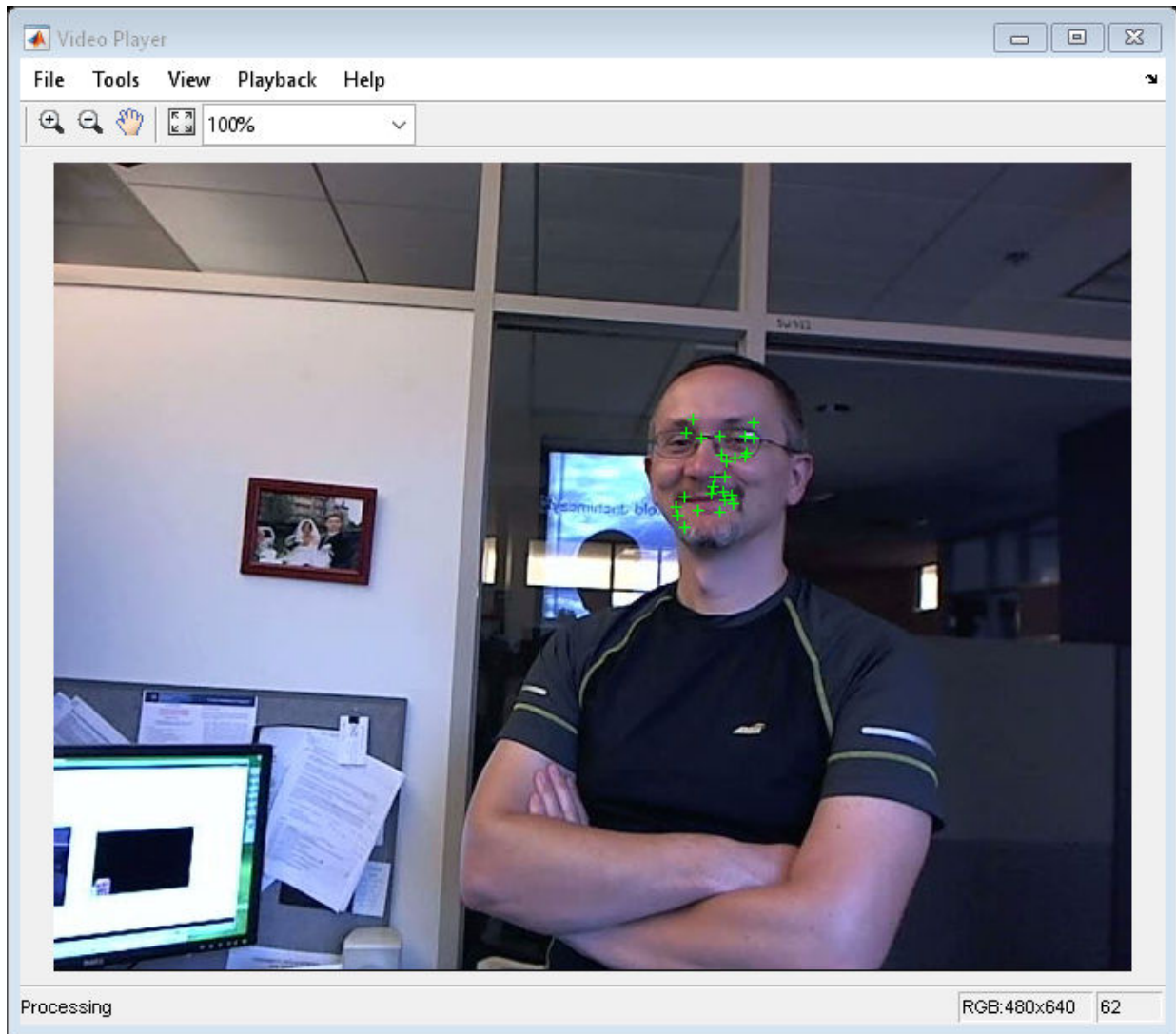

Release the video reader and player.

```
release(videoPlayer);
```
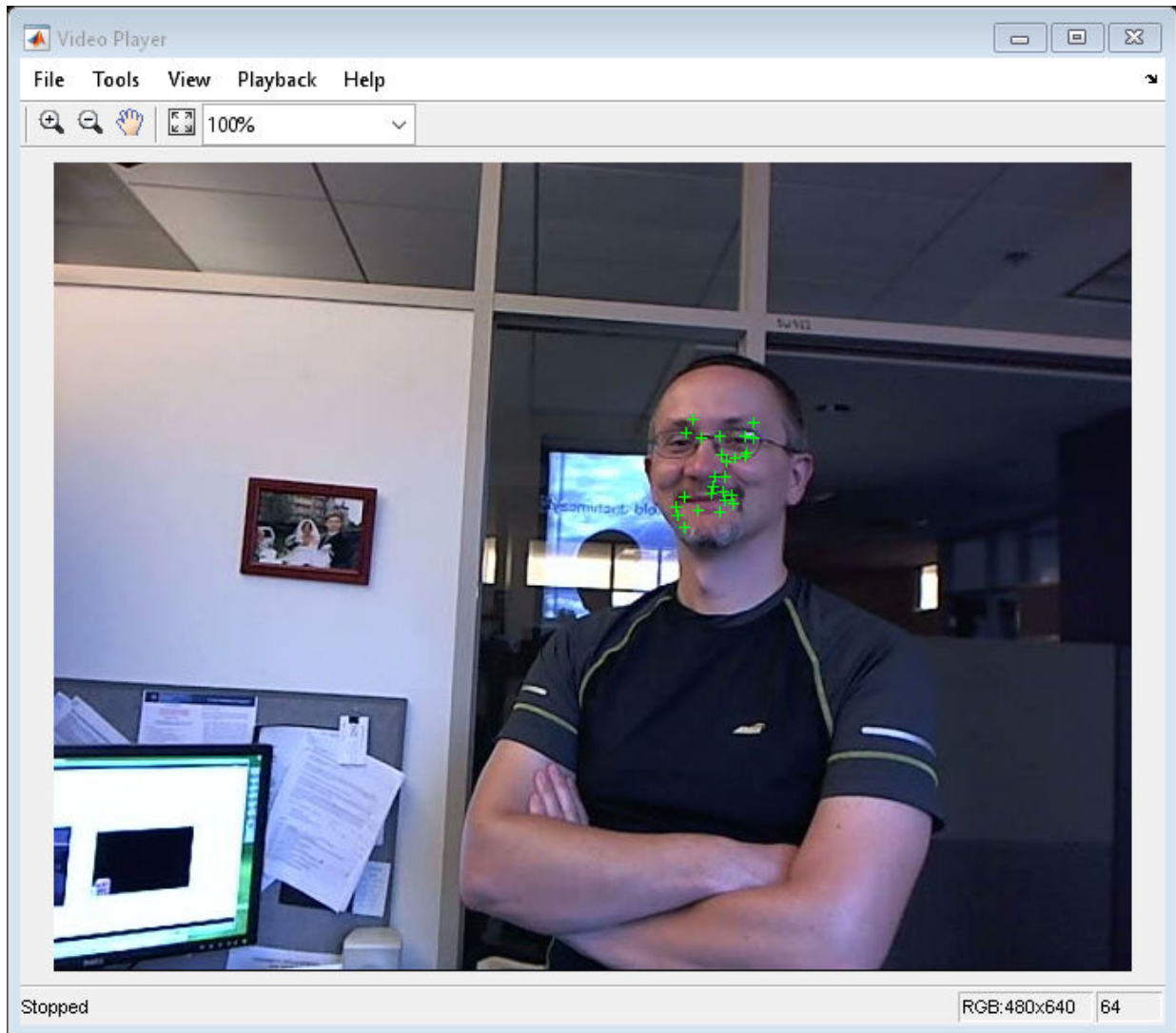
```
release(videoFileReader);
```

Assign Detections to Tracks in a Single Video Frame

This example shows you how to assign a detection to a track for a single video frame.

Set the predicted locations of objects in the current frame. Obtain predictions using the Kalman filter System object.

```
predictions = [1,1;2,2];
```

Set the locations of the objects detected in the current frame. For this example, there are 2 tracks and 3 new detections. Thus, at least one of the detections is unmatched, which can indicate a new track.

```
detections = [1.1,1.1;2.1,2.1;1.5,3];
```

Preallocate a cost matrix.

```
cost = zeros(size(predictions,1),size(detections,1));
```

Compute the cost of each prediction matching a detection. The cost here, is defined as the Euclidean distance between the prediction and the detection.

```
for i = 1:size(predictions, 1)
    diff = detections - repmat(predictions(i,:),[size(detections,1),1]);
    cost(i, :) = sqrt(sum(diff .^ 2,2));
end
```

Associate detections with predictions. Detection 1 should match to track 1, and detection 2 should match to track 2. Detection 3 should be unmatched.

```
[assignment,unassignedTracks,unassignedDetections] = ...
        assignDetectionsToTracks(cost,0.2);
  figure;
  plot(predictions(:,1),predictions(:,2),'*',detections(:,1),...
          detections(:,2),'ro');
  hold on;
  legend('predictions','detections');
  for i = 1:size(assignment,1)
    text(predictions(assignment(i, 1),1)+0.1,...
          predictions(assignment(i,1),2)-0.1,num2str(i));
    text(detections(assignment(i, 2),1)+0.1,...
          detections(assignment(i,2),2)-0.1,num2str(i));
  end
  for i = 1:length(unassignedDetections)
```

```
    text(detections(unassignedDetections(i),1)+0.1,...
            detections(unassignedDetections(i),2)+0.1,'unassigned');
end
xlim([0,4]);
ylim([0,4]);
```
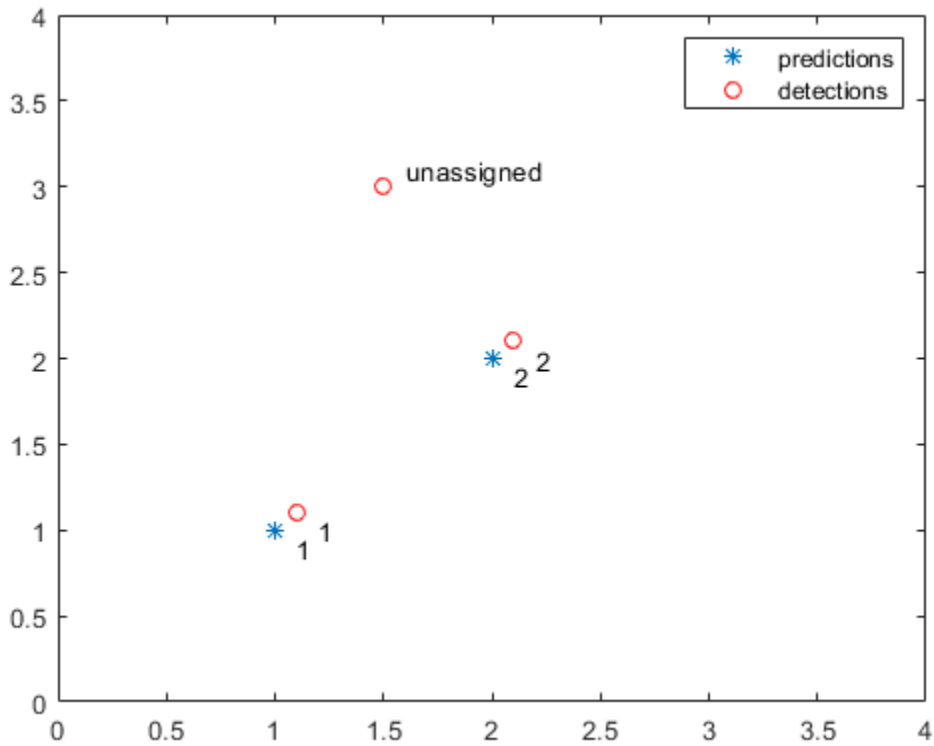
Create 3-D Stereo Display

Load parameters for a calibrated stereo pair of cameras.

```
load('webcamsSceneReconstruction.mat')
```

Load a stereo pair of images.

```
I1 = imread('sceneReconstructionLeft.jpg');
I2 = imread('sceneReconstructionRight.jpg');
```

Rectify the stereo images.

```
[J1, J2] = rectifyStereoImages(I1, I2, stereoParams);
```

Create the anaglyph.

```
A = stereoAnaglyph(J1, J2);
```

Display the anaglyph. Use red-blue stereo glasses to see the stereo effect.

```
figure; imshow(A);
```

Measure Distance from Stereo Camera to a Face

Load stereo parameters.

```
load('webcamsSceneReconstruction.mat');
```

Read in the stereo pair of images.

```
I1 = imread('sceneReconstructionLeft.jpg');
I2 = imread('sceneReconstructionRight.jpg');
```

Undistort the images.

```
I1 = undistortImage(I1,stereoParams.CameraParameters1);
I2 = undistortImage(I2,stereoParams.CameraParameters2);
```

Detect a face in both images.

```
faceDetector = vision.CascadeObjectDetector;
face1 = faceDetector(I1);
face2 = faceDetector(I2);
```

Find the center of the face.

```
center1 = face1(1:2) + face1(3:4)/2;
center2 = face2(1:2) + face2(3:4)/2;
```

Compute the distance from camera 1 to the face.

```
point3d = triangulate(center1, center2, stereoParams);
distanceInMeters = norm(point3d)/1000;
```

Display the detected face and distance.

```
distanceAsString = sprintf('%0.2f meters', distanceInMeters);
I1 = insertObjectAnnotation(I1,'rectangle',face1,distanceAsString,'FontSize',18);
I2 = insertObjectAnnotation(I2,'rectangle',face2, distanceAsString,'FontSize',18);
I1 = insertShape(I1,'FilledRectangle',face1);
I2 = insertShape(I2,'FilledRectangle',face2);

imshowpair(I1, I2, 'montage');
```

Reconstruct 3-D Scene from Disparity Map

Load the stereo parameters.

```
load('webcamsSceneReconstruction.mat');
```

Read in the stereo pair of images.

```
I1 = imread('sceneReconstructionLeft.jpg');
I2 = imread('sceneReconstructionRight.jpg');
```

Rectify the images.

```
[J1, J2] = rectifyStereoImages(I1,I2,stereoParams);
```

Display the images after rectification.

```
figure
imshow(cat(3,J1(:,:,1),J2(:,:,2:3)),'InitialMagnification',50);
```


Compute the disparity.

```
disparityMap = disparity(rgb2gray(J1), rgb2gray(J2));
figure
imshow(disparityMap,[0,64],'InitialMagnification',50);
```


Reconstruct the 3-D world coordinates of points corresponding to each pixel from the disparity map.

```
xyzPoints = reconstructScene(disparityMap,stereoParams);
```

Segment out a person located between 3.2 and 3.7 meters away from the camera.

```
Z = xyzPoints(:,:,3);
mask = repmat(Z > 3200 & Z < 3700,[1,1,3]);
J1(~mask) = 0;
imshow(J1,'InitialMagnification',50);
```

Visualize Stereo Pair of Camera Extrinsic Parameters

Specify calibration images.

```
imageDir = fullfile(toolboxdir('vision'),'visiondata', ...
    'calibration','stereo');
leftImages = imageDatastore(fullfile(imageDir,'left'));
rightImages = imageDatastore(fullfile(imageDir,'right'));
```

Detect the checkerboards.

```
[imagePoints,boardSize] = detectCheckerboardPoints(...
    leftImages.Files,rightImages.Files);
```

Specify world coordinates of checkerboard keypoints. Square size is in millimeters.

```
squareSize = 108;
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Calibrate the stereo camera system. Both cameras have the same resolution.

```
I = readimage(leftImages,1);
imageSize = [size(I, 1), size(I, 2)];
cameraParams = estimateCameraParameters(imagePoints,worldPoints, ...
                                'ImageSize',imageSize);
```

Visualize pattern locations.

```
figure;
showExtrinsics(cameraParams);
```
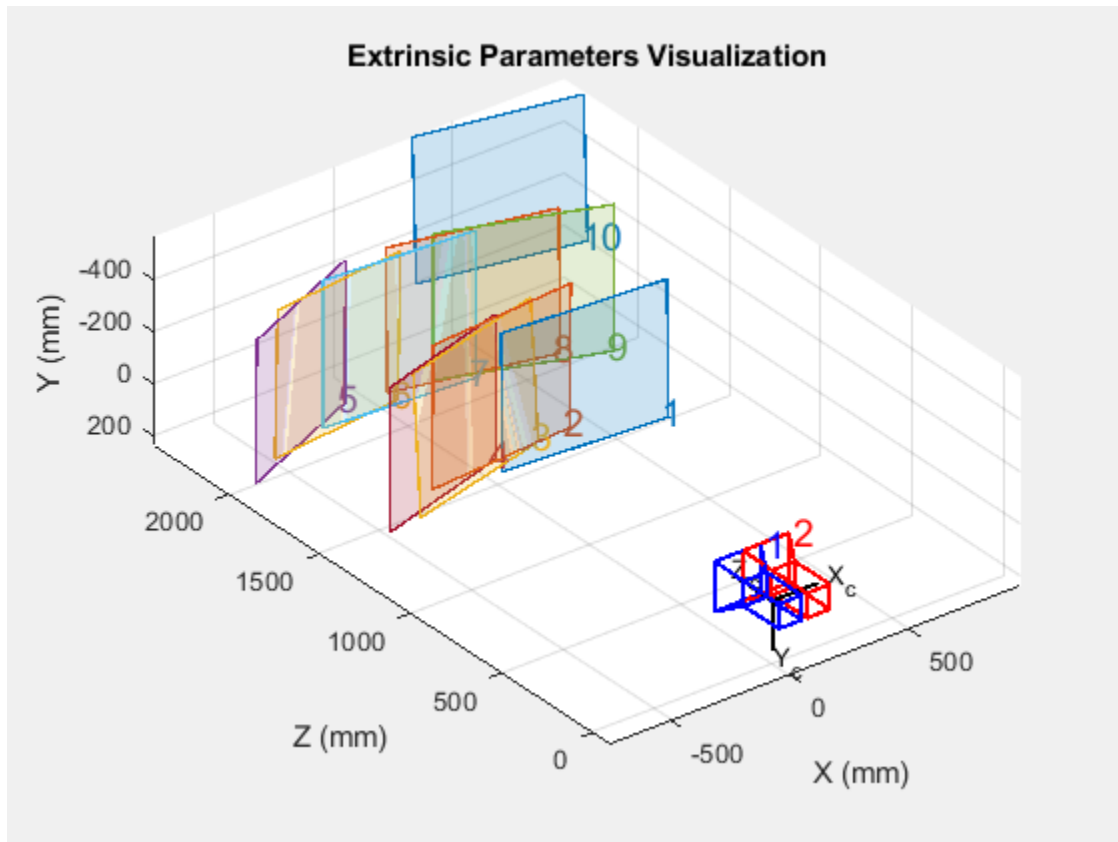
Visualize camera locations.

```
figure;
showExtrinsics(cameraParams,'patternCentric');
```
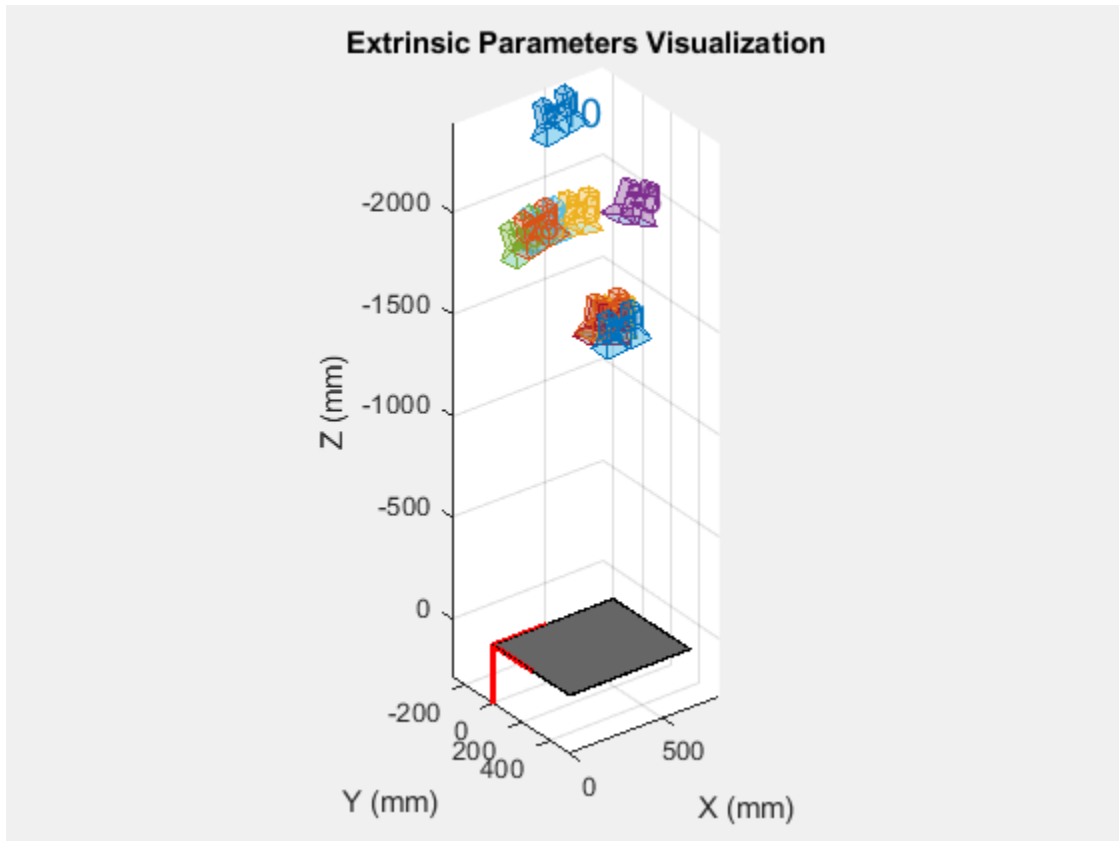
Read Point Cloud from a PLY File

```
ptCloud = pcread('teapot.ply');
pcshow(ptCloud);
```
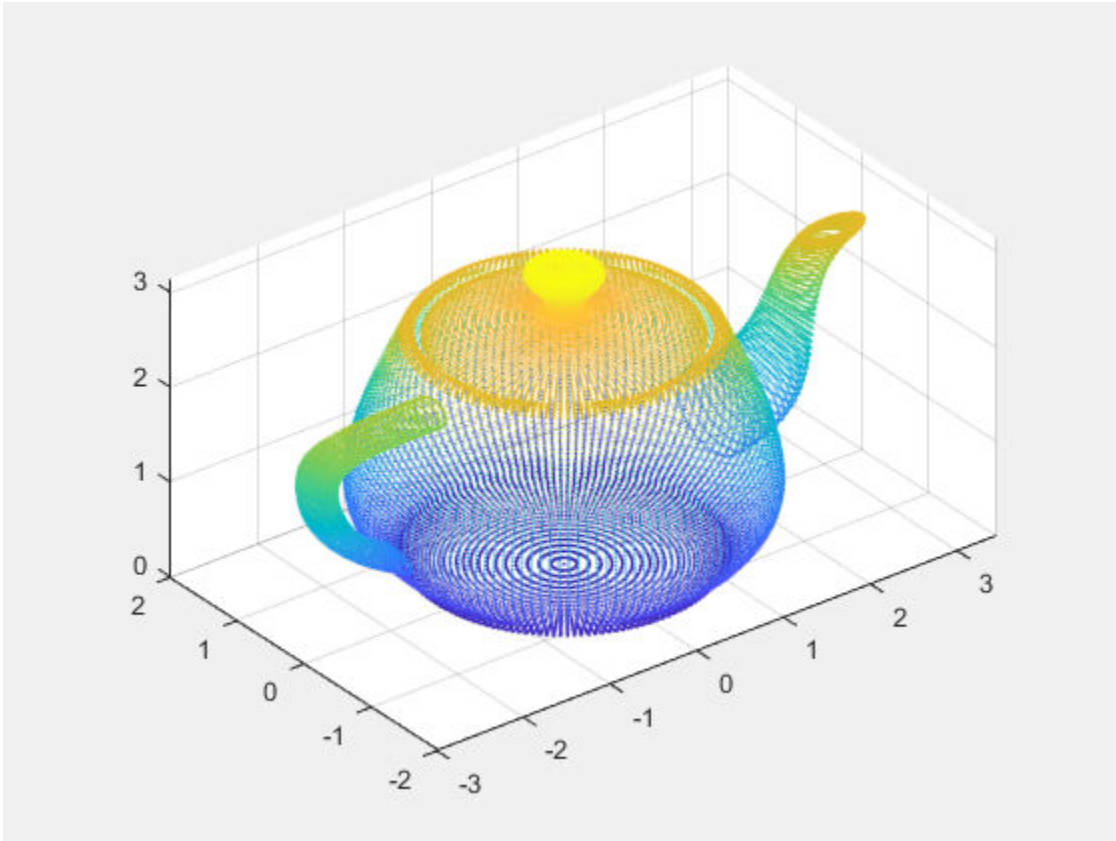
Write 3-D Point Cloud to PLY File

```
ptCloud = pcread('teapot.ply');
pcshow(ptCloud);
```
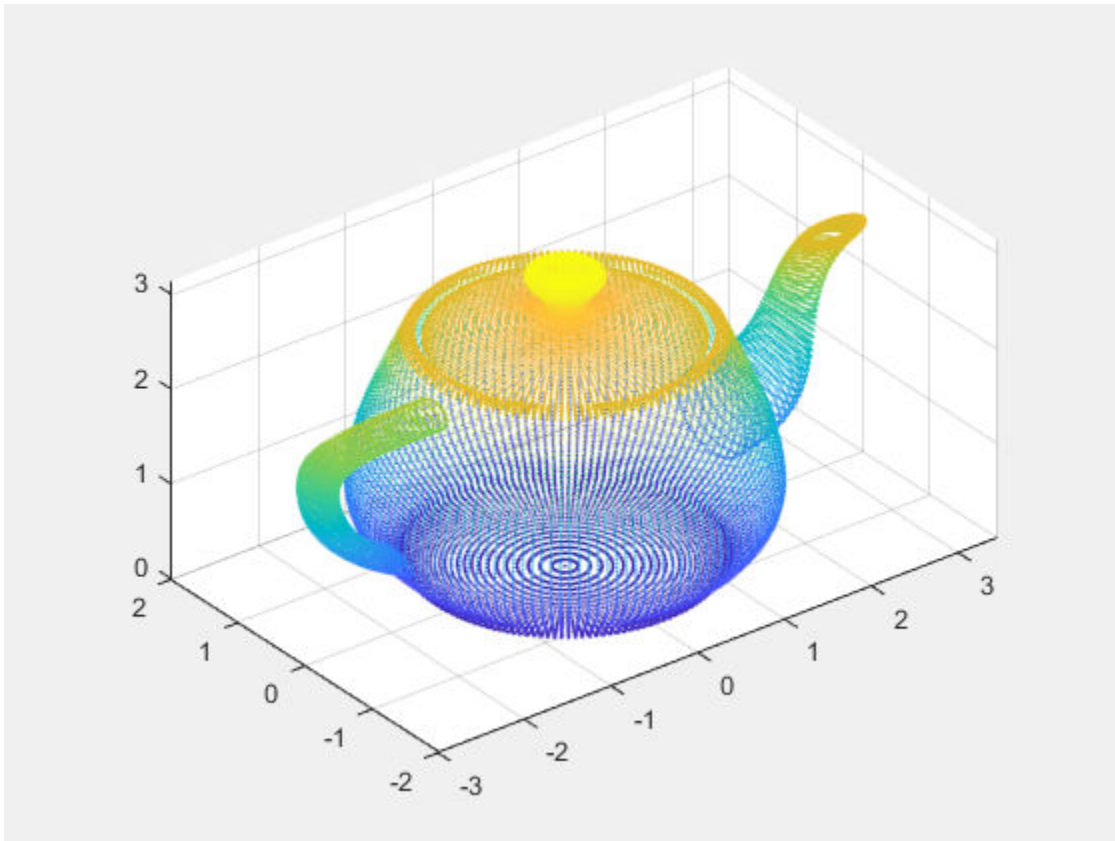


```
pcwrite(ptCloud,'teapotOut','PLYFormat','binary');
```

Visualize the Difference Between Two Point Clouds

Load two point clouds that were captured using a Kinect device in a home setting.

```
load('livingRoom');

pc1 = livingRoomData{1};
pc2 = livingRoomData{2};
```

Plot and set the viewpoint of point clouds.

```
figure
pcshowpair(pc1,pc2,'VerticalAxis','Y','VerticalAxisDir','Down')
title('Difference Between Two Point Clouds')
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
```

Difference Between Two Point Clouds
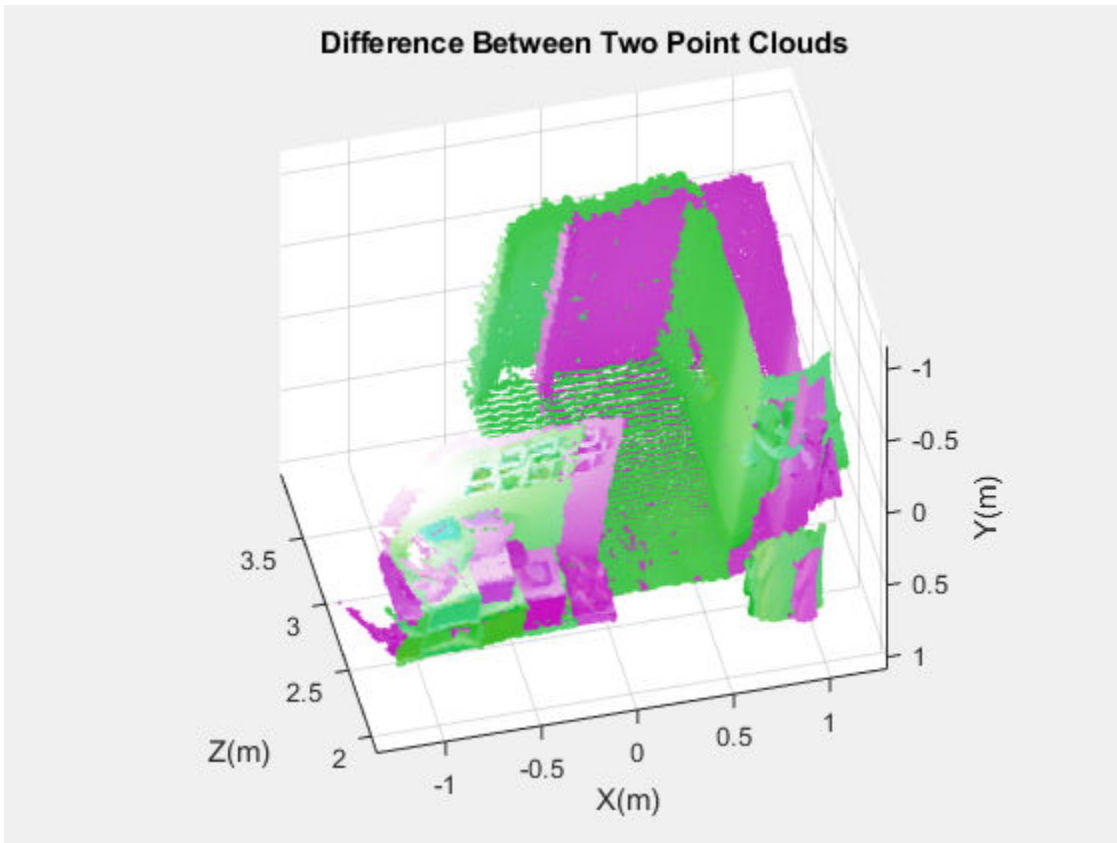
View Rotating 3-D Point Cloud

Load point cloud.

```
ptCloud = pcread('teapot.ply');
```

Define a rotation matrix and 3-D transform.

```
x = pi/180;
R = [ cos(x) sin(x) 0 0
     -sin(x) cos(x) 0 0
      0       0   1 0
      0       0   0 1];

tform = affine3d(R);
```

Compute *x*-_y_ limits that ensure that the rotated teapot is not clipped.

```
lower = min([ptCloud.XLimits ptCloud.YLimits]);
upper = max([ptCloud.XLimits ptCloud.YLimits]);

xlimits = [lower upper];
ylimits = [lower upper];
zlimits = ptCloud.ZLimits;
```

Create the player and customize player axis labels.

```
player = pcplayer(xlimits,ylimits,zlimits);

xlabel(player.Axes,'X (m)');
ylabel(player.Axes,'Y (m)');
zlabel(player.Axes,'Z (m)');
```
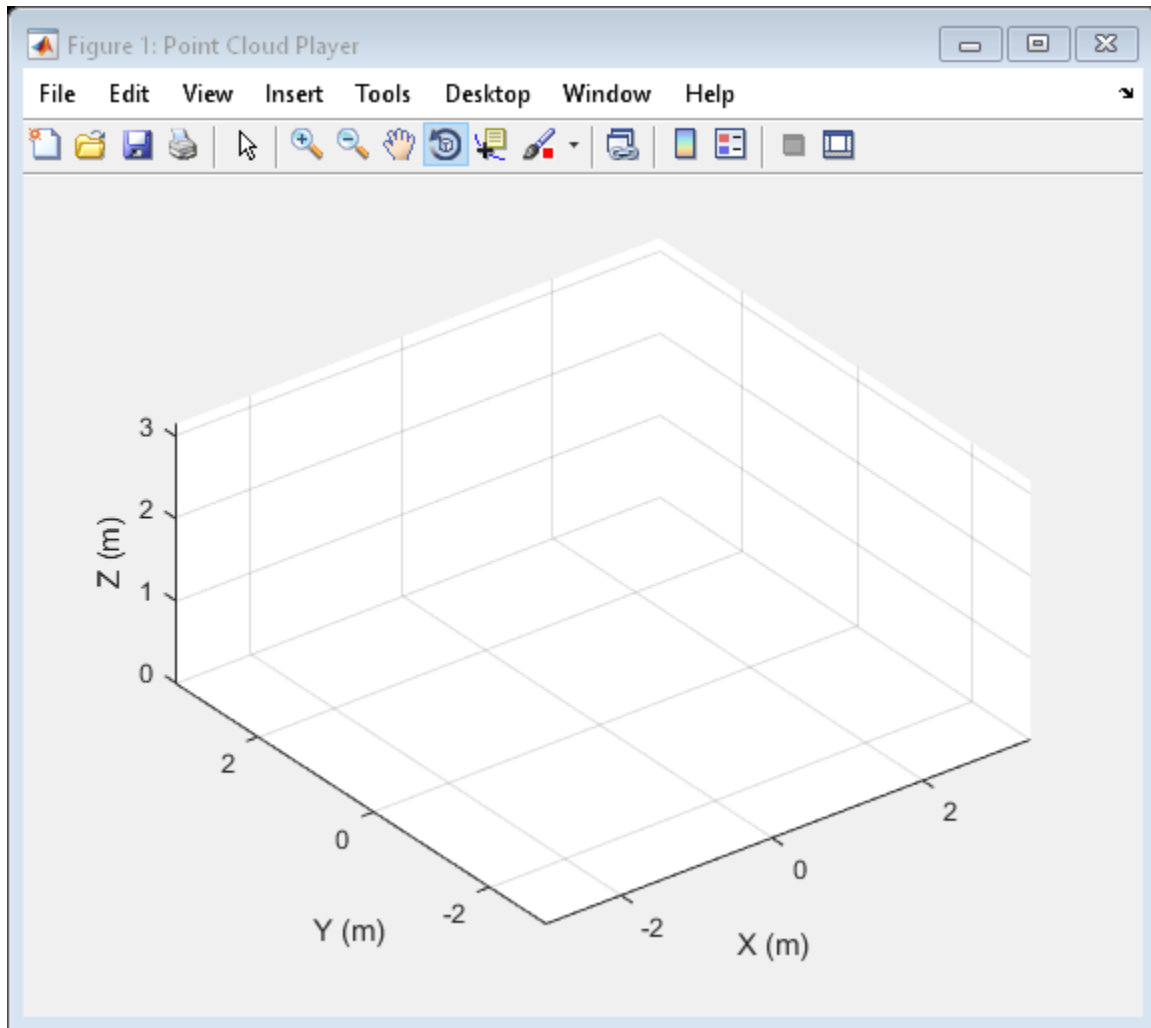
Rotate the teapot around the *z*-axis.

```
for i = 1:360
    ptCloud = pctransform(ptCloud,tform);
    view(player,ptCloud);
end
```
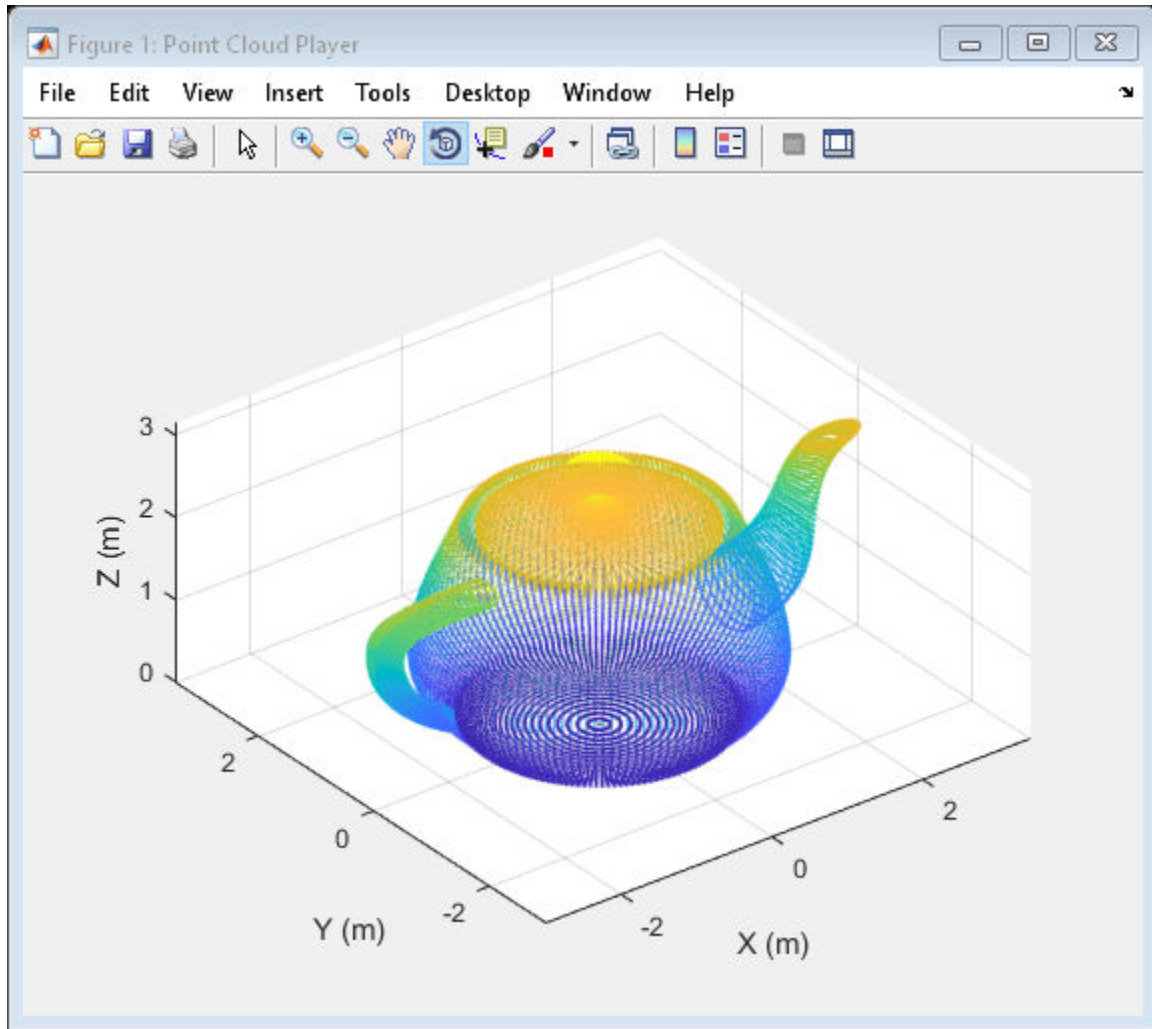
Hide and Show 3-D Point Cloud Figure

Load point cloud.

```
ptCloud = pcread('teapot.ply');
```

Create the player and customize player axis labels.

```
player = pcplayer(ptCloud.XLimits,ptCloud.YLimits,ptCloud.ZLimits);
```
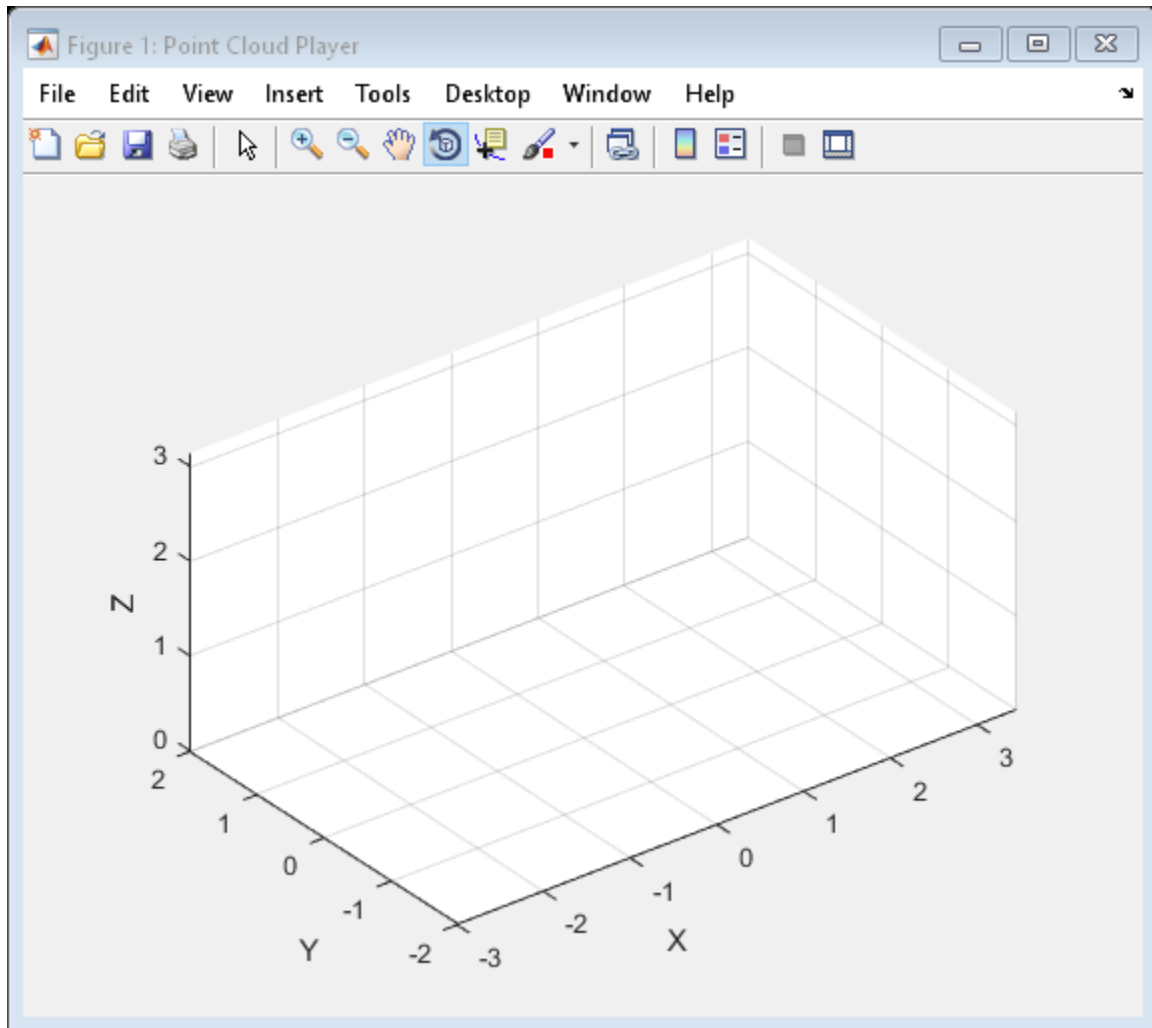
Hide figure.

```
hide(player)
```

Show figure.

```
show(player)
view(player,ptCloud);
```
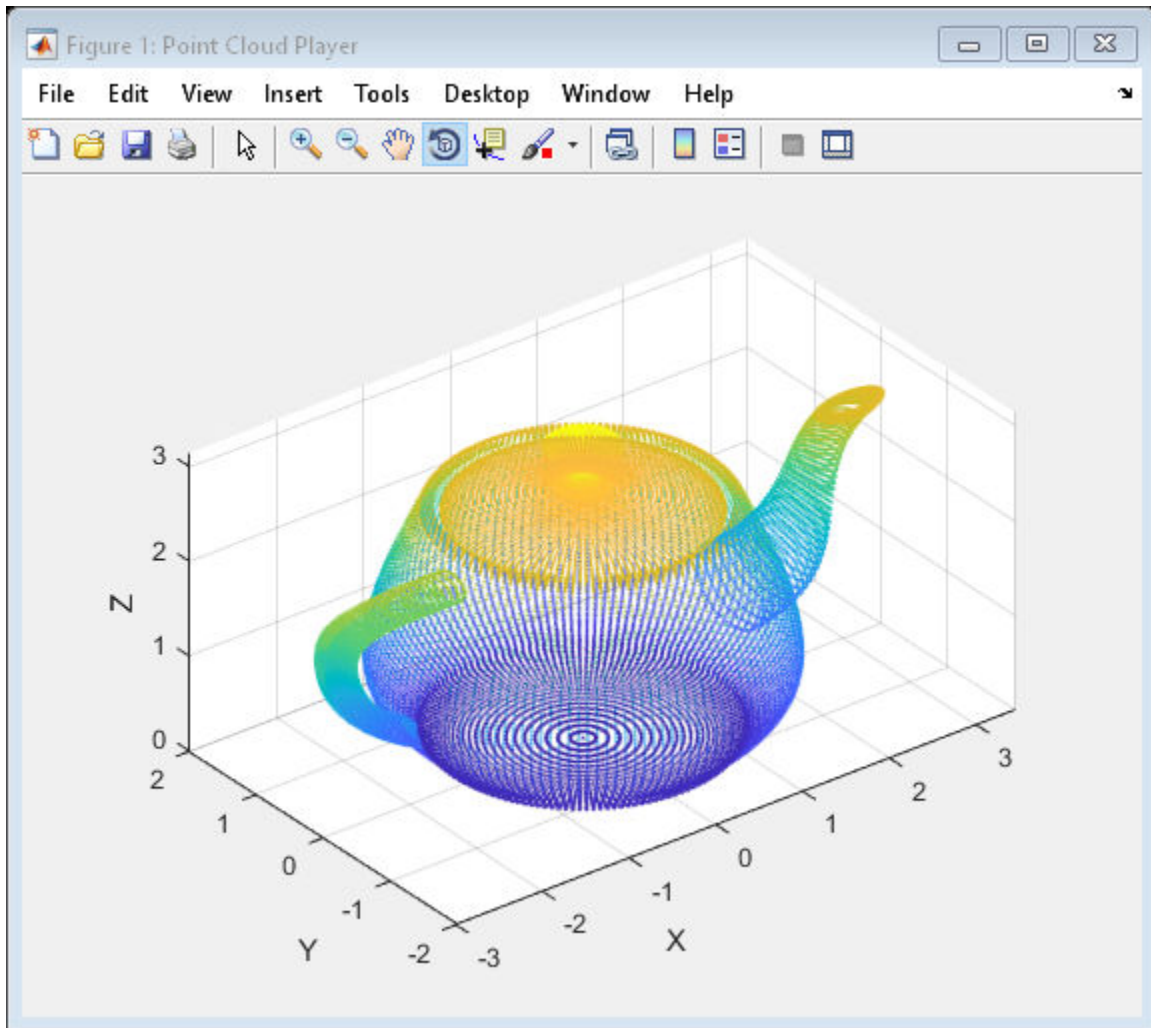
Align Two Point Clouds Using ICP Algorithm

Load point cloud data.

```
ptCloud = pcread('teapot.ply');
```

```
pcshow(ptCloud);
title('Teapot');
```


Teapot
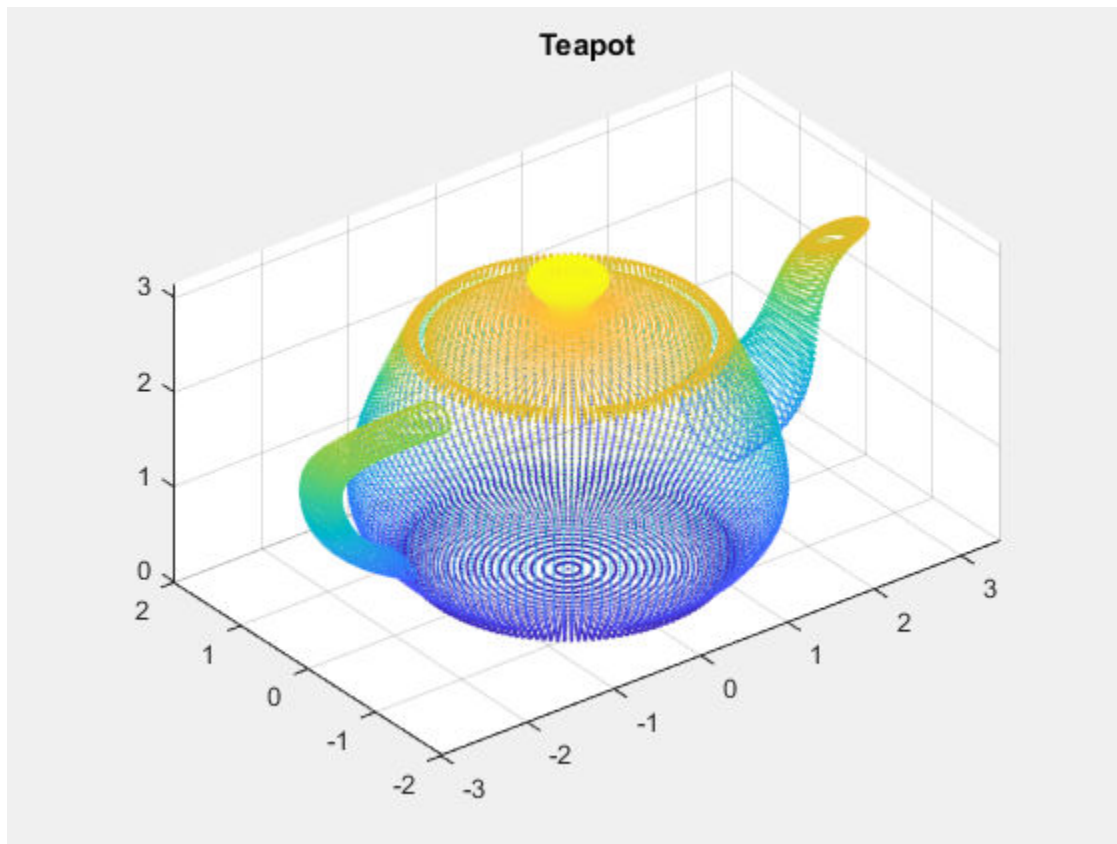
Create a transform object with 30 degree rotation along z -axis and translation [5,5,10].

```
A = [cos(pi/6) sin(pi/6) 0 0; ...
    -sin(pi/6) cos(pi/6) 0 0; ...
            0         0 1 0; ...
```

```
              5              5 10 1];
tform1 = affine3d(A);
```

Transform the point cloud.

```
ptCloudTformed = pctransform(ptCloud,tform1);

pcshow(ptCloudTformed);
title('Transformed Teapot');
```
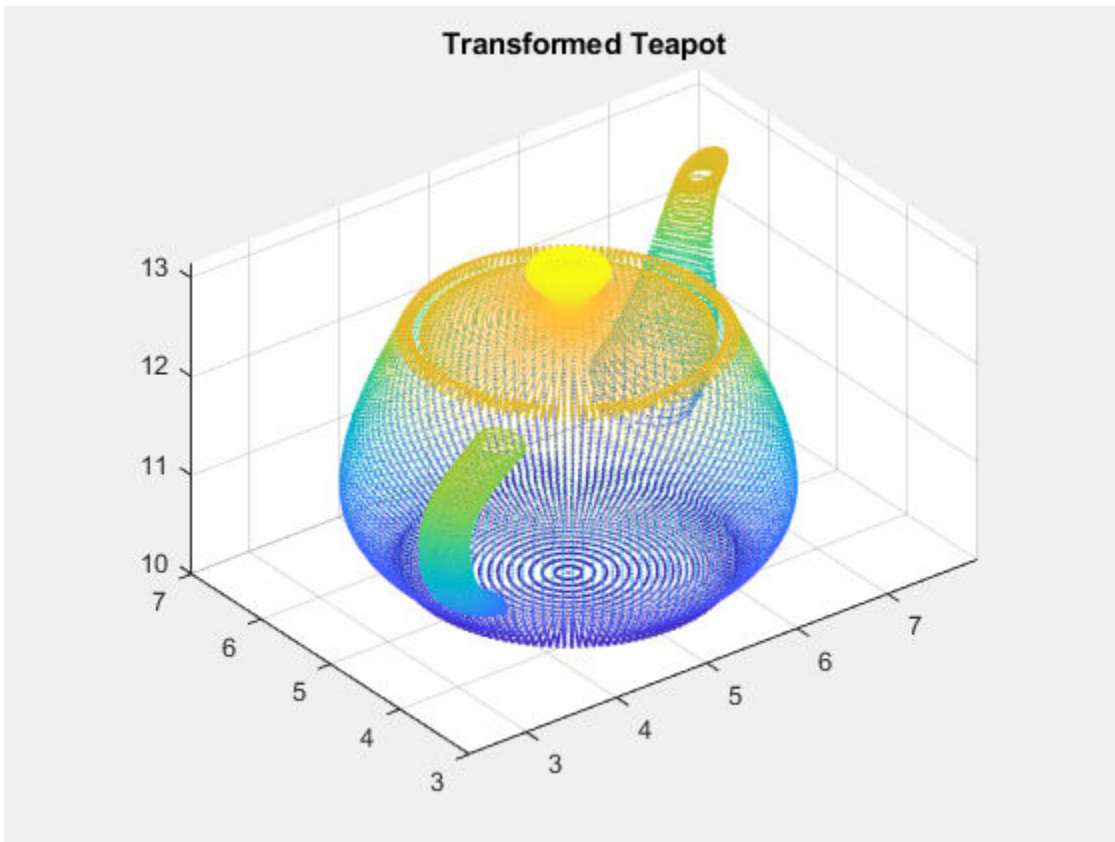

Apply the rigid registration.

```
tform = pcregistericp(ptCloudTformed,ptCloud,'Extrapolate',true);
```

Compare the result with the true transformation.

```
disp(tform1.T);

    0.8660    0.5000         0         0
   -0.5000    0.8660         0         0
         0         0    1.0000         0
    5.0000    5.0000   10.0000    1.0000
```

```
tform2 = invert(tform);
disp(tform2.T);

    0.8660    0.5000    0.0000         0
   -0.5000    0.8660    0.0000         0
    0.0000   -0.0000    1.0000         0
    5.0000    5.0000   10.0000    1.0000
```

Rotate 3-D Point Cloud

Read a point cloud.

```
ptCloud = pcread('teapot.ply');
```

Plot the original data.

```
figure
pcshow(ptCloud);
xlabel('X');
ylabel('Y');
zlabel('Z');
```
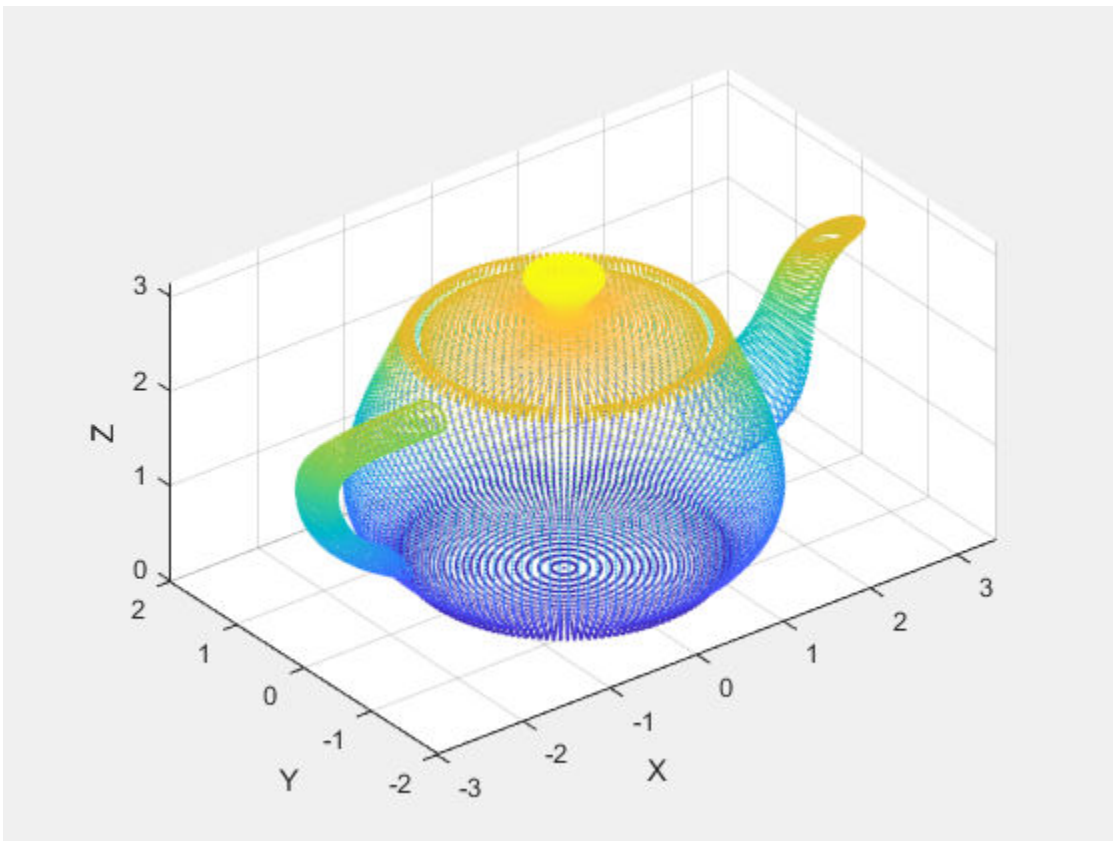
Create a transform object with a 45 degrees rotation along the z-axis.

```
A = [cos(pi/4) sin(pi/4) 0 0; ...
     -sin(pi/4) cos(pi/4) 0 0; ...
     0 0 1 0; ...
     0 0 0 1];
tform = affine3d(A);
```

Transform the point cloud.

```
ptCloudOut = pctransform(ptCloud,tform);
```

Plot the transformed point cloud.

```
figure
pcshow(ptCloudOut);
xlabel('X');
ylabel('Y');
zlabel('Z');
```
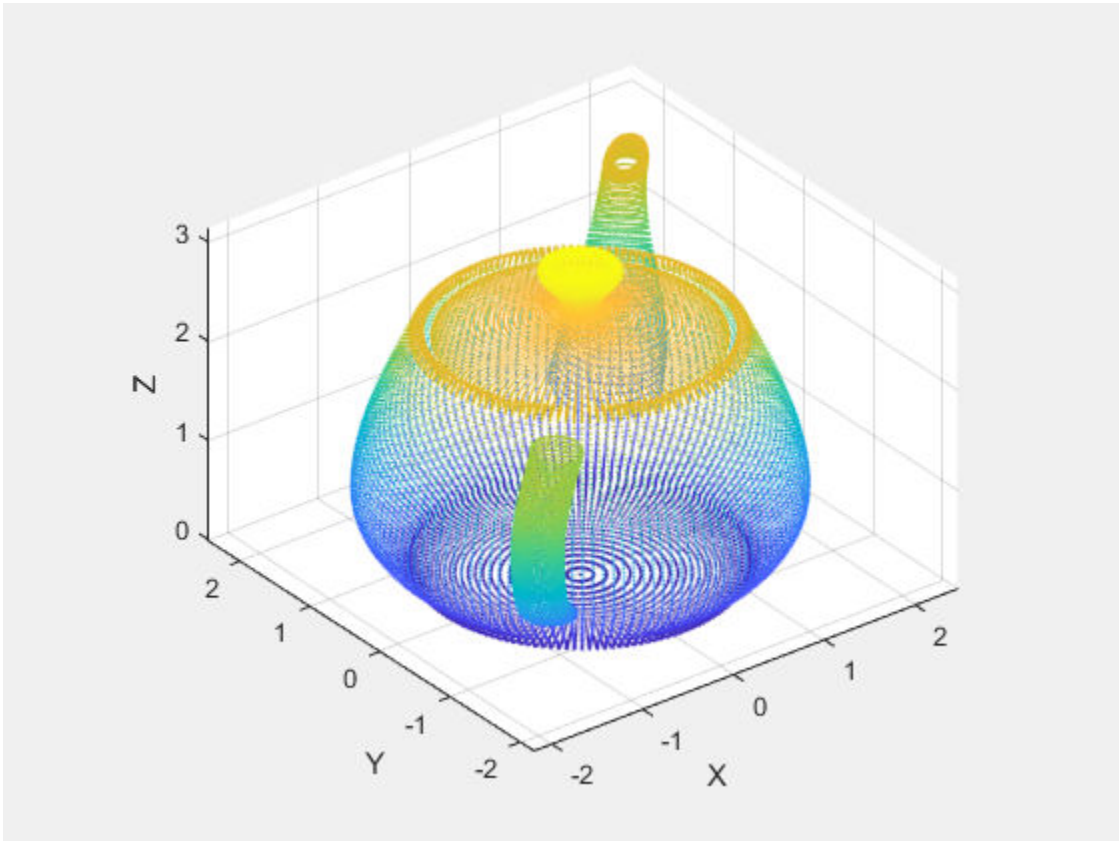
Merge Two Identical Point Clouds Using Box Grid Filter

Create two identical point clouds.

```
ptCloudA = pointCloud(100*rand(1000,3));
ptCloudB = copy(ptCloudA);
```

Merge the two point clouds.

```
ptCloud = pcmerge(ptCloudA,ptCloudB,1);
pcshow(ptCloud);
```
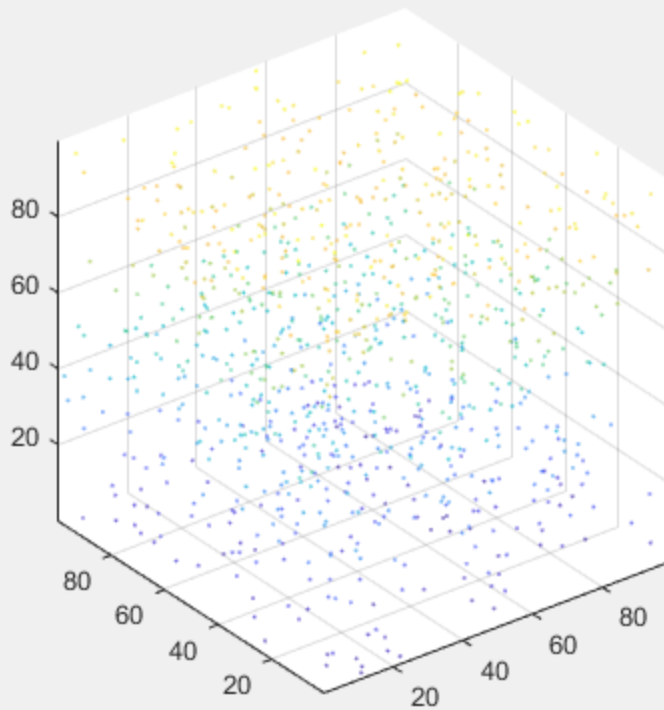
Extract Cylinder from Point Cloud

Load the point cloud.

```
load('object3d.mat');
```

Display the point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Original Point Cloud')
```
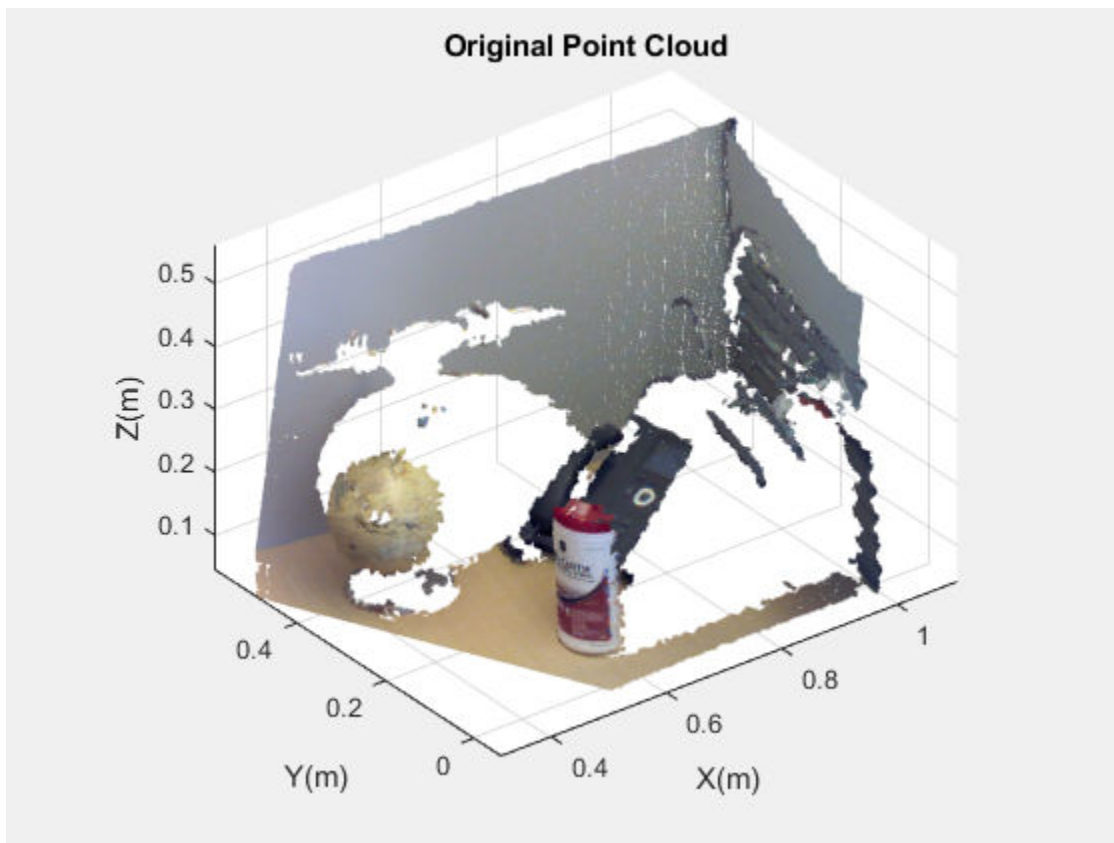
Set the maximum point-to-cylinder distance (5 mm) for cylinder fitting.

```
maxDistance = 0.005;
```

Set the region of interest to constrain the search.

```
roi = [0.4,0.6,-inf,0.2,0.1,inf];
sampleIndices = findPointsInROI(ptCloud,roi);
```

Set the orientation constraint.

```
referenceVector = [0,0,1];
```

Detect the cylinder and extract it from the point cloud by specifying the inlier points.

```
[model,inlierIndices] = pcfitcylinder(ptCloud,maxDistance,...
        referenceVector,'SampleIndices',sampleIndices);
pc = select(ptCloud,inlierIndices);
```

Plot the extracted cylinder.

```
figure
pcshow(pc)
title('Cylinder Point Cloud')
```

Cylinder Point Cloud
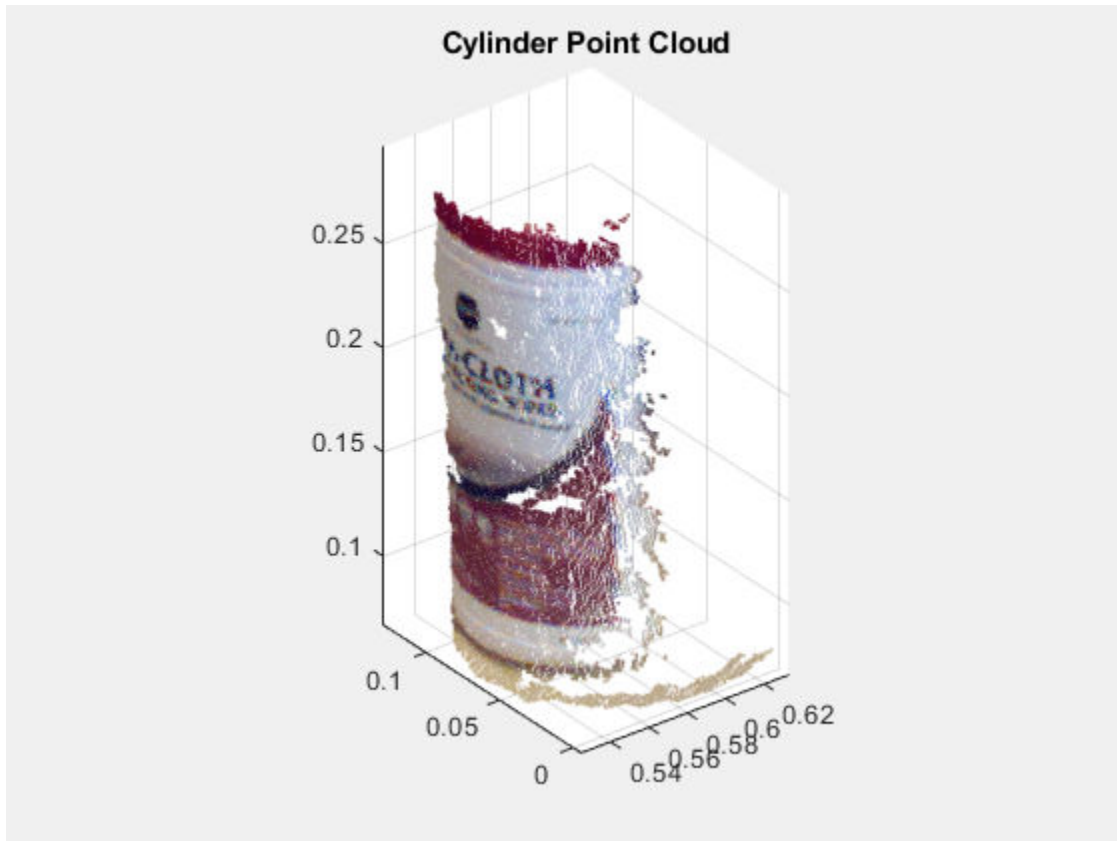
Detect Multiple Planes from Point Cloud

Load the point cloud.

```
load('object3d.mat')
```

Display and label the point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Original Point Cloud')
```
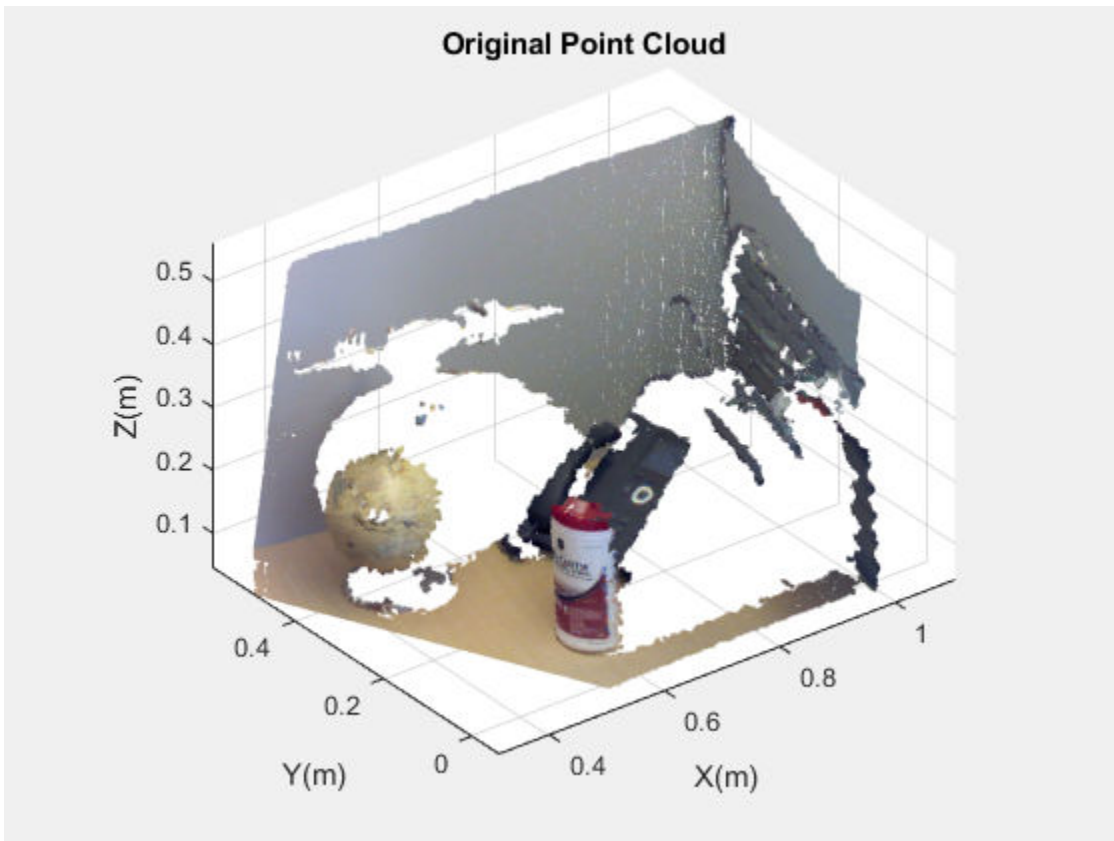
Set the maximum point-to-plane distance (2cm) for plane fitting.

```
maxDistance = 0.02;
```

Set the normal vector of the plane.

```
referenceVector = [0,0,1];
```

Set the maximum angular distance to 5 degrees.

```
maxAngularDistance = 5;
```

Detect the first plane, the table, and extract it from the point cloud.

```
[model1,inlierIndices,outlierIndices] = pcfitplane(ptCloud,...
            maxDistance,referenceVector,maxAngularDistance);
plane1 = select(ptCloud,inlierIndices);
remainPtCloud = select(ptCloud,outlierIndices);
```

Set the region of interest to constrain the search for the second plane, left wall.

```
roi = [-inf,inf;0.4,inf;-inf,inf];
sampleIndices = findPointsInROI(remainPtCloud,roi);
```

Detect the left wall and extract it from the remaining point cloud.

```
[model2,inlierIndices,outlierIndices] = pcfitplane(remainPtCloud,...
            maxDistance,'SampleIndices',sampleIndices);
plane2 = select(remainPtCloud,inlierIndices);
remainPtCloud = select(remainPtCloud,outlierIndices);
```

Plot the two planes and the remaining points.

```
figure
pcshow(plane1)
title('First Plane')
```

First Plane
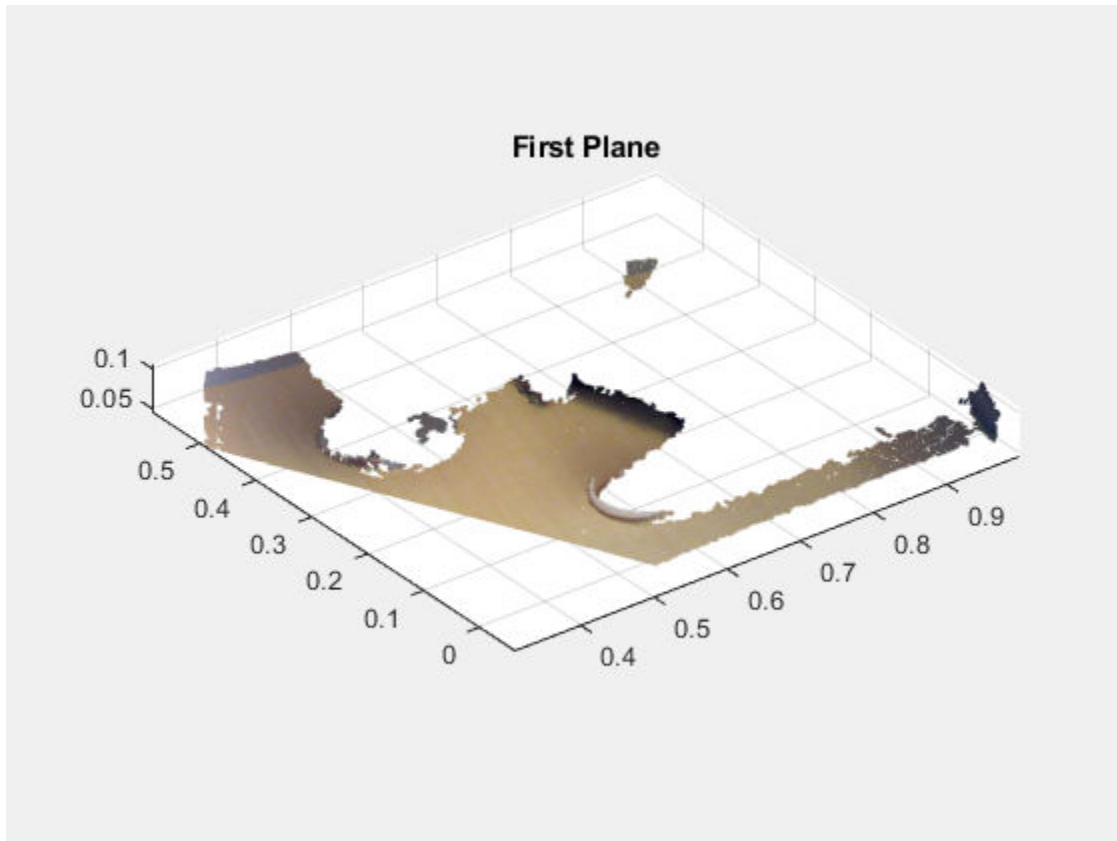
```
figure
pcshow(plane2)
title('Second Plane')
```

Second Plane
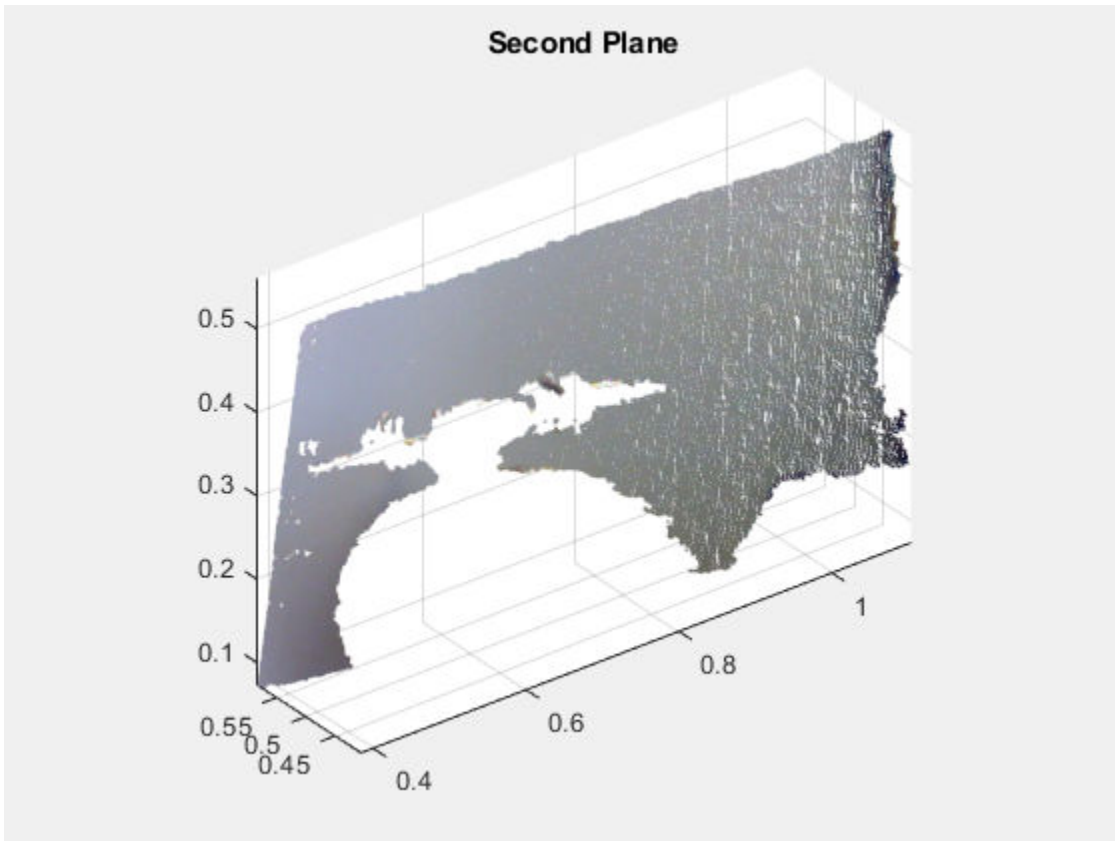
```
figure
pcshow(remainPtCloud)
title('Remaining Point Cloud')
```

Remaining Point Cloud
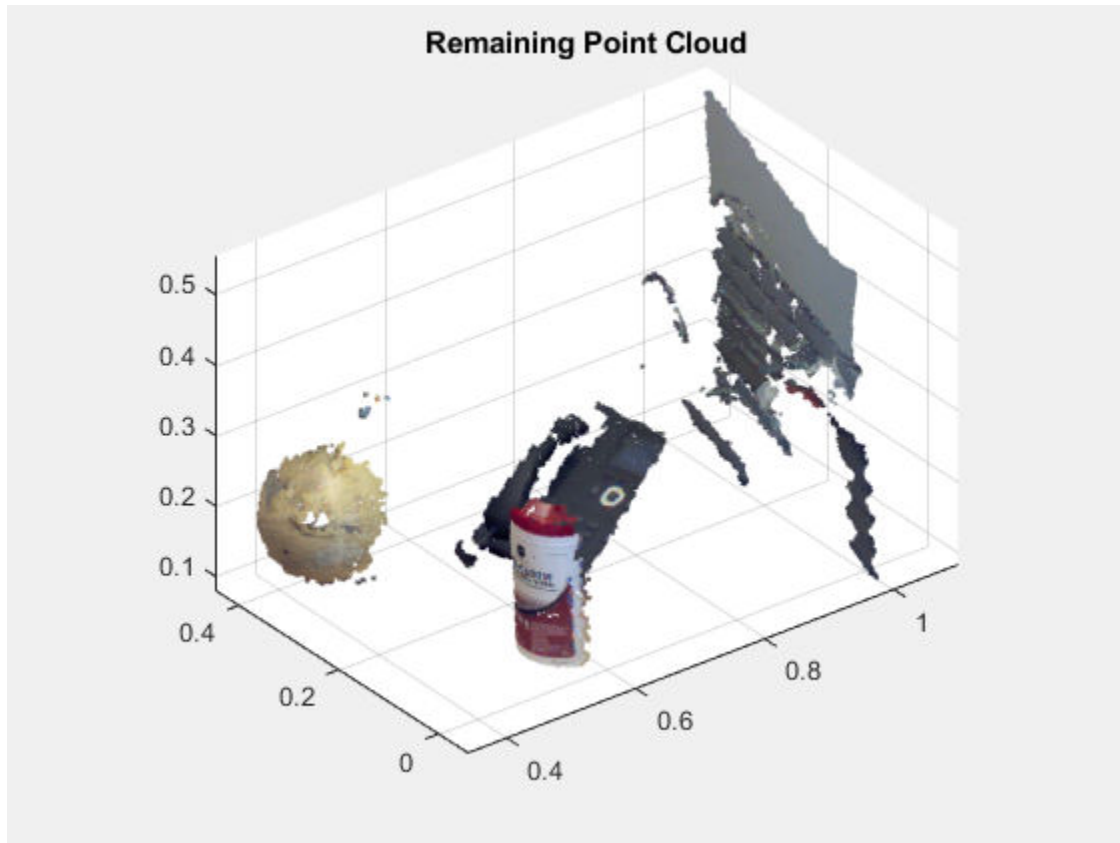
Detect Sphere from Point Cloud

Load data file.

```
load('object3d.mat');
```

Display original point cloud.

```
figure
pcshow(ptCloud)
xlabel('X(m)')
ylabel('Y(m)')
zlabel('Z(m)')
title('Original Point Cloud')
```
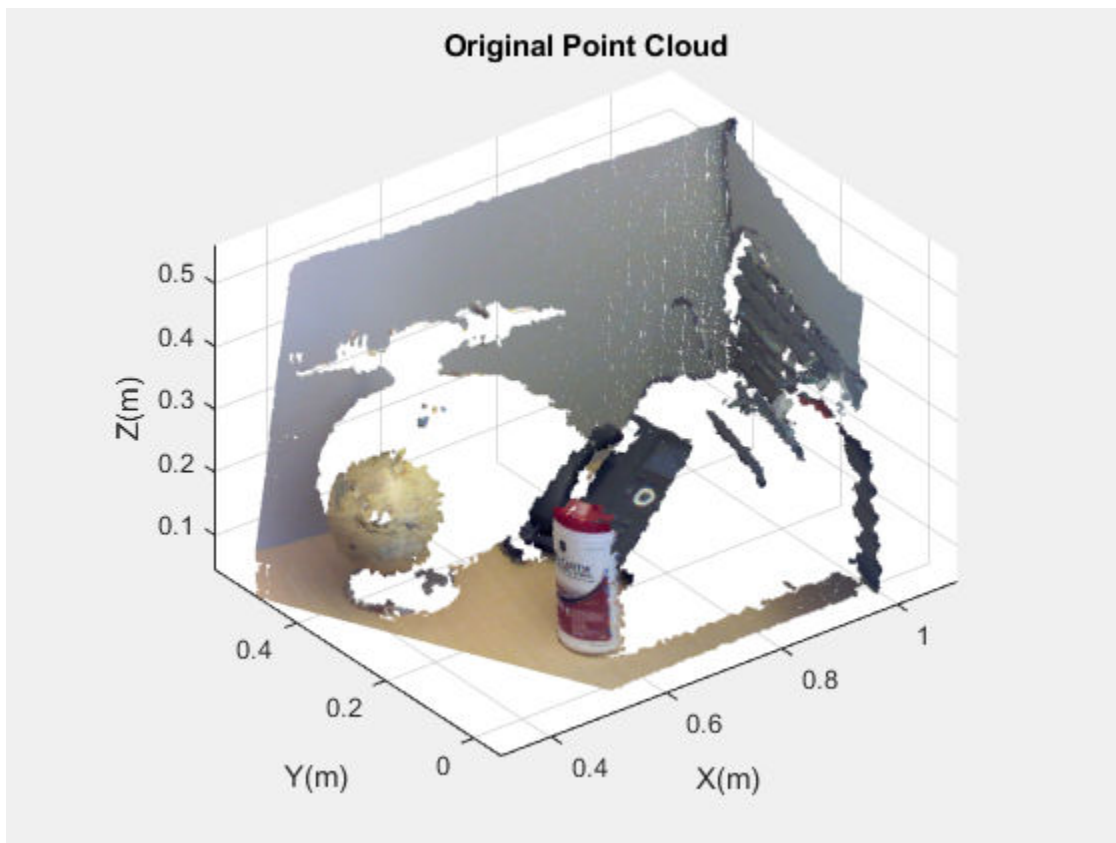
Set a maximum point-to-sphere distance of 1cm for sphere fitting.

```
maxDistance = 0.01;
```

Set the roi to constrain the search.

```
roi = [-inf,0.5,0.2,0.4,0.1,inf];
sampleIndices = findPointsInROI(ptCloud,roi);
```

Detect the sphere, a globe, and extract it from the point cloud.

```
[model,inlierIndices] = pcfitsphere(ptCloud,maxDistance,...
            'SampleIndices',sampleIndices);
globe = select(ptCloud,inlierIndices);
```

Plot the globe.

```
hold on
plot(model)
```
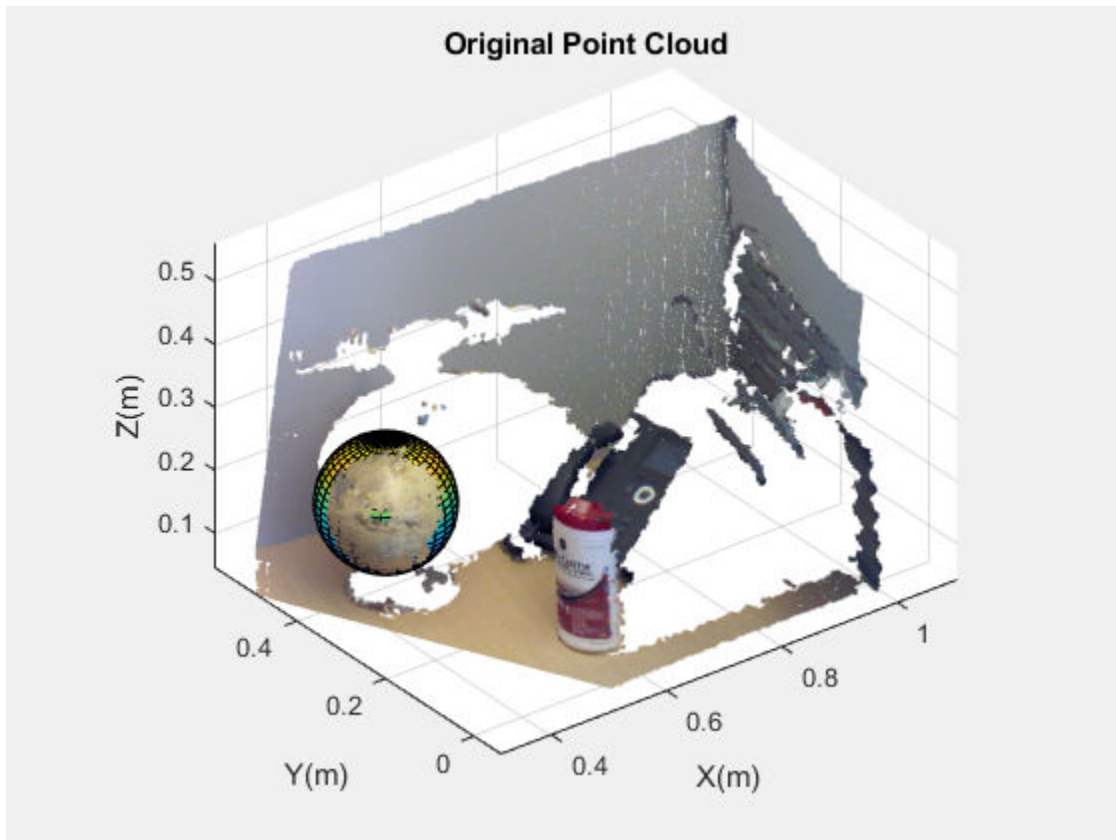
```
figure
pcshow(globe)
title('Globe Point Cloud')
```
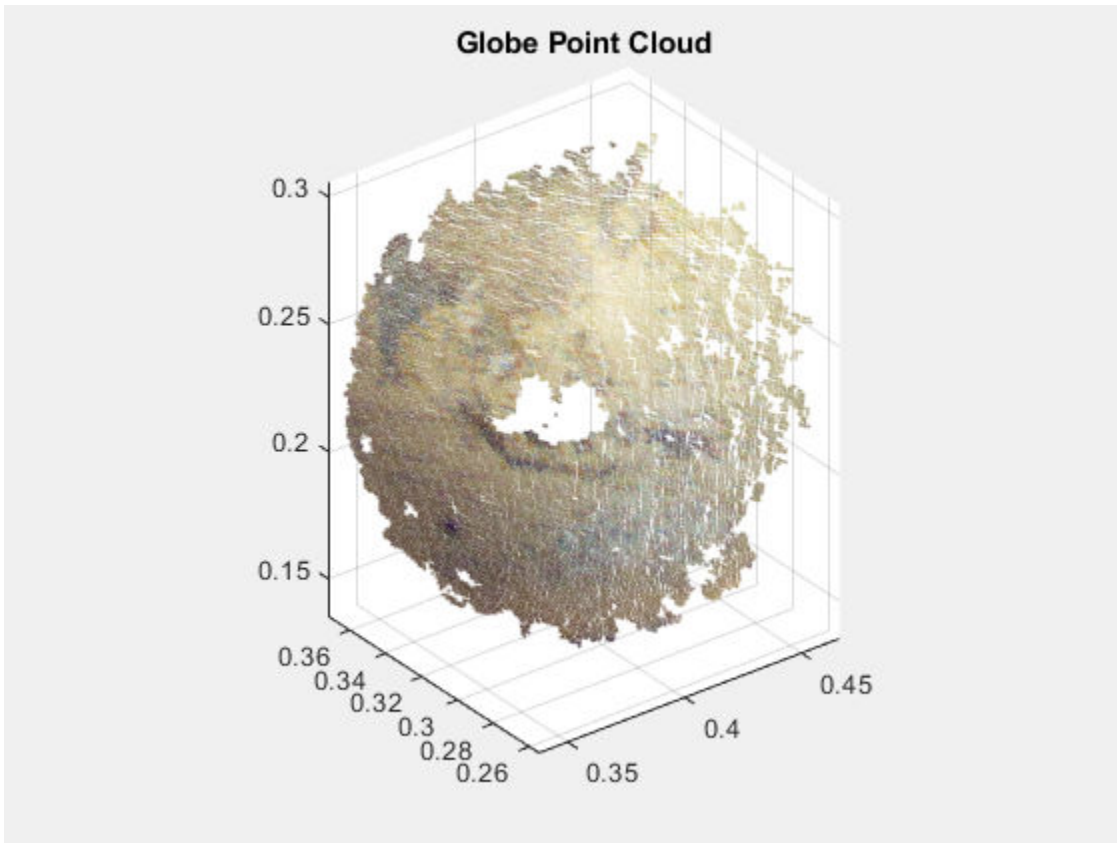
Globe Point Cloud

Remove Outliers from Noisy Point Cloud

Create a plane point cloud.

```
gv = 0:0.01:1;
[X,Y] = meshgrid(gv,gv);
ptCloud = pointCloud([X(:),Y(:),0.5*ones(numel(X),1)]);

figure
pcshow(ptCloud);
title('Original Data');
```


Add uniformly distributed random noise.

```
noise = rand(500, 3);
ptCloudA = pointCloud([ptCloud.Location; noise]);

figure
pcshow(ptCloudA);
title('Noisy Data');
```
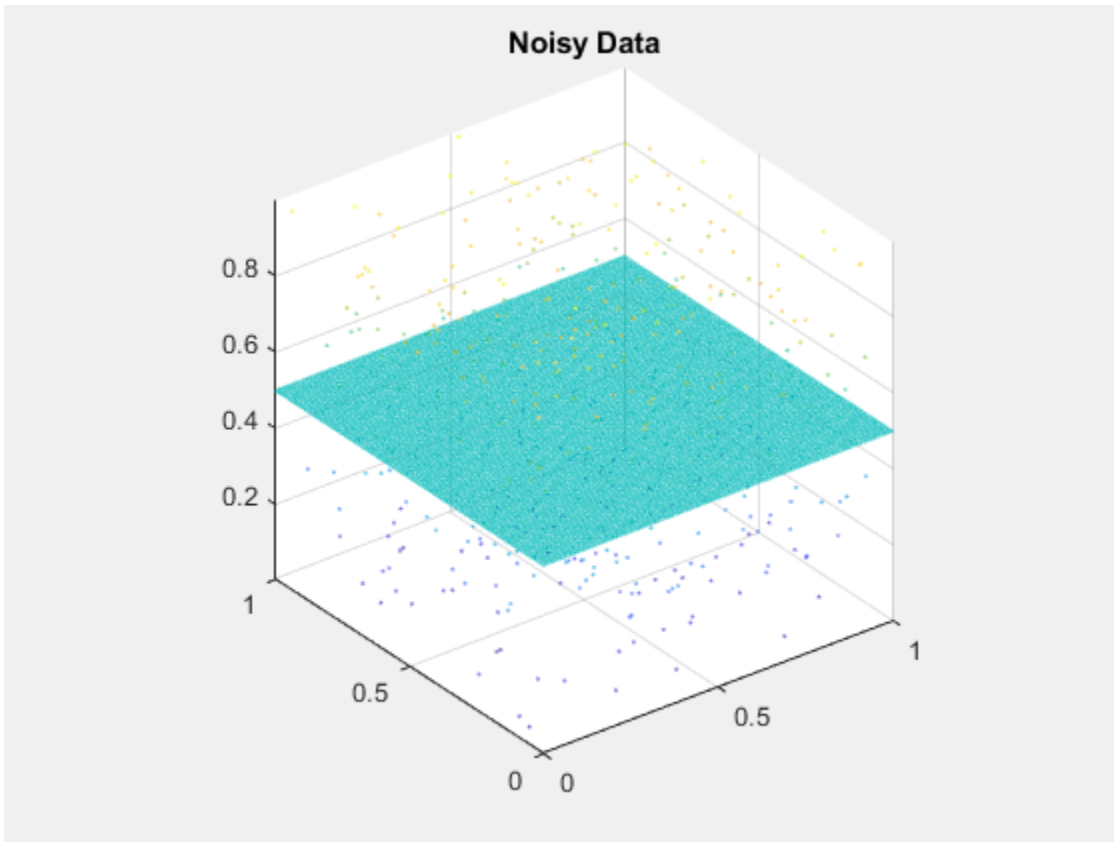

Remove outliers.

```
ptCloudB = pcdenoise(ptCloudA);

figure;
pcshow(ptCloudB);
title('Denoised Data');
```

Denoised Data
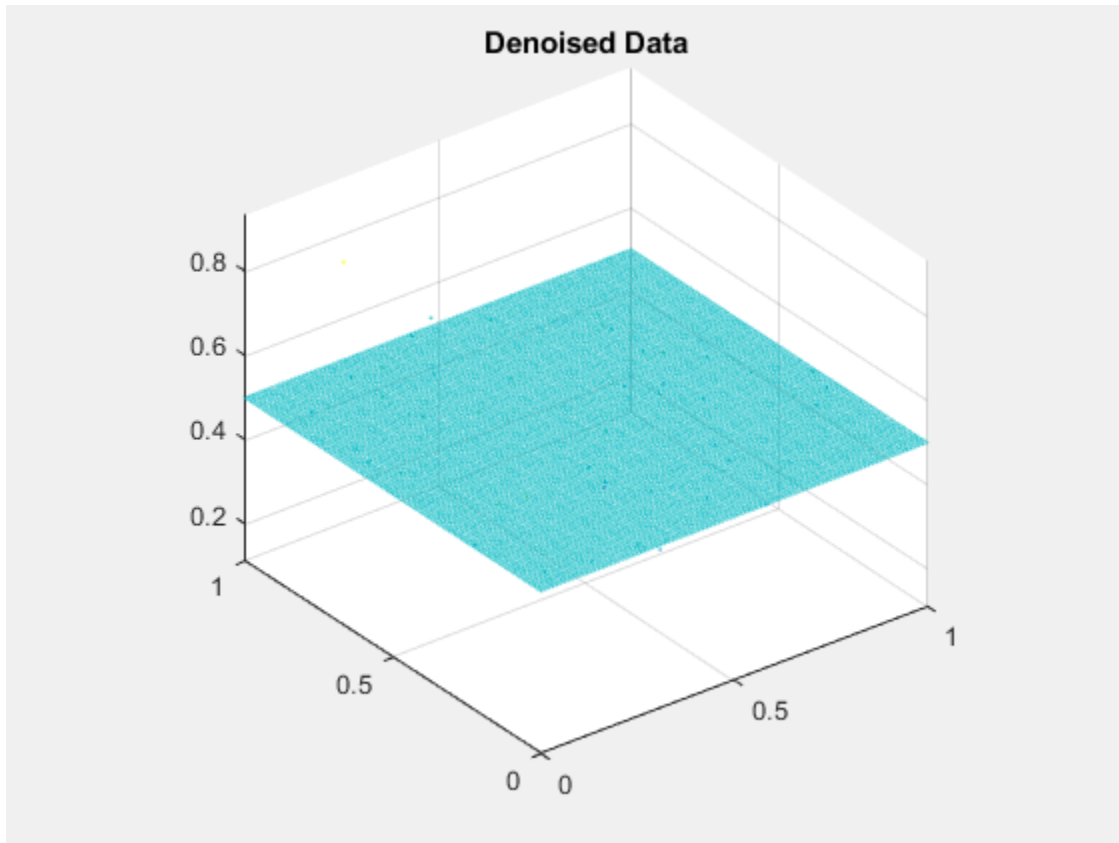
Downsample Point Cloud Using Box Grid Filter

Read a point cloud.

```
ptCloud = pcread('teapot.ply');
```

Set the 3-D resolution to be (0.1 x 0.1 x 0.1).

```
gridStep = 0.1;
ptCloudA = pcdownsample(ptCloud,'gridAverage',gridStep);
```

Visualize the downsampled data.

```
figure;
pcshow(ptCloudA);
```
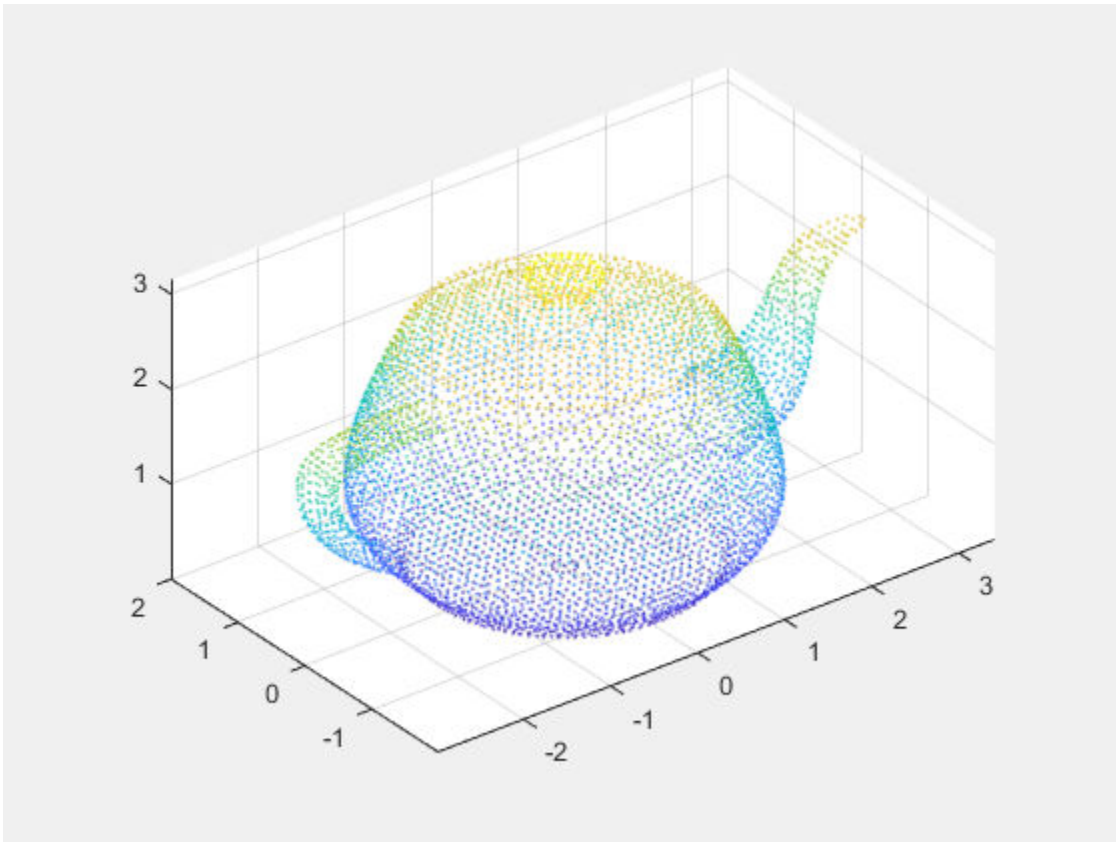
Compare the point cloud to data that is downsampled using a fixed step size.

```
stepSize = floor(ptCloud.Count/ptCloudA.Count);
indices = 1:stepSize:ptCloud.Count;
ptCloudB = select(ptCloud, indices);

figure;
pcshow(ptCloudB);
```

Measure Distance from Stereo Camera to a Face

Load stereo parameters.

```
load('webcamsSceneReconstruction.mat');
```

Read in the stereo pair of images.

```
I1 = imread('sceneReconstructionLeft.jpg');
I2 = imread('sceneReconstructionRight.jpg');
```

Undistort the images.

```
I1 = undistortImage(I1,stereoParams.CameraParameters1);
I2 = undistortImage(I2,stereoParams.CameraParameters2);
```

Detect a face in both images.

```
faceDetector = vision.CascadeObjectDetector;
face1 = faceDetector(I1);
face2 = faceDetector(I2);
```

Find the center of the face.

```
center1 = face1(1:2) + face1(3:4)/2;
center2 = face2(1:2) + face2(3:4)/2;
```

Compute the distance from camera 1 to the face.

```
point3d = triangulate(center1, center2, stereoParams);
distanceInMeters = norm(point3d)/1000;
```

Display the detected face and distance.

```
distanceAsString = sprintf('%0.2f meters', distanceInMeters);
I1 = insertObjectAnnotation(I1,'rectangle',face1,distanceAsString,'FontSize',18);
I2 = insertObjectAnnotation(I2,'rectangle',face2, distanceAsString,'FontSize',18);
I1 = insertShape(I1,'FilledRectangle',face1);
I2 = insertShape(I2,'FilledRectangle',face2);

imshowpair(I1, I2, 'montage');
```

Remove Motion Artifacts From Image

Create a deinterlacer object.

```
hdinterlacer = vision.Deinterlacer;
```

Read an image with motion artifacts.

```
I = imread('vipinterlace.png');
```

Apply the deinterlacer to the image.

```
clearimage = hdinterlacer(I);
```

Display the results.

```
imshow(I);
title('Original Image');
```

Original Image


```
figure, imshow(clearimage);
title('Image after deinterlacing');
```

Image after deinterlacing

Find Vertical and Horizontal Edges in Image

Construct Haar-like wavelet filters to find vertical and horizontal edges in an image.

Read the input image and compute the integral image.

```
I = imread('pout.tif');
intImage = integralImage(I);
```

Construct Haar-like wavelet filters. Use the dot notation to find the vertical filter from the horizontal filter.

```
horiH = integralKernel([1 1 4 3; 1 4 4 3],[-1, 1]);
vertH = horiH.'

vertH =
  integralKernel with properties:

    BoundingBoxes: [2x4 double]
          Weights: [-1 1]
     Coefficients: [4x6 double]
           Center: [2 3]
             Size: [4 6]
      Orientation: 'upright'
```

Display the horizontal filter.

```
imtool(horiH.Coefficients, 'InitialMagnification','fit');
```

Compute the filter responses.

```
horiResponse = integralFilter(intImage,horiH);
vertResponse = integralFilter(intImage,vertH);
```

Display the results.

```
figure;
imshow(horiResponse,[]);
title('Horizontal edge responses');
```

Horizontal edge responses


```
figure;
imshow(vertResponse,[]);
title('Vertical edge responses');
```

Vertical edge responses

Single Camera Calibration

Create a set of calibration images.

```
images = imageSet(fullfile(toolboxdir('vision'),'visiondata',...
              'calibration','mono'));
imageFileNames = images.ImageLocation;
```

Detect the calibration pattern.

```
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);
```

Generate the world coordinates of the corners of the squares.

```
squareSizeInMM = 29;
worldPoints = generateCheckerboardPoints(boardSize,squareSizeInMM);
```

Calibrate the camera.

```
I = readimage(images,1);
imageSize = [size(I, 1),size(I, 2)];
params = estimateCameraParameters(imagePoints,worldPoints, ...
                                  'ImageSize',imageSize);
```

Visualize the calibration accuracy.

```
showReprojectionErrors(params);
```
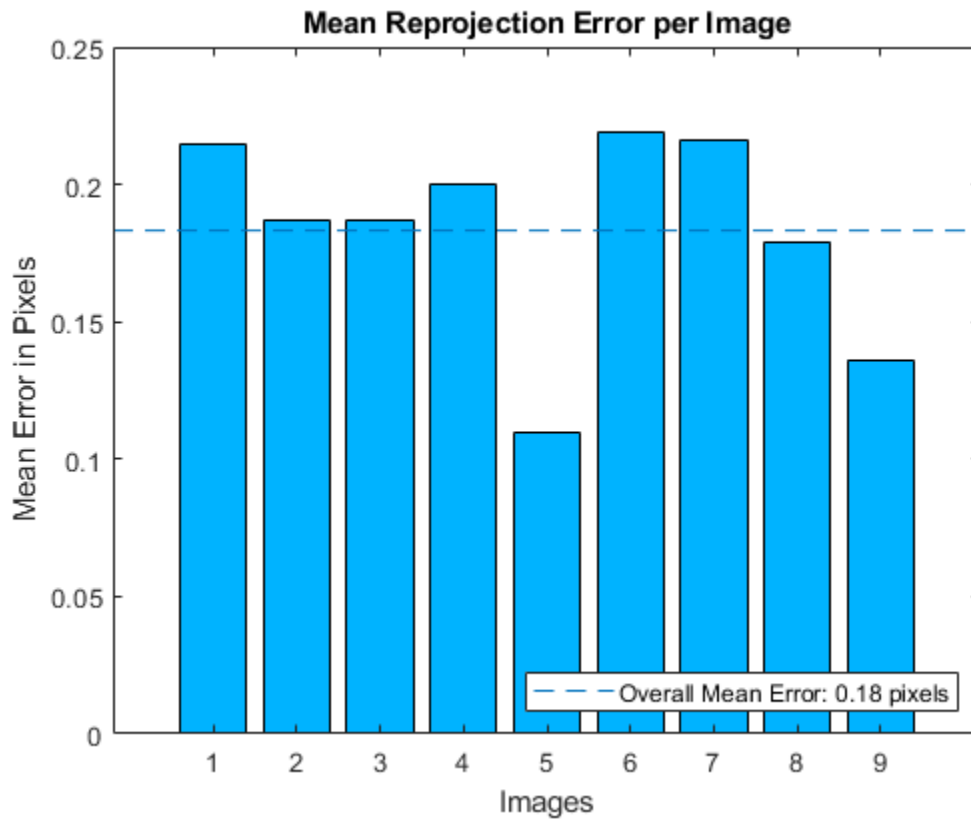
Visualize camera extrinsics.

```
figure;
showExtrinsics(params);
```

Extrinsic Parameters Visualization
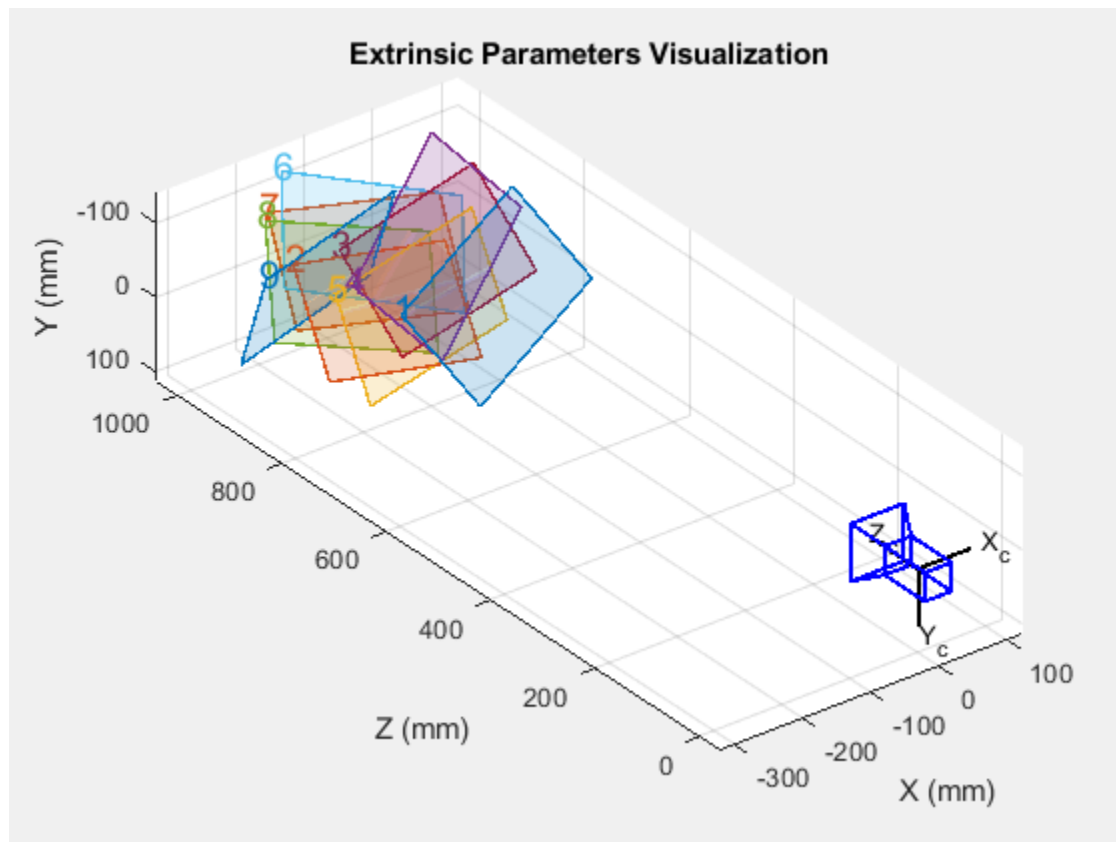
```
drawnow;
```

Plot detected and reprojected points.

```
figure;
imshow(imageFileNames{1});
hold on;
plot(imagePoints(:,1,1), imagePoints(:,2,1),'go');
plot(params.ReprojectedPoints(:,1,1),params.ReprojectedPoints(:,2,1),'r+');
legend('Detected Points','ReprojectedPoints');
hold off;
```

Remove Distortion from an Image Using the Camera Parameters Object

Use the camera calibration functions to remove distortion from an image. This example creates a vision.CameraParameters object manually, but in practice, you would use the estimateCameraParameters or the cameraCalibrator app to derive the object.

Create a vision.CameraParameters object manually.

```
IntrinsicMatrix = [715.2699 0 0; 0 711.5281 0; 565.6995 355.3466 1];
radialDistortion = [-0.3361 0.0921];
cameraParams = cameraParameters('IntrinsicMatrix',IntrinsicMatrix,'RadialDistortion',ra
```

Remove distortion from the images.

```
I = imread(fullfile(matlabroot,'toolbox','vision','visiondata','calibration','mono','ir
J = undistortImage(I,cameraParams);
```

Display the original and the undistorted images.

```
figure; imshowpair(imresize(I, 0.5),imresize(J, 0.5),'montage');
title('Original Image (left) vs. Corrected Image (right)');
```


Original Image (left) vs. Corrected Image (right)

Plot Spherical Point Cloud with Texture Mapping

Generate a sphere consisting of 600-by-600 faces.

```
numFaces = 600;
[x,y,z] = sphere(numFaces);
```

Plot the sphere using the default color map.

```
figure;
pcshow([x(:),y(:),z(:)]);
title('Sphere with Default Color Map');
xlabel('X');
ylabel('Y');
zlabel('Z');
```
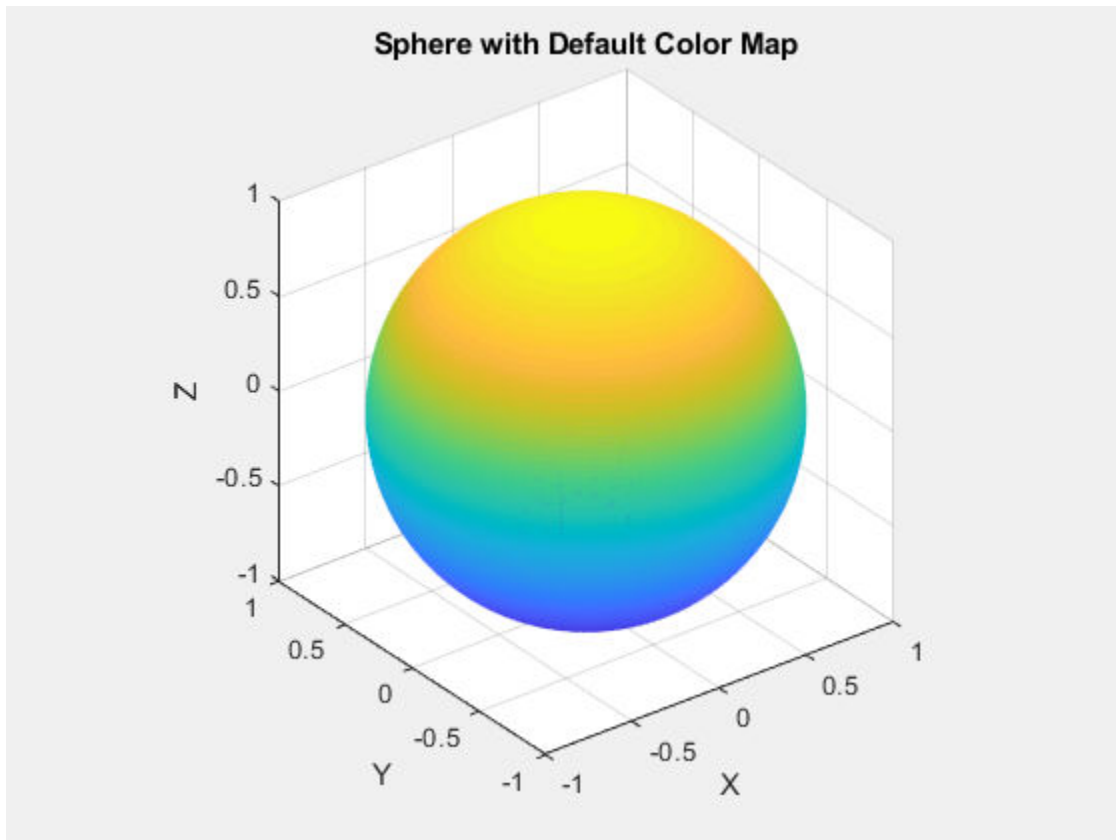
Load and display an image for texture mapping.

```
I = im2double(imread('visionteam1.jpg'));
imshow(I);
```

Resize and flip the image for mapping the coordinates.

```
J = flipud(imresize(I,size(x)));
```

Plot the sphere with the color texture.

```
pcshow([x(:),y(:),z(:)],reshape(J,[],3));
title('Sphere with Color Texture');
xlabel('X');
ylabel('Y');
zlabel('Z');
```

Sphere with Color Texture
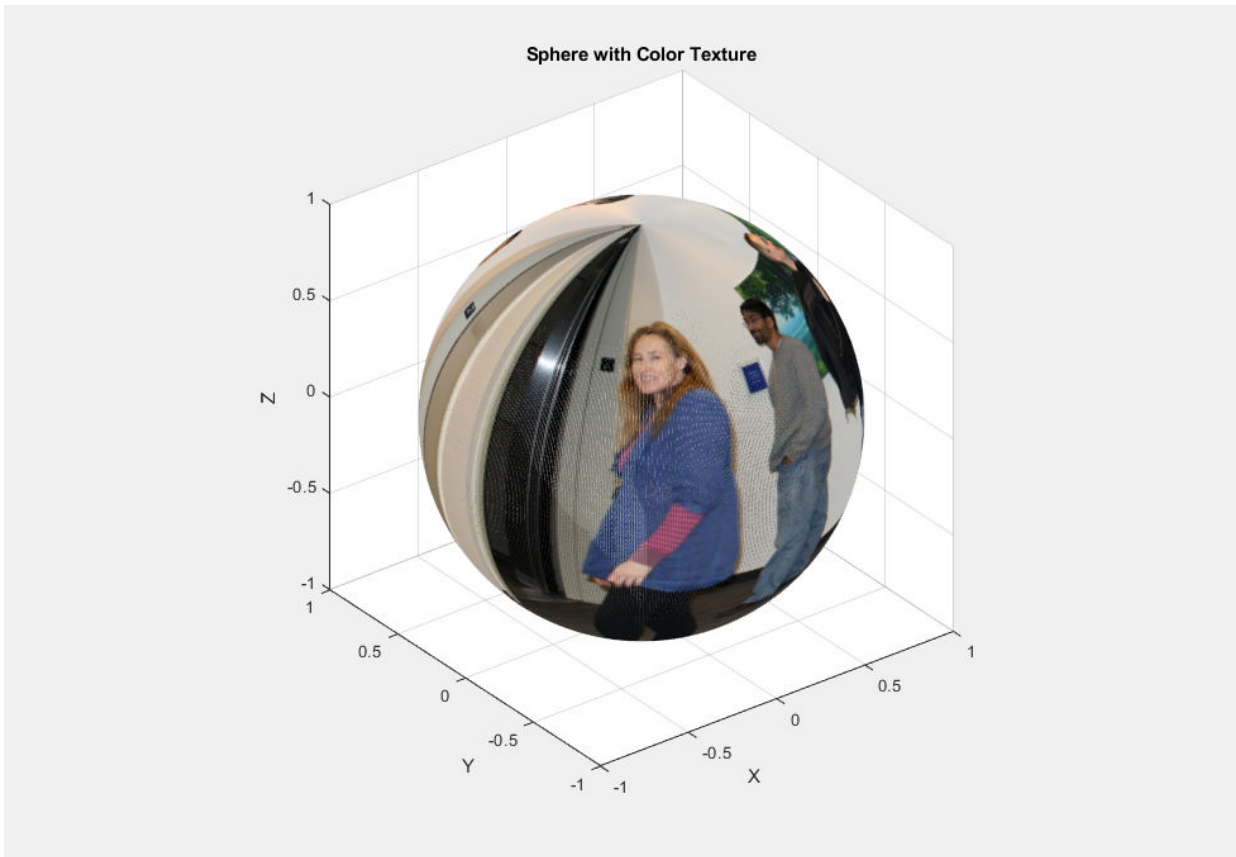
Plot Color Point Cloud from Kinect for Windows

This example shows how to plot a color point cloud from Kinect images. This example requires the Image Acquisition Toolbox™ software and the Kinect camera and a connection to the camera.

Create a System object™ for the color device.

```
colorDevice = imaq.VideoDevice('kinect',1)
```

Change the returned type of color image from `single` to `unint8`.

```
colorDevice.ReturnedDataType = 'uint8';
```

Create a System object for the depth device.

```
depthDevice = imaq.VideoDevice('kinect',2)
```

Initialize the camera.

```
step(colorDevice);
step(depthDevice);
```

Load one frame from the device.

```
colorImage = step(colorDevice);
depthImage = step(depthDevice);
```

Extract the point cloud.

```
ptCloud = pcfromkinect(depthDevice,depthImage,colorImage);
```

Initialize a point cloud player to visualize 3-D point cloud data. The axis is set appropriately to visualize the point cloud from Kinect.

```
player = pcplayer(ptCloud.XLimits,ptCloud.YLimits,ptCloud.ZLimits,...
    'VerticalAxis','y','VerticalAxisDir','down');

xlabel(player.Axes,'X (m)');
ylabel(player.Axes,'Y (m)');
zlabel(player.Axes,'Z (m)');
```

Acquire and view 500 frames of live Kinect point cloud data.

```
for i = 1:500
    colorImage = step(colorDevice);
```

```
    depthImage = step(depthDevice);

    ptCloud = pcfromkinect(depthDevice,depthImage,colorImage);

    view(player,ptCloud);
end
```
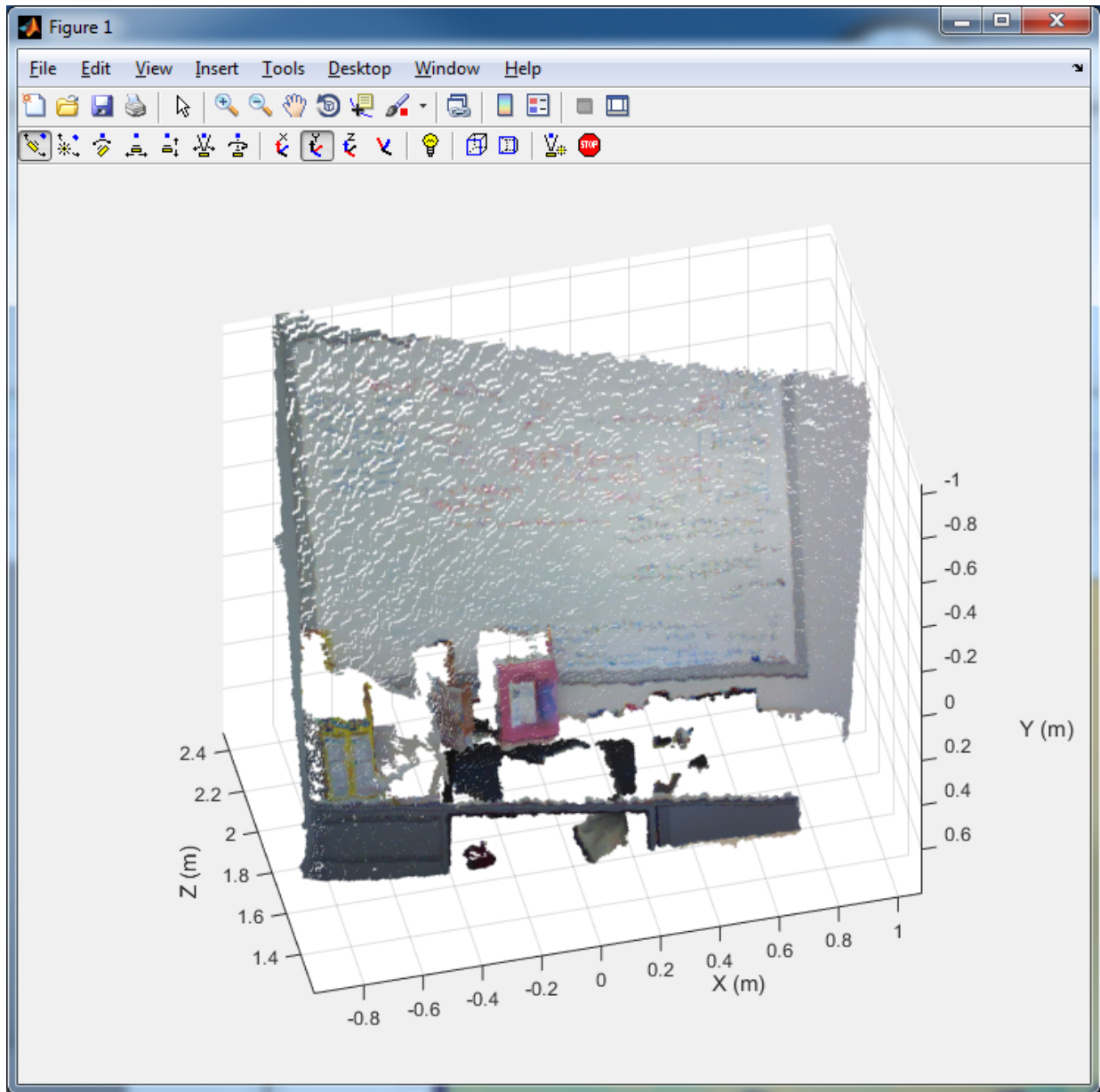
Release the objects.

```
release(colorDevice);
release(depthDevice);
```

Optical Flow Estimation Using the Farneback Algorithm

Load a video.

```
vidReader = VideoReader('visiontraffic.avi','CurrentTime',11);
```

Set up an optical flow object to do the estimate.

```
opticFlow = opticalFlowFarneback;
```

Read in video frames and estimate optical flow of each frame. Display the video frames with flow vectors.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB)
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',2)
    hold off
end
```
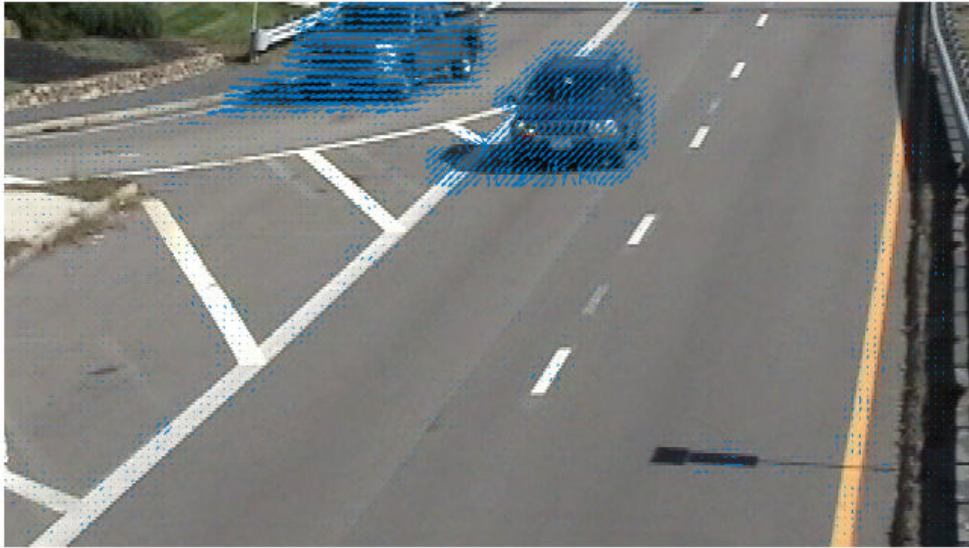
Compute Optical Flow Using Lucas-Kanade derivative of Gaussian

Read in a video file.

```
vidReader = VideoReader('viptraffic.avi');
```

Create optical flow object.

```
opticFlow = opticalFlowLKDoG('NumFrames',3);
```

Estimate the optical flow of the objects and display the flow vectors.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB);
    hold on;
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',25);
    hold off;
end
```

Compute Optical Flow Using Horn-Schunck Method

Read in a video file.

```
vidReader = VideoReader('viptraffic.avi');
```

Create optical flow object.

```
opticFlow = opticalFlowHS;
```

Estimate the optical flow of objects in the video.

```
while hasFrame(vidReader)
    frameRGB = readFrame(vidReader);
    frameGray = rgb2gray(frameRGB);

    flow = estimateFlow(opticFlow,frameGray);

    imshow(frameRGB)
    hold on
    plot(flow,'DecimationFactor',[5 5],'ScaleFactor',25)
    hold off
end
```

Construct Optical Flow Object and Plot Its Velocity

Construct an optical flow object.

```
opflow = opticalFlow(randn(100,100),randn(100,100));
```

Plot the velocity of the object as a quiver plot.

```
plot(opflow,'DecimationFactor',[10 10],'ScaleFactor',10);
```
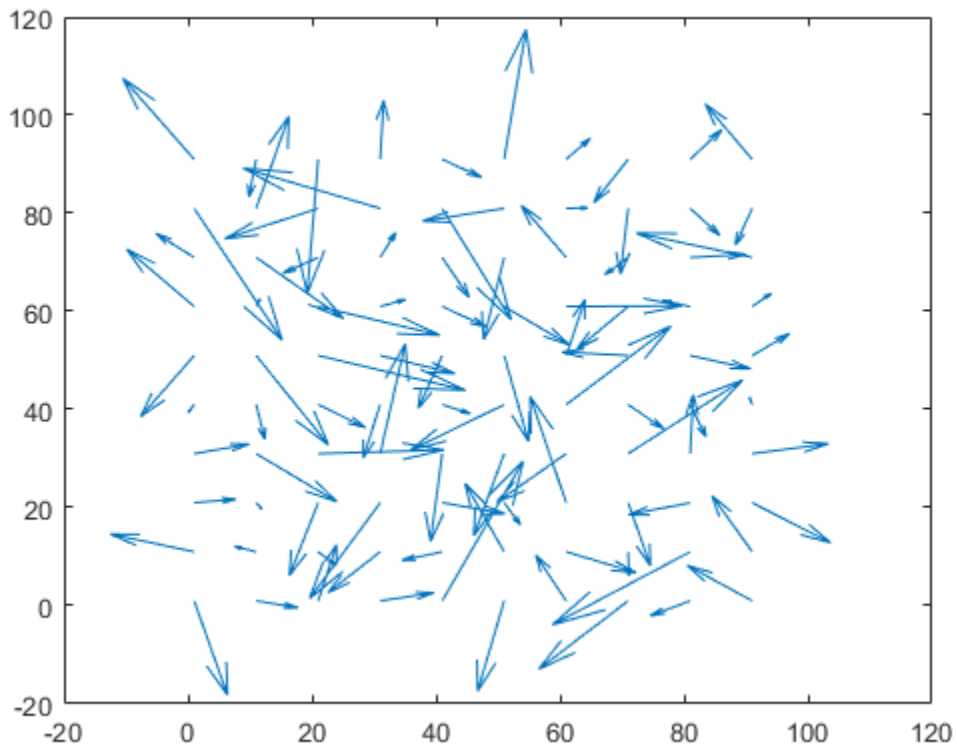
Point Cloud Processing

Point Cloud Registration Workflow

Complete Point Cloud Registration Workflow

Point Cloud A   Point Cloud B

Remove outliers
`pcdenoise`
Optional - slow
Remove outliers
`pcdenoise`

Downsample
`pcdownsample`
Critical
Random *or* **Grid Average**
Downsample
`pcdownsample`

Rigid Registration

Rigid Registration
`pcregistericp`

Match points between moving and fixed point clouds - kd Tree

Remove incorrect matches - Outlier Filter   `'InlierRatio'`

Recover rotation and translation - Minimize Error   `'pointToPoint'`
`'pointToPlane'`

Check if algorithm stops   `'Tolerance'`

Alignment
`pctransform`
Optional - check alignment

Merge
`pcmerge`
Optional - stitch point clouds

See Also

pcregistericp

Related Examples

- "3-D Point Cloud Registration and Stitching"

The PLY Format

In this section...
"File Header" on page 2-4
"Data" on page 2-6
"Common Elements and Properties" on page 2-7

The version 1.0 PLY format, also known as the Stanford Triangle Format, defines a flexible and systematic scheme for storing 3D data. The ASCII header specifies what data is in the file by defining "elements" each with a set of "properties." Many PLY files only have vertex and face data, however, it is possible to also include other data such as color information, vertex normals, or application-specific properties.

Note The Computer Vision System Toolbox point cloud data functions only support the (x,y,z) coordinates, normals, and color properties.

File Header

An example header (italicized text is comment):

ply	*file ID*
format binary_big_endian 1.0	*specify data format and version*
element vertex 9200	*define "vertex" element*
property float x	
property float y	
property float z	
element face 18000	*define "face" element*
property list uchar int vertex_indices	
end_header	*data starts after this line*

The file begins with "ply," identifying that it is a PLY file. The header must also include a format line with the syntax

format <data format> <PLY version>

Supported data formats are "ascii" for data stored as text and "binary_little_endian" and "binary_big_endian" for binary data (where little/big endian refers to the byte ordering of multi-byte data). Element definitions begin with an "element" line followed by element property definitions

```
element <element name><number in file>
property <data type><property name 1>
property <data type><property name 2>
property <data type><property name 3>
...
```

For example, "element vertex 9200" defines an element "vertex" and specifies that 9200 vertices are stored in the file. Each element definition is followed by a list of properties of that element. There are two kinds of properties, scalar and list. A scalar property definition has the syntax

```
property <data type><property name>
```

where <data type> is

Name	Type
char	(8-bit) character
uchar	(8-bit) unsigned character
short	(16-bit) short integer
ushort	(16-bit) unsigned short integer
int	(32-bit) integer
uint	(32-bit) unsigned integer
float	(32-bit) single-precision float
double	(64-bit) double-precision float

For compatibility between systems, note that the number of bits in each data type must be consistent. A list type is stored with a count followed by a list of scalars. The definition syntax for a list property is

```
property list <count data type><data type><property name>
```

For example,

```
property list uchar int vertex_index
```

defines vertex_index properties are stored starting with a byte count followed by integer values. This is useful for storing polygon connectivity as it has the flexibility to specify a variable number of vertex indices in each face.

The header can also include comments. The syntax for a comment is simply a line beginning with "comment" followed by a one-line comment:

```
comment<comment text>
```

Comments can provide information about the data like the file's author, data description, data source, and other textual data.

Data

Following the header, the element data is stored as either ASCII or binary data (as specified by the format line in the header). After the header, the data is stored in the order the elements and properties were defined. First, all the data for the first element type is stored. In the example header, the first element type is "vertex" with 9200 vertices in the file, and with float properties "x," "y," and "z."

`float` vertex[1].x
`float` vertex[1].y
`float` vertex[1].z
`float` vertex[2].x
`float` vertex[2].y
`float` vertex[2].z
...
`float` vertex[9200].x
`float` vertex[9200].y
`float` vertex[9200].z

In general, the properties data for each element is stored one element at a time.

<property 1><property 2> ... <property N> element[1]

<property 1><property 2> ... <property N> element[2]
...

The list type properties are stored beginning with a count and followed by a list of scalars. For example, the "face" element type has the list property "vertex_indices" with uchar count and int scalar type.

`uchar` count
`int` face[1].vertex_indices[1]
`int` face[1].vertex_indices[2]
`int` face[1].vertex_indices[3]
...
`int` face[1].vertex_indices[count]

`uchar` count
`int` face[2].vertex_indices[1]
`int` face[2].vertex_indices[2]
`int` face[2].vertex_indices[3]
...
`int` face[2].vertex_indices[count]

...

Common Elements and Properties

While the PLY format has the flexibility to define many types of elements and properties, a common set of elements are understood between programs to communicate common 3-D data types. Turk suggests elements and property names that programs should try to make standard.

Required Core Property	Element	Property	Data Type	Property Description
✓	vertex	x	`float`	*x,y,z* coordinates
✓		y	`float`	
✓		z	`float`	
		nx	`float`	*x,y,z* of normal
		ny	`float`	
		nz	`float`	
		red	`uchar`	vertex color
		green	`uchar`	
		blue	`uchar`	
		alpha	`uchar`	amount of transparency
		material_index	`int`	index to list of materials
	face	vertex_indices	list of int	indices to vertices
		back_red	`uchar`	backside color
		back_green	`uchar`	
		back_blue	`uchar`	
	edge	vertex1	`int`	index to vertex
		vertex2	`int`	index to other vertex
		crease_tag	`uchar`	crease in subdivision surface
	material	red	`uchar`	material color
		green	`uchar`	
		blue	`uchar`	
		alpha	`uchar`	amount of transparency
		reflect_coeff	`float`	amount of light reflected

Requi red Core Prope rty	Eleme nt	Property	Data Type	Property Description
		refract_coeff	`float`	amount of light refracted
		refract_index	`float`	index of refraction
		extinct_coeff	`float`	extinction coefficent

See Also

`pcread` | `pcwrite`

Using the Installer for Computer Vision System Toolbox Product

Install Computer Vision System Toolbox Add-on Support Files

After you install third-party support files, you can use the data with the Computer Vision System Toolbox product. Use one of two ways to install the Add-on support files.

- Select **Get Add-ons** from the **Add-ons** drop-down menu from the MATLAB® desktop. The Add-on files are in the "MathWorks Features" section.
- Type `visionSupportPackages` in a MATLAB Command Window and follow the prompts.

Note You must have write privileges for the installation folder.

When a new version of MATLAB software is released, repeat this process to check for updates. You can also check for updates between releases.

Install OCR Language Data Files

In this section...
"Installation" on page 3-3
"Pretrained Language Data and the ocr function" on page 3-3

OCR Language Data files contain pretrained language data from the OCR Engine, tesseract-ocr, to use with the `ocr` function.

Installation

After you install third-party support files, you can use the data with the Computer Vision System Toolbox product. Use one of two ways to install the Add-on support files.

- Select **Get Add-ons** from the **Add-ons** drop-down menu from the MATLAB desktop. The Add-on files are in the "MathWorks Features" section.
- Type `visionSupportPackages` in a MATLAB Command Window and follow the prompts.

Note You must have write privileges for the installation folder.

When a new version of MATLAB software is released, repeat this process to check for updates. You can also check for updates between releases.

Pretrained Language Data and the ocr function

After you install the pretrained language data files, you can specify one or more additional languages using the `Language` property of the `ocr` function. Use the appropriate language character vector with the property.

```
txt = ocr(img,'Language','Finnish');
```

List of OCR language data in support package

- `'Afrikaans'`
- `'Albanian'`

- 'AncientGreek'
- 'Arabic'
- 'Azerbaijani'
- 'Basque'
- 'Belarusian'
- 'Bengali'
- 'Bulgarian'
- 'Catalan'
- 'Cherokee'
- 'ChineseSimplified'
- 'ChineseTraditional'
- 'Croatian'
- 'Czech'
- 'Danish'
- 'Dutch'
- 'English'
- 'Esperanto'
- 'EsperantoAlternative'
- 'Estonian'
- 'Finnish'
- 'Frankish'
- 'French'
- 'Galician'
- 'German'
- 'Greek'
- 'Hebrew'
- 'Hindi'
- 'Hungarian'
- 'Icelandic'
- 'Indonesian'

- 'Italian'
- 'ItalianOld'
- 'Japanese'
- 'Kannada'
- 'Korean'
- 'Latvian'
- 'Lithuanian'
- 'Macedonian'
- 'Malay'
- 'Malayalam'
- 'Maltese'
- 'MathEquation'
- 'MiddleEnglish'
- 'MiddleFrench'
- 'Norwegian'
- 'Polish'
- 'Portuguese'
- 'Romanian'
- 'Russian'
- 'SerbianLatin'
- 'Slovakian'
- 'Slovenian'
- 'Spanish'
- 'SpanishOld'
- 'Swahili'
- 'Swedish'
- 'Tagalog'
- 'Tamil'
- 'Telugu'
- 'Thai'

- 'Turkish'
- 'Ukrainian'

See Also

OCR Trainer | ocr | visionSupportPackages

Related Examples

- "Recognize Text Using Optical Character Recognition (OCR)"

Install and Use Computer Vision System Toolbox OpenCV Interface

Use the OpenCV Interface files to integrate your OpenCV C++ code into MATLAB and build MEX-files that call OpenCV functions. The support package also contains graphics processing unit (GPU) support.

In this section...
"Installation" on page 3-7
"Support Package Contents" on page 3-7
"Create MEX-File from OpenCV C++ file" on page 3-8
"Use the OpenCV Interface C++ API" on page 3-9
"Create Your Own OpenCV MEX-files" on page 3-10
"Run OpenCV Examples" on page 3-10

Installation

After you install third-party support files, you can use the data with the Computer Vision System Toolbox product. Use one of two ways to install the Add-on support files.

- Select **Get Add-ons** from the **Add-ons** drop-down menu from the MATLAB desktop. The Add-on files are in the "MathWorks Features" section.
- Type `visionSupportPackages` in a MATLAB Command Window and follow the prompts.

Note You must have write privileges for the installation folder.

When a new version of MATLAB software is released, repeat this process to check for updates. You can also check for updates between releases.

Support Package Contents

The OpenCV Interface support files are installed in the `visionopencv` folder. To find the path to this folder, type the following command:

```
fileparts(which('mexOpenCV'))
```

The `visionopencv` folder contain these files and folder.

Files	Contents
`example` folder	Template Matching, Foreground Detector, and Oriented FAST and Rotated BRIEF (ORB) examples, including a GPU version. Each subfolder in the example folder contains a `README.txt` file with step-by-step instructions.
`registry` folder	Registration files.
`mexOpenCV.m` file	Function to build MEX-files.
`README.txt` file	Help file.

The `mex` function uses prebuilt OpenCV libraries, which ship with the Computer Vision System Toolbox product. Your compiler must be compatible with the one used to build the libraries. The following compilers are used to build the OpenCV libraries for MATLAB host:

Operating System	Compatible Compiler
Windows® 64 bit	Microsoft® Visual Studio® 2015 Professional
Linux® 64 bit	gcc-4.9.3 (g++)
Mac 64 bit	Xcode 6.2.0 (Clang++)

Create MEX-File from OpenCV C++ file

This example creates a MEX-file from a wrapper C++ file and then tests the newly created file. The example uses the OpenCV template matching algorithm wrapped in a C++ file, which is located in the `example/TemplateMatching` folder.

1 Change your current working folder to the `example/TemplateMatching` folder:

```
cd(fullfile(fileparts(which('mexOpenCV')),'example',filesep,'TemplateMatching'))
```

2 Create the MEX-file from the source file:

```
mexOpenCV matchTemplateOCV.cpp
```

3   Run the test script, which uses the generated MEX-file:

```
testMatchTemplate
```

Use the OpenCV Interface C++ API

The `mexOpenCV` interface utility functions convert data between OpenCV and MATLAB. These functions support CPP-linkage only. GPU support is available on glnxa64, win64, and Mac platforms. The GPU-specific utility functions support CUDA-enabled NVIDIA GPU with compute capability 2.0 or higher. See the Parallel Computing Toolbox™ System Requirements, The GPU utility functions require the Parallel Computing Toolbox software.

Function	Description
`ocvCheckFeaturePointsStruct`	Check that MATLAB struct represents feature points
`ocvStructToKeyPoints`	Convert MATLAB feature points struct to OpenCV `KeyPoint` vector
`ocvKeyPointsToStruct`	Convert OpenCV `KeyPoint` vector to MATLAB struct
`ocvMxArrayToCvRect`	Convert a MATLAB struct representing a rectangle to an OpenCV `CvRect`
`ocvCvRectToMxArray`	Convert OpenCV `CvRect` to a MATLAB struct
`ocvCvBox2DToMxArray`	Convert OpenCV `CvBox2D` to a MATLAB struct
`ocvCvRectToBoundingBox_{DataType}`	Convert `vector<cv::Rect>` to *M*-by-4 `mxArray` of bounding boxes
`ocvMxArrayToSize_{DataType}`	Convert 2-element `mxArray` to `cv::Size`
`ocvMxArrayToImage_{DataType}`	Convert column major `mxArray` to row major `cv::Mat` for image
`ocvMxArrayToMat_{DataType}`	Convert column major `mxArray` to row major `cv::Mat` for generic matrix
`ocvMxArrayFromImage_{DataType}`	Convert row major `cv::Mat` to column major `mxArray` for image
`ocvMxArrayFromMat_{DataType}`	Convert row major `cv::Mat` to column major `mxArray` for generic matrix.
`ocvMxArrayFromVector`	Convert numeric `vectorT` to `mxArray`
`ocvMxArrayFromPoints2f`	Converts `vector<cv::Point2f>` to `mxArray`

GPU Function	Description
ocvMxGpuArrayToGpuMat_{DataType}	Create cv::gpu::GpuMat from gpuArray
ocvMxGpuArrayFromGpuMat_{DataType}	Create gpuArray from cv::gpu::GpuMat

Create Your Own OpenCV MEX-files

Call the mxArray function with your source file.

```
mexOpenCV yourfile.cpp
```

For help creating MEX files, at the MATLAB command prompt, type:

```
help mexOpenCV
```

Run OpenCV Examples

Each example subfolder in the OpenCV Interface support package contains all the files you need to run the example. To run an example, you must call the mexOpenCV function with one of the supplied source files.

Run Template Matching Example

1 Change your current working folder to the example/TemplateMatching folder:

```
cd(fullfile(fileparts(which('mexOpenCV')),'example',filesep,'TemplateMatching'))
```

2 Create the MEX-file from the source file:

```
mexOpenCV matchTemplateOCV.cpp
```

3 Run the test script, which uses the generated MEX-file:

```
testMatchTemplate
```

Run Foreground Detector Example

1 Change your current working folder to the example/ForegroundDetector folder:

```
cd(fullfile(fileparts(which('mexOpenCV')),'example',filesep,'ForegroundDetector'))
```

2 Create the MEX-file from the source file:

```
mexOpenCV backgroundSubtractorOCV.cpp
```

3 Run the test script that uses the generated MEX-file:

`testBackgroundSubtractor.m`

Run Oriented FAST and Rotated BRIEF (ORB) Detector Example

1 Change your current working folder to the `example/ORB` folder:

`cd(fullfile(fileparts(which('mexOpenCV')),'example',filesep,'ORB'))`

2 Create the MEX-file for the detector from the source file:

`mexOpenCV detectORBFeaturesOCV.cpp`

3 Create the MEX-file for the extractor from the source file:

`mexOpenCV extractORBFeaturesOCV.cpp`

4 Run the test script, which uses the generated MEX-files:

`testORBFeaturesOCV.m`

Run Detect ORB Features (GPU Version) Example

1 Change your current working folder to the `example/ORB_GPU` folder:

`cd(fullfile(fileparts(which('mexOpenCV')),'example',filesep,'ORB_GPU'))`

2 Create the MEX-file for the detector from the source file.

PC:

`mexOpenCV detectORBFeaturesOCV_GPU.cpp -lgpu -lmwocvgpumex -largeArrayDims`

Linux/Mac:

`mexOpenCV detectORBFeaturesOCV_GPU.cpp -lmwgpu -lmwocvgpumex -largeArrayDims`

3 Run the test script, which uses the generated MEX-file:

`testORBFeaturesOCV_GPU.m`

See Also

"C Matrix API" (MATLAB) | `mxArray`

More About

- "Install Computer Vision System Toolbox Add-on Support Files" on page 3-2
- Using OpenCV with MATLAB

4

Input, Output, and Conversions

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

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 at the MATLAB command line.

```
ex_export_to_mmf
```

1   Run your model.

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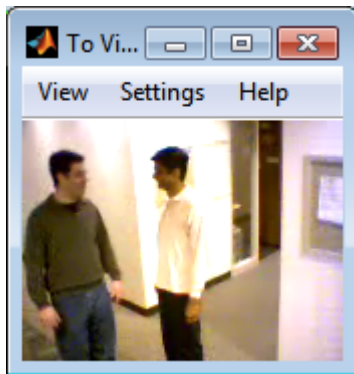
By increasing the red, green, and blue color values, you increase the contrast of the video. The To Multimedia File block exports the video data from the Simulink model to a multimedia file that it creates in your current folder.

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

Setting Block Parameters for this Example

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

Block	Parameter
Gain	The Gain blocks are used to increase the red, green, and blue values of the video stream. This increases the contrast of the video: • **Main** pane, **Gain** = `1.2` • **Signal Attributes** pane, **Output data type** = `Inherit: Same as input`
To Multimedia File	The To Multimedia File block exports the video to a multimedia file: • **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 at the MATLABcommand line.

```
ex_import_mmf
```

1   Run your model.

2   View your video in the To Video Display window that automatically appears when you start your simulation.
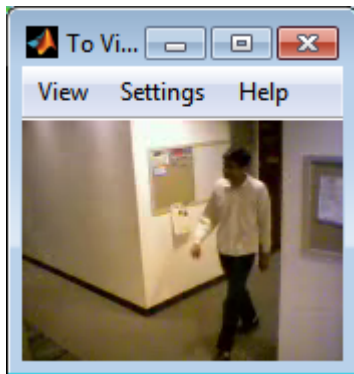
You have now imported and displayed a multimedia file in the Simulink model. In the "Export to Video Files" on page 4-2 example you can manipulate your video stream and export it to a multimedia file.

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	Use the From Multimedia File block to import the multimedia file into the model: • If you do not have your own multimedia file, use the default `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`. By default, the **Number of times to play file** parameter is set to `inf`. The model continues to play the file until the simulation stops.
To Video Display	Use the To Video Display block to view the multimedia file. • **Image signal**: `Separate color signals` Set this parameter from the **Settings** menu of the display viewer.

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','imdata');
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 "Image Sequences and Batch Processing" (Image Processing Toolbox) section for the Image Processing Toolbox™.

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 4-6 example, which stores images in a workspace variable.

You can open the example model by typing at the MATLAB command line.

```
ex_display_sequence_of_images
```

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 "Callbacks for Customized Model Behavior" (Simulink).

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 by typing at the MATLAB command line.

```
ex_vision_partition_video_frames_to_multiple_files
```

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:

```
implay output1.avi
implay output2.avi
implay output3.avi
```

4   Press the **Play** button.

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

Setting Block Parameters for this Example

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

Block	Parameter
From Multimedia File	The From Multimedia File block imports an AVI file into the model. • Cleared **Inherit sample time from file** checkbox.

Block	Parameter
Insert Text	The example uses the Insert Text block to annotate the video stream with frame numbers. The block writes the frame number in green, in the upper-left corner of the output video stream. • **Text**: `'Frame %d'` • **Color**: `[0 1 0]` • **Location**: `[10 10]`
To Multimedia File	The To Multimedia File blocks send the video stream to three separate AVI files. These block parameters were modified as follows: • **File name**: `output1.avi`, `output2.avi`, and `output3.avi`, respectively • **Write**: `Video only`
Counter	The Counter block counts the number of video frames. The example uses this information to specify which frames are sent to which file. The block parameters are modified as follows: • **Number of bits**: 8 • **Sample time**: 1/30
Bias	The bias block adds a bias to the input. The block parameters are modified as follows: • **Bias**: 1
Compare To Constant	The Compare to Constant block sends frames 1 to 9 to the first AVI file. The block parameters are modified as follows: • **Operator**: < • **Constant value**: 10

Block	Parameter
Compare To Constant1 **Compare To Constant2**	The Compare to Constant1 and Compare to Constant2 blocks send frames 10 to 19 to the second AVI file. The block parameters are modified as follows: • **Operator**: >= • **Constant value**: 10 The Compare to Constant2 block parameters are modified as follows: • **Operator**: < • **Constant value**: 20
Compare To Constant3	The Compare to Constant3 block send frames 20 to 30 to the third AVI file. The block parameters are modified as follows: • **Operator**: >= • **Constant value**: 20
Compare To Constant4	The Compare to Constant4 block stops the simulation when the video reaches frame 30. The block parameters are modified as follows: • **Operator**: == • **Constant value**: 30 • **Output data type**: 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.
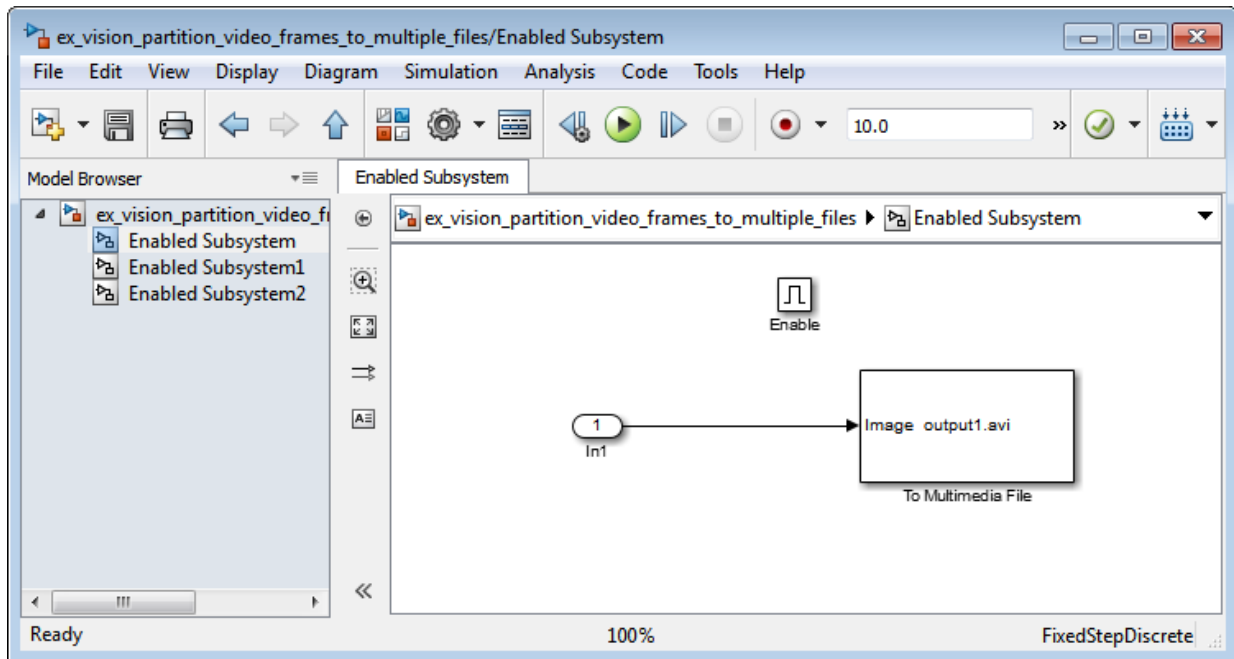
Configuration Parameters

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:

- **Type** = Fixed-step
- **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 at the MATLAB command line.

ex_combine_video_and_audio_streams

1   Run your model. The model creates a multimedia file called `output.avi` in your current folder.
2   Play the multimedia file using a media player. The original video file now has an audio component to it.

Setting Up the Video Input Block

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

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

Setting Up the Audio Input Block

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

The **Samples per audio channel** 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.

4-15

frame period = (*frame size*)/(*frequency*)
frame period = (735 *samples per frame*)/(22050 *samples per second*)
frame period = 0.0333 seconds per frame
Alternatively, you can verify that the frame period of the audio and video signals is the same using a Simulink Probe block.

Setting Up the Output Block

The To Multimedia File block is used to output the audio and video signals to a single multimedia file. The Video and audio option is selected for the **Write** parameter and One multidimensional signal for the **Image signal** parameter. The other default parameters are accepted.

Configuration Parameters

You can locate the Configuration Parameters by selecting **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.

Source Block Parameters: Video From Workspace

Video From Workspace (mask) (link)

Outputs video frames from the MATLAB workspace at successive sample times.

If the video signal is a M-by-N-by-T workspace array, the block outputs an intensity video signal, where M and N are the number of rows and columns in a single video frame, and T is the number of frames in the video signal.

If the video signal is a M-by-N-by-C-by-T workspace array, the block outputs a color video signal, where M and N are the number of rows and columns in a single video frame, C is the number of outputs from the block, and T is the number of frames in the video stream.

Parameters

Signal:

rand(32,32,3,5)

Sample time:

1

Form output after final value by: Cyclic repetition

Image signal: One multidimensional signal

OK   Cancel   Help   Apply

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.

Resample Image Chroma

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

You can open the example model by typing at the MATLAB command line.

```
ex_vision_resample_image_chroma
```
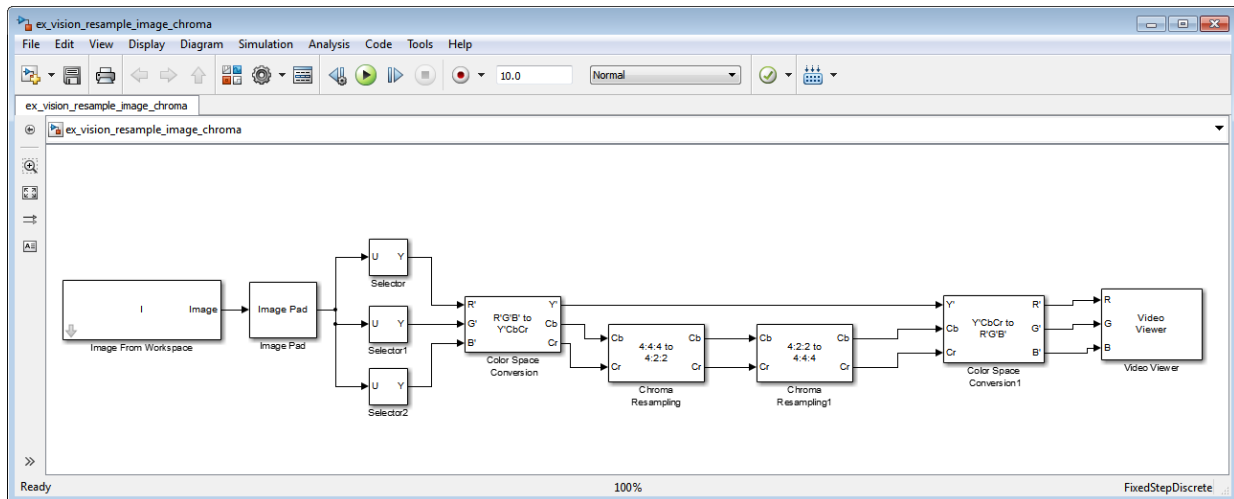

1. Define an RGB image in the MATLAB workspace. To do so, at the MATLAB command prompt, type:

   ```
   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 **Display** > **Signals & Ports** > **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 I.

Block	Parameter
Image Pad	Change dimensions of the input I array from 206-by-345-by-3 to 206-by-346-by-3. You are changing these dimensions because the Chroma Resampling block requires that the dimensions of the input be divisible by 2. Set the block parameters as follows: • **Method** = Symmetric • **Add columns to** = Right • **Number of added columns** = 1 • **Add row to** = No padding The Image Pad block adds one column to the right of each plane of the array by repeating its border values. This padding minimizes the effect of the pixels outside the image on the processing of the image. **Note** When you process video streams, be aware that it is computationally expensive to pad every video frame. You should change the dimensions of the video stream before you process it with Computer Vision System Toolbox blocks.

Block	Parameter
Selector, Selector1, Slector2	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: **Selector** • **Number of input dimensions** = 3 • **Index 1** = `Select all` • **Index 2** = `Select all` • **Index 3** = `Index vector (dialog)` and **Index** = 1 **Selector1** • **Number of input dimensions** = 3 • **Index 1** = `Select all` • **Index 2** = `Select all` • **Index 3** = `Index vector (dialog)` and **Index** = 2 **Selector2** • **Number of input dimensions** = 3 • **Index 1** = `Select all` • **Index 2** = `Select all` • **Index 3** = `Index vector (dialog)` and **Index** = 3
Color Space Conversion	Convert the input values from the R'G'B' color space to the Y'CbCr color space. The prime symbol indicates a gamma corrected signal. Set the **Image signal** parameter to `Separate color signals`.
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`.

Block	Parameter
Color Space Conversion1	Convert the input values from the Y'CbCr color space to the R'G'B' color space. Set the block parameters as follows: • **Conversion** = `Y'CbCr to R'G'B'` • **Image signal** = `Separate color signals`
Video Viewer	Display the recovered image. Select **File>Image signal** to set **Image signal** to `Separate color signals`.

Configuration Parameters

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

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

Convert Intensity Images 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

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 at the MATLAB command line.

ex_vision_thresholding_intensity

1   You can create a new Simulink model and add the blocks shown in the 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.
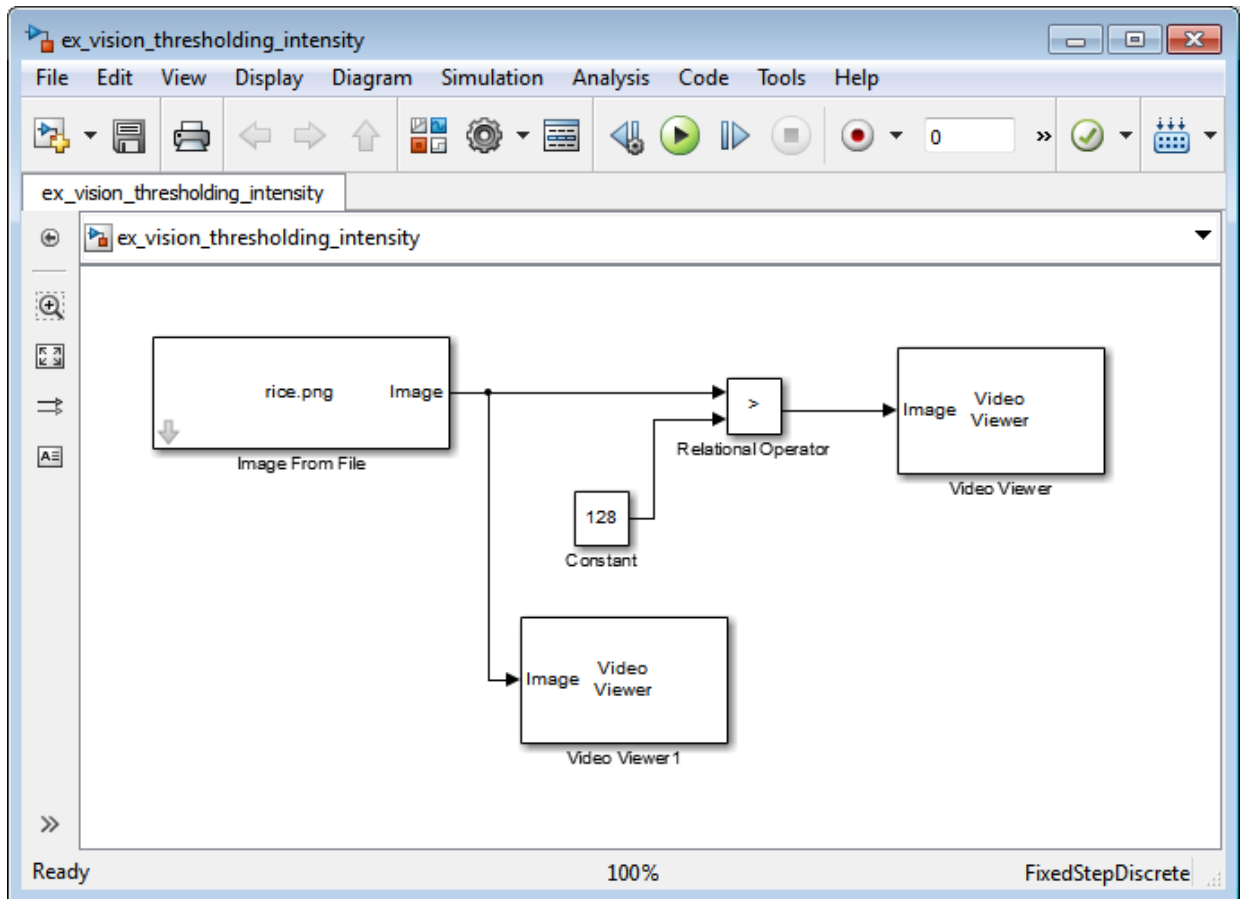
8   Set the configuration parameters. Open the Configuration dialog box by selecting **Simulation > Model Configuration Parameters** menu. Set the parameters as follows:

- **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.
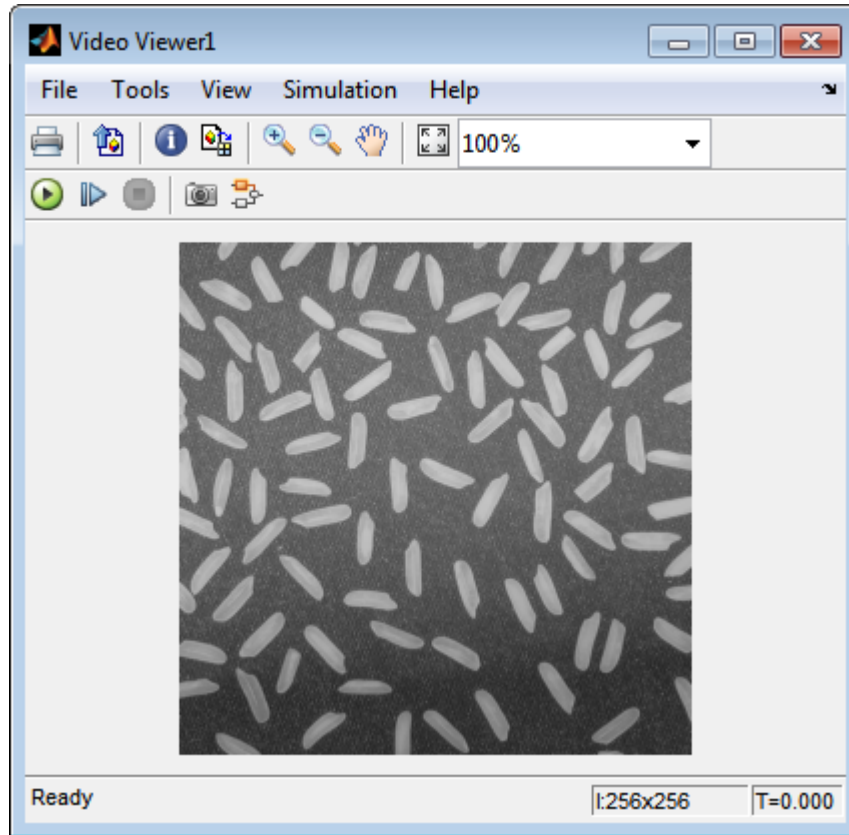
The binary image appears in the Video Viewer window.
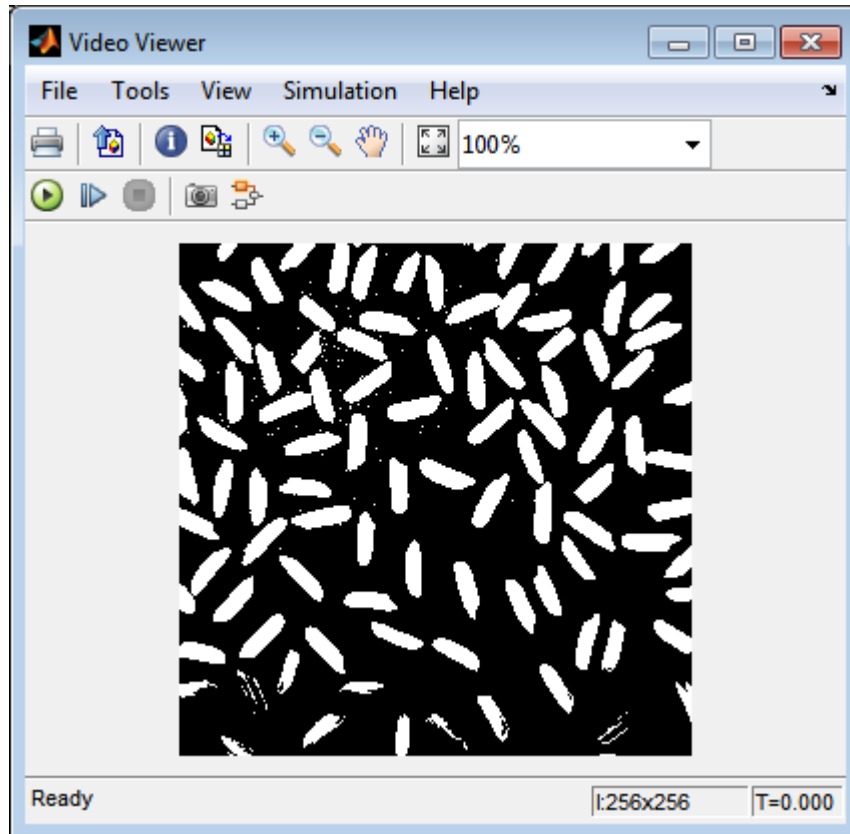
Note A single threshold value was unable to effectively threshold this image due to its uneven lighting. For information on how to address this problem, see "Correct Nonuniform Illumination" on page 11-7.

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. For additional information, see "Convert Between Image Types" (Image Processing Toolbox).

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 4-24 is not open on your desktop, you can open the model by typing

   ex_vision_thresholding_intensity

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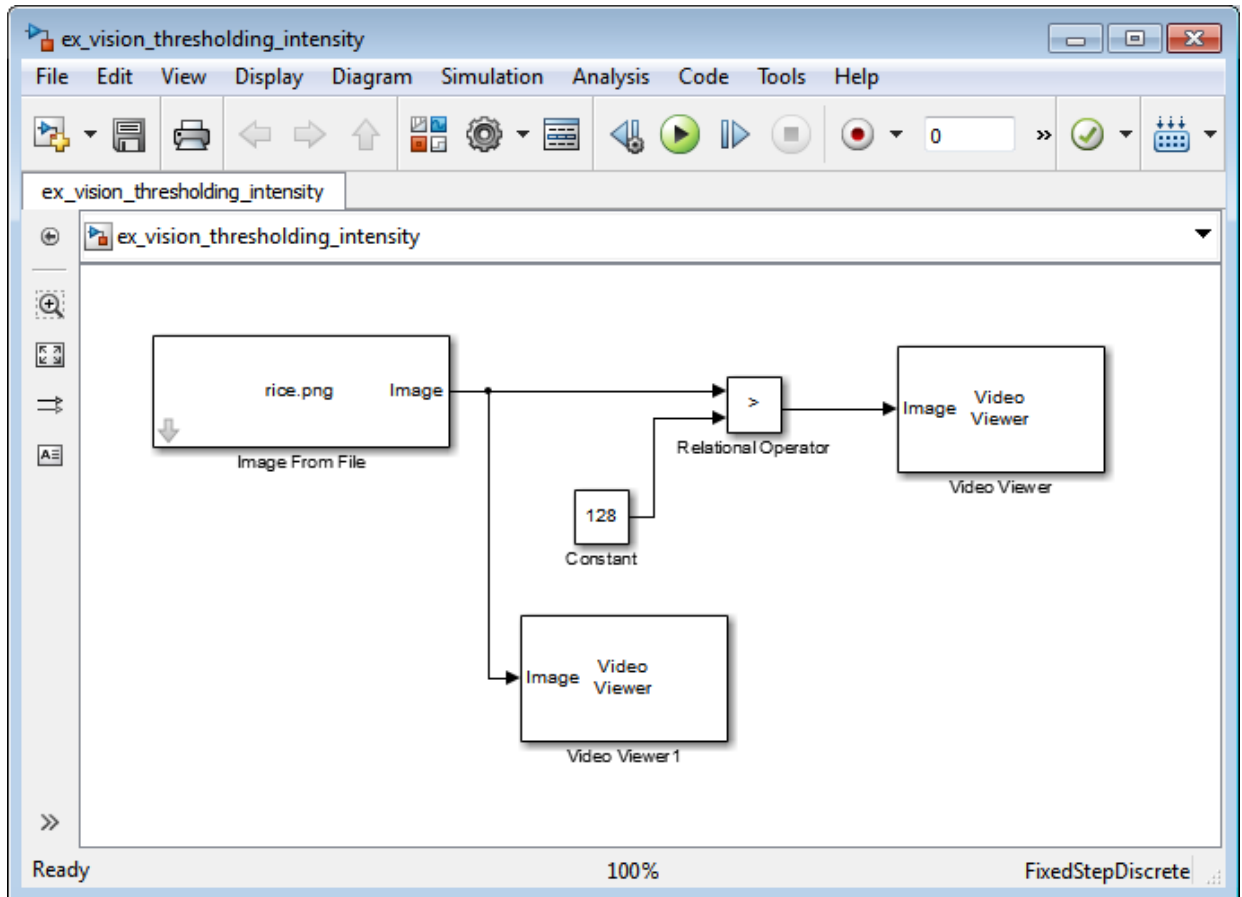
2. Use the Image from File block to import your image. In this example the image file is a 256-by-256 matrix of 8-bit unsigned integer values that range from 0 to 255. Set the **File name** parameter to `rice.png`

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 Authothreshold block.

Your model should look similar to the following figure:
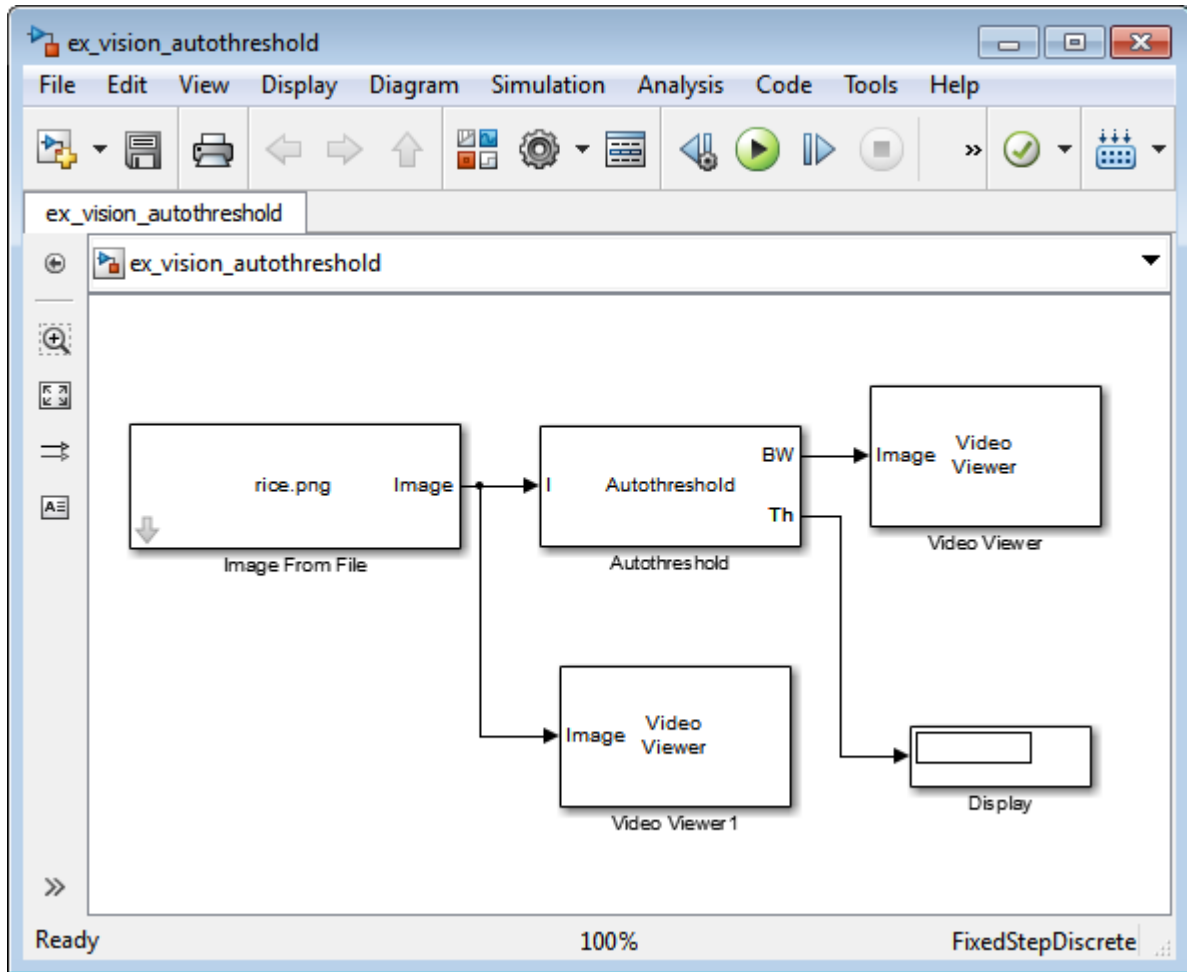
7   If you have not already done so, set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

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

8 Run the model.

The original intensity image appears in the Video Viewer1 window.
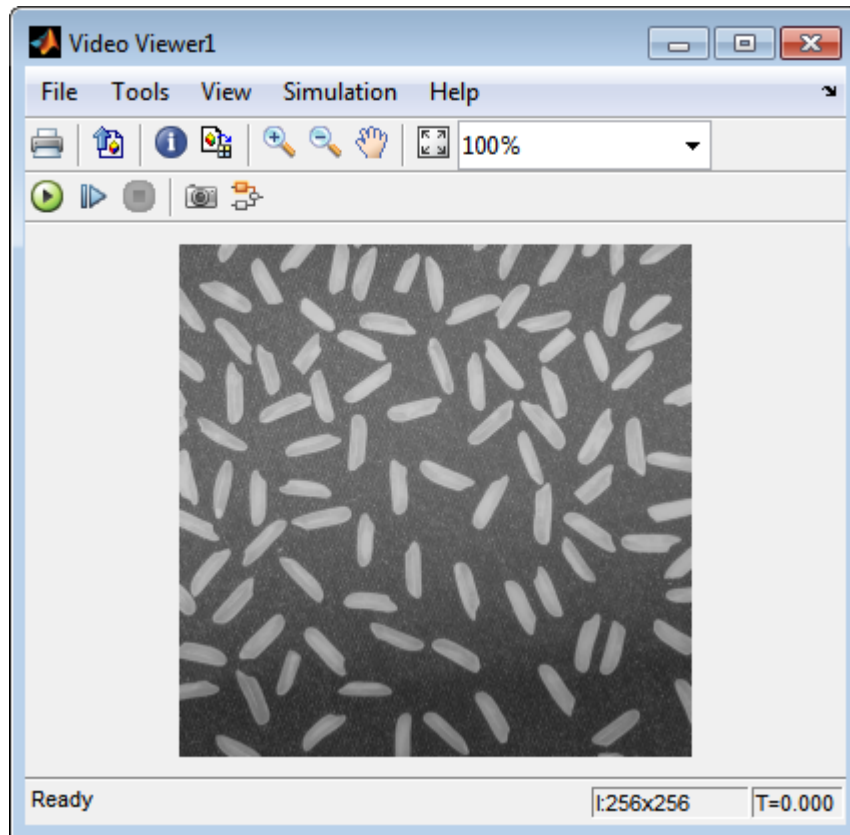
The binary image appears in the Video Viewer window.
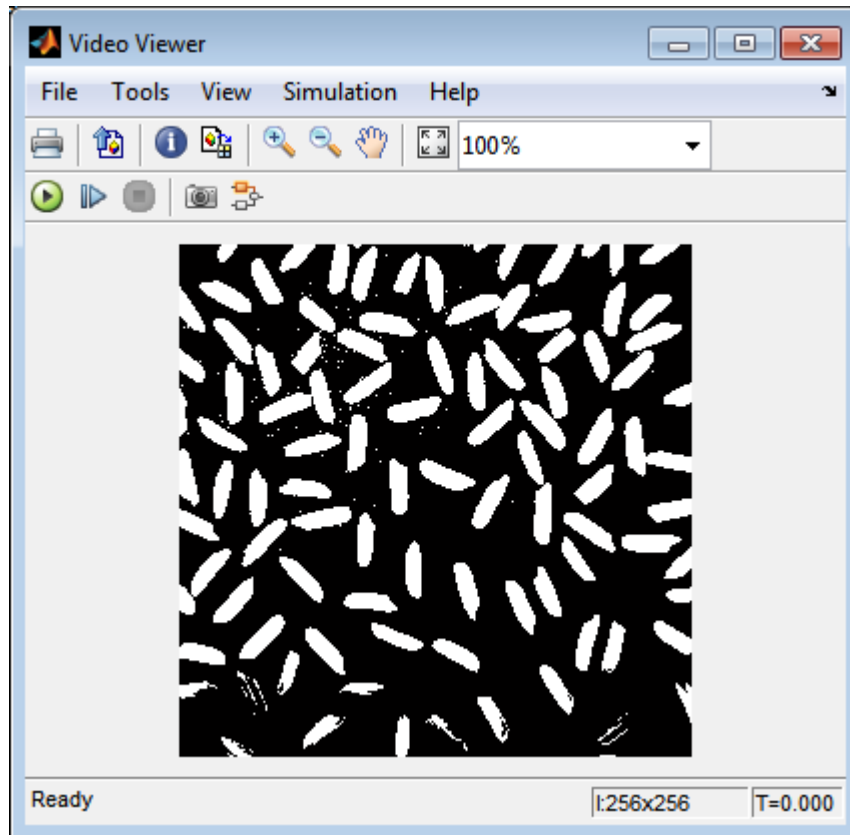
In the model window, the Display block shows the threshold value, calculated by the Autothreshold block, that separated the rice grains from the background.
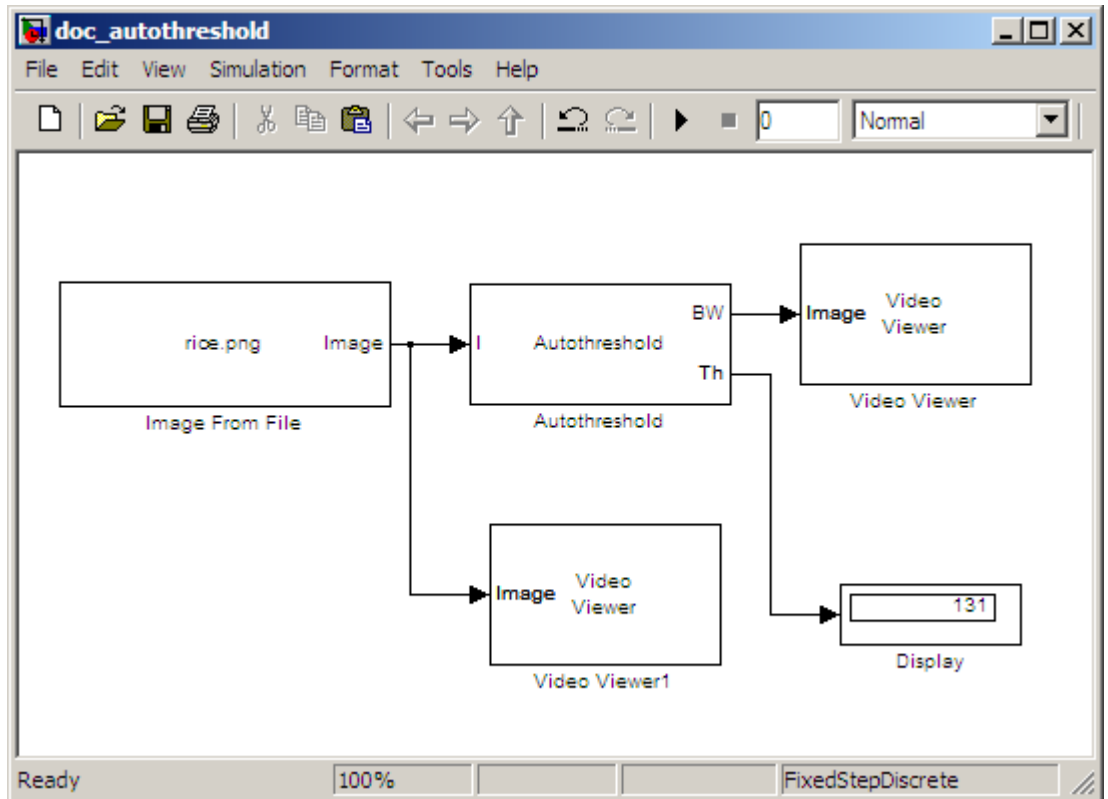
You have used the Autothreshold block to convert an intensity image to a binary image. For more information about this block, see the Autothreshold block reference page in the *Computer Vision System Toolbox Reference*. To open an example model that uses this block, type `vipstaples` at the MATLAB command 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)
```
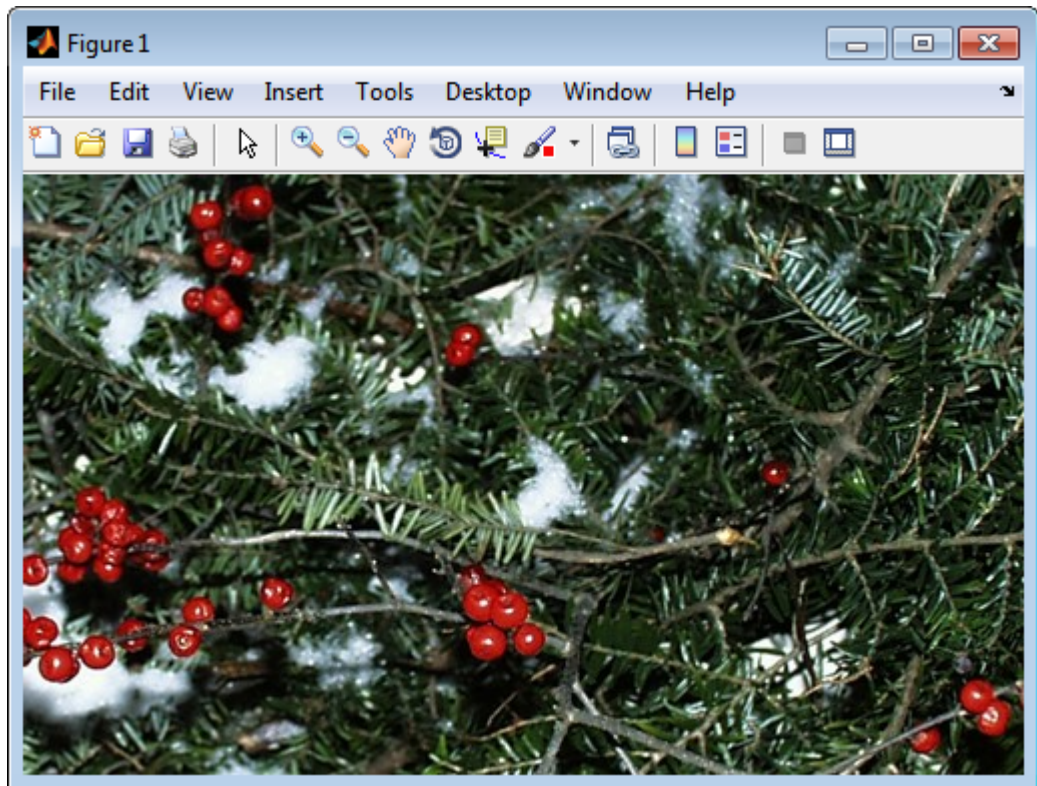
3   Create a new Simulink model, and add to it the blocks shown in the following table.

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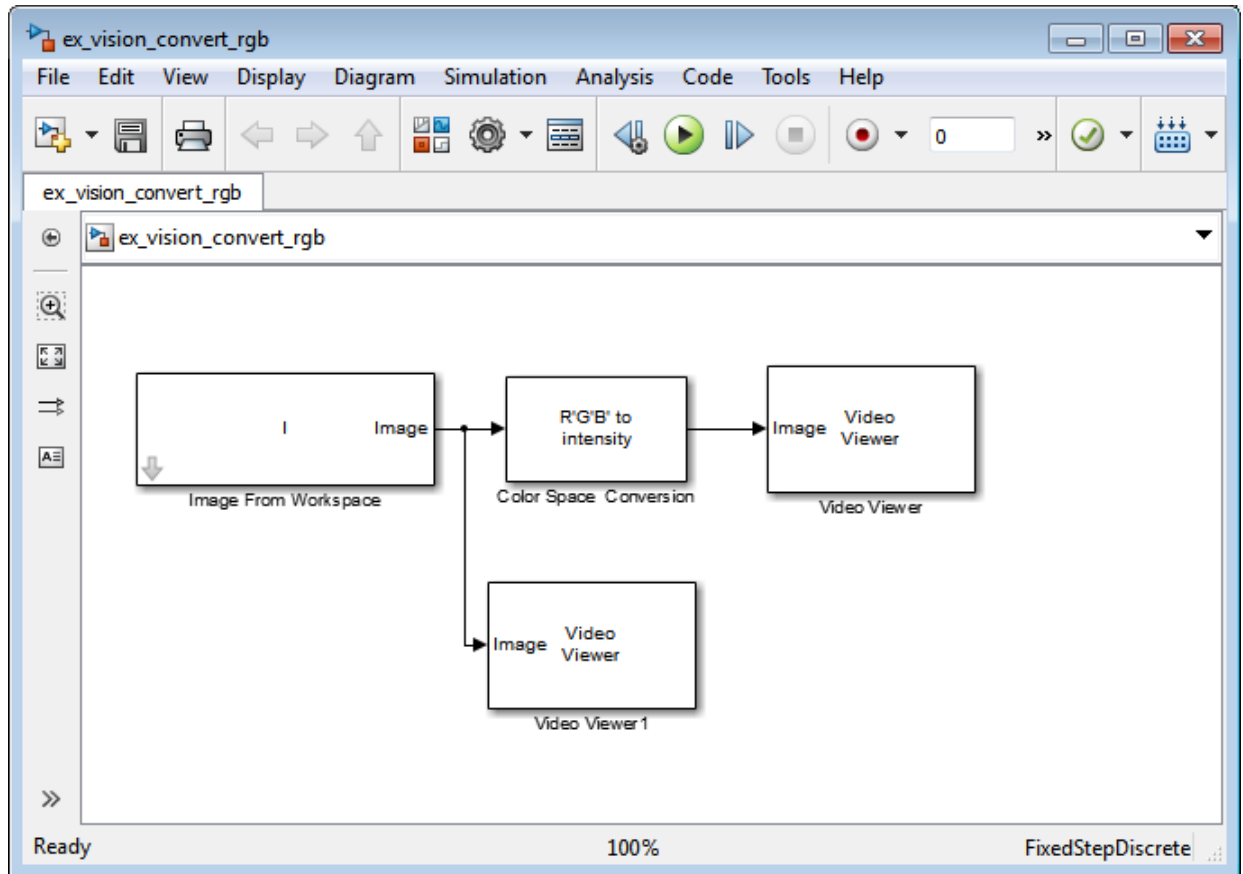
8   Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

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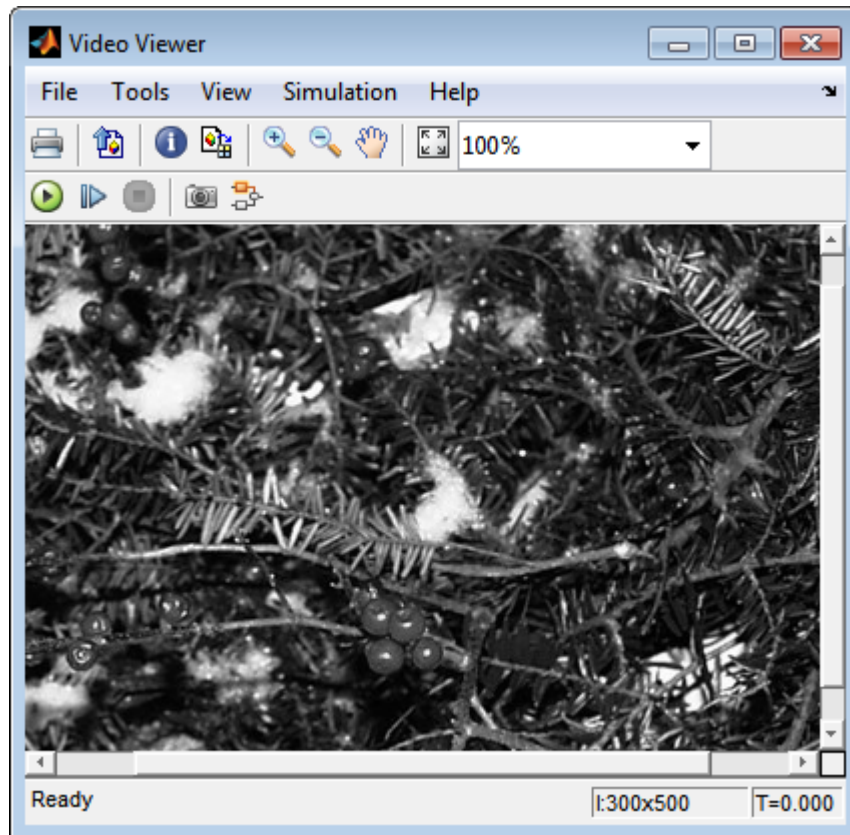
In this topic, you used the Color Space Conversion block to convert color information from the R'G'B' color space to intensity. For more information on this block, see the Color Space Conversion block reference page.

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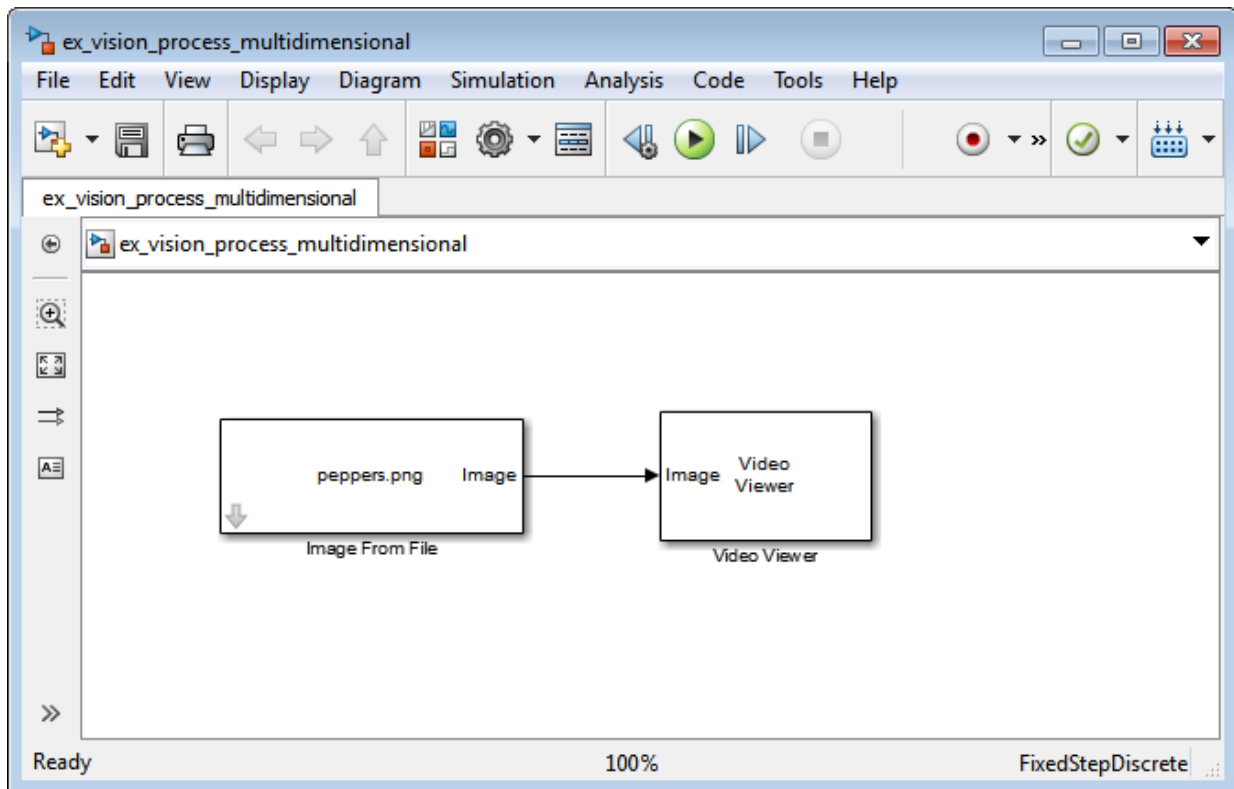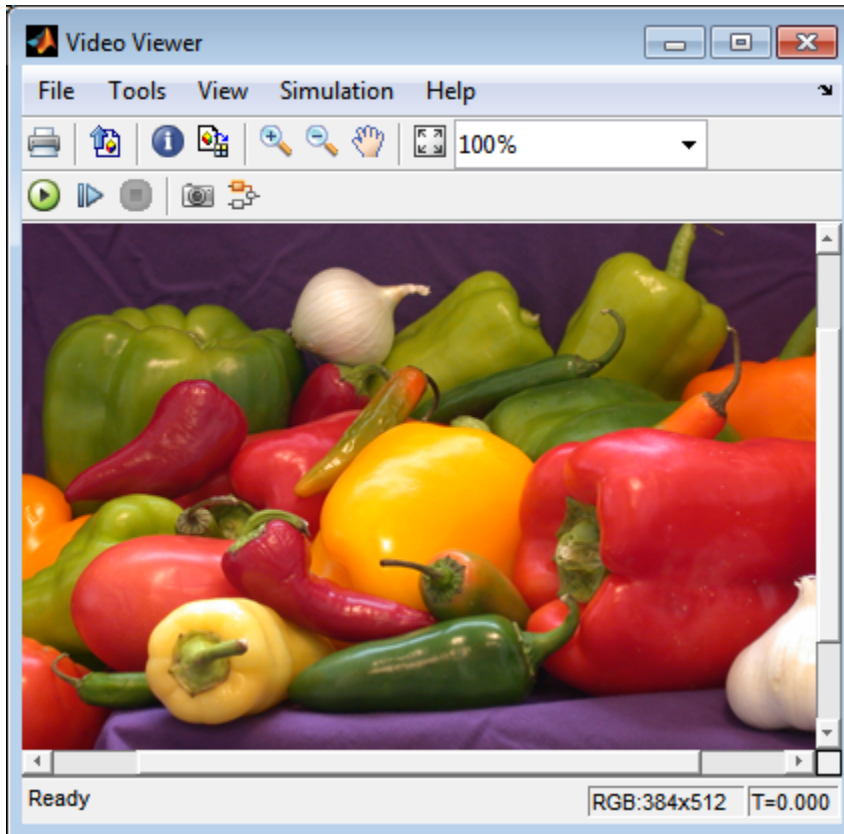
You can choose to process the image as a multidimensional array by setting the **Image signal** parameter to `One multidimensional signal` in the Image From File block dialog box.

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`
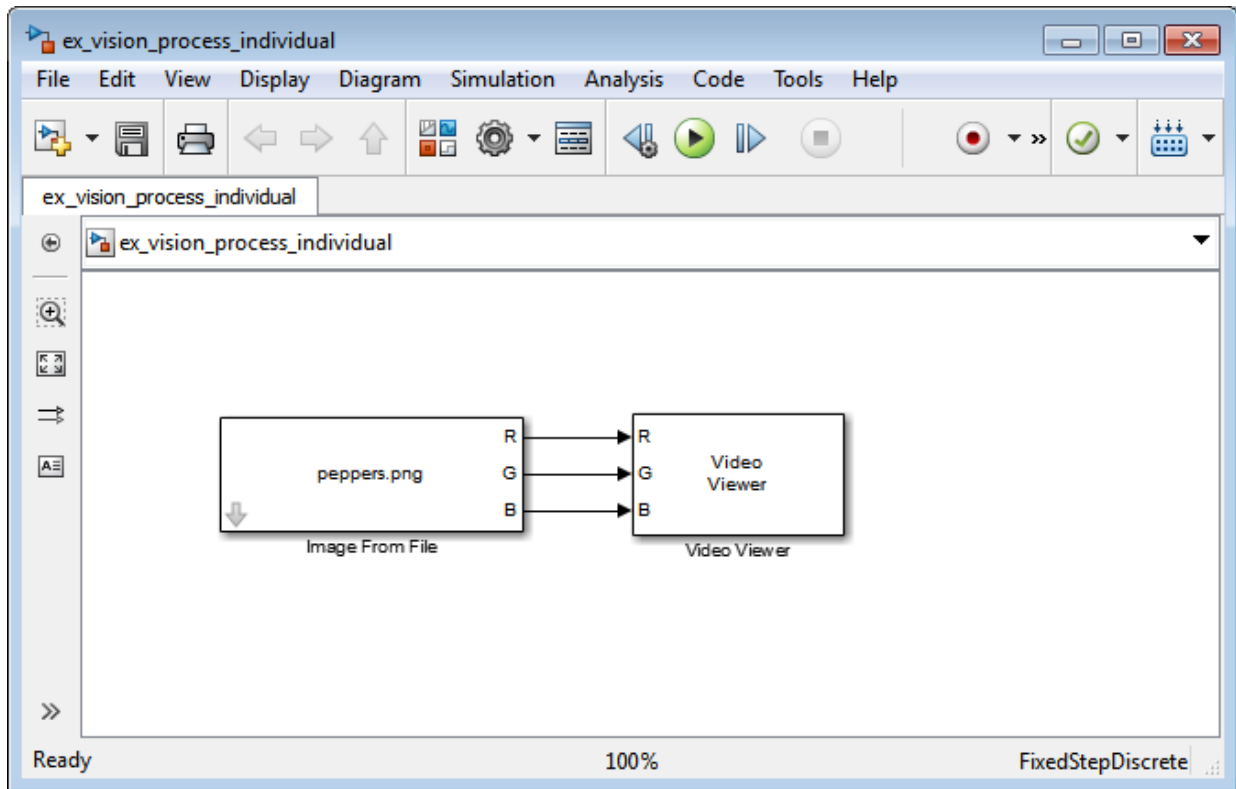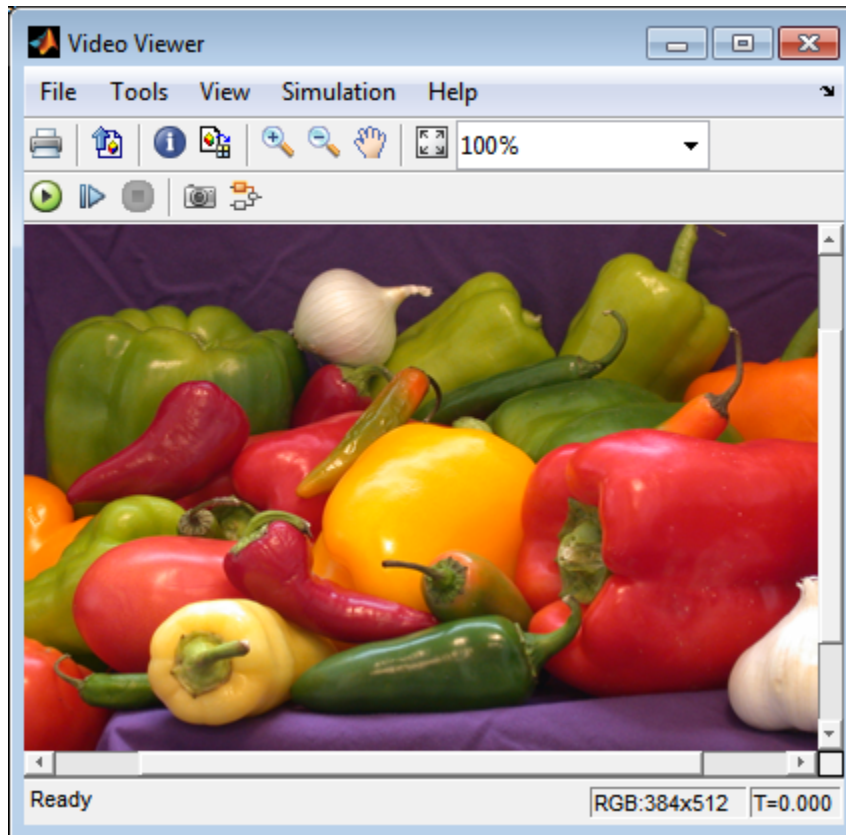
To process the individual color planes of an image or video signal, set the **Image signal** parameter to `Separate color signals` in both the Image From File and Video Viewer block dialog boxes.
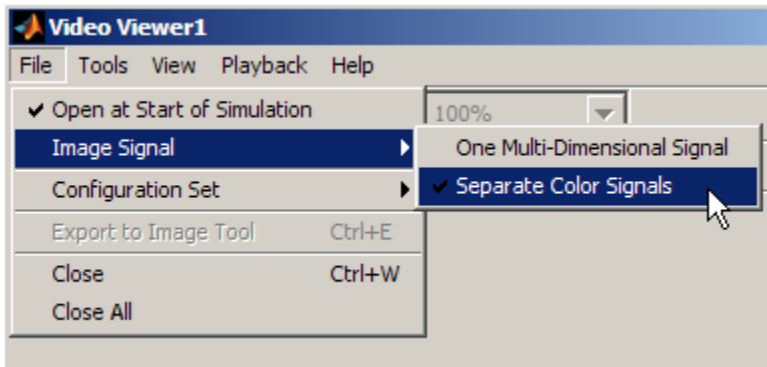
Note The ability to output separate color signals is a legacy option. It is recommend that you use multidimensional signals to represent color data.

If you are working with a block that only outputs multidimensional arrays, you can use the Selector block to separate the color planes. If you are working with a block that only accepts multidimensional arrays, you can use the Matrix Concatenation block to create a multidimensional array. For an example of this process, see Find the Histogram of an Image on page 11-2.

Video Formats

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 **Display** menu, point to **Signals & Ports**, 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" (Image Processing Toolbox).

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" (Image Processing Toolbox).

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" (Image Processing Toolbox).

5

Display and Graphics

Display, Stream, and Preview Videos

In this section...

View Streaming Video in MATLAB

Basic Video Streaming

Use the video player `vision.VideoPlayer` System object when you require a simple video display in MATLAB for streaming video.

Code Generation Supported Video Streaming Object

Use the deployable video player `vision.DeployableVideoPlayer` System object as a basic display viewer designed for optimal performance. This object supports code generation on all platforms.

Preview Video in MATLAB

Use the Image Processing Toolbox `implay` function to view and represent videos as variables in the MATLAB workspace. It is a full featured video player with toolbar controls. The `implay` player enables you to view videos directly from files without having to load all the video data into memory at once.

You can open several instances of the `implay` function simultaneously to view multiple video data sources at once. You can also dock these `implay` players 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 players.

View Video in Simulink

Code Generation Supported Video Streaming 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.

Simulation Control and Video Analysis 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 player interface. The block integrates play, pause, and step features while running the model and also provides video analysis tools such as pixel region viewer.

View Video Signals Without Adding Blocks

The `implay` function enables you to view video signals in Simulink models without adding blocks to your model. You can open several instances of the `implay` player simultaneously to view multiple video data sources at once. You can also dock these players 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 players.

Set Simulink simulation mode to `Normal` to use `implay`. `implay` does not work when you use "Accelerating Simulink Models" on page 13-8.

Example 5.1. Use `implay` to view a Simulink signal:

1   Open a Simulink model.

2   Open an `implay` player by typing `implay` on the MATLAB command line.

3   Run the Simulink model.

4   Select the signal line you want to view.

5
On the `implay` toolbar, select **File** > **Connect to Simulink Signal** or click the icon.

The video appears in the player window.

6   You can use multiple `implay` players to display different Simulink signals.

Note During code generation, the Simulink Coder™ does not generate code for the `implay` player.

Annotate Video Files with Frame Numbers

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

You can open the example model by typing at the MATLAB command line.

`ex_vision_annotate_video_file_with_frame_numbers`

1   Run your model.
2   The model displays the original and annotated videos.
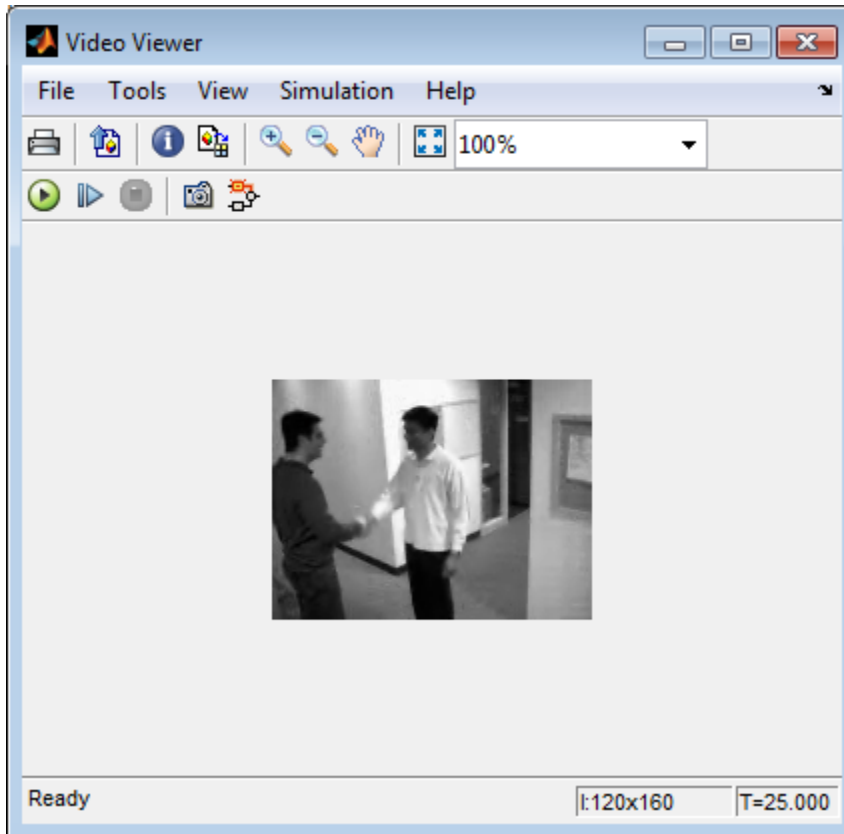
Color Formatting

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]** = [85 2]
- **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 character vector. 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 it's location on the image. The table shows the format for the points input for the different shapes.
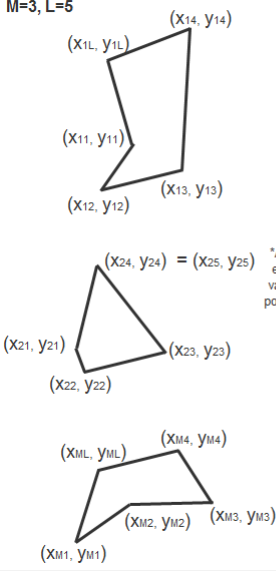
Rectangle

Shape	PTS input	Drawn Shape
Single Rectangle	Four-element row vector `[x y width height]` where • 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.	(x,y) $\quad$ (x + width-1, y + height-1)
M Rectangles	*M*-by-4 matrix $$\begin{bmatrix} x_1 & y_1 & width_1 & height_1 \\ x_2 & y_2 & width_2 & height_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_M & y_M & width_M & height_M \end{bmatrix}$$ where each row of the matrix corresponds to a different rectangle and is of the same form as the vector for a single rectangle.	M=2 (x_1,y_1) $(x_1 + width_1\text{-}1, y_1 + height_1\text{-}1)$ (x,y) $(x_2 + width_2\text{-}1, y_2 + height_2\text{-}1)$

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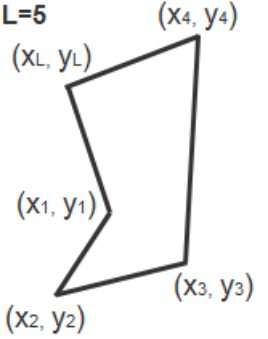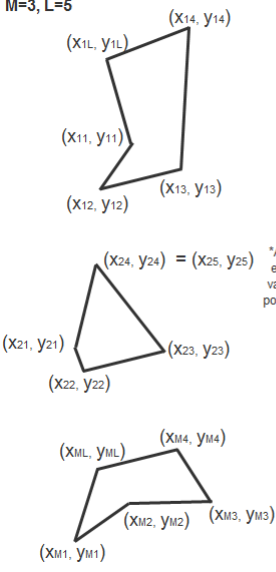
You can draw one or more lines, and one or more polylines. A polyline contains a series of connected line segments.

Shape	PTS input	Drawn Shape
Single Line	Four-element row vector $[x_1 \ y_1 \ x_2 \ y_2]$ where • 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.	
Single Polyline with (L-1) Segments	Vector of size $2L$, where L is the number of vertices, with format, $[x_1, \ y_1, \ x_2, \ y_2, \ \ldots, \ x_L, \ y_L]$. • 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 $(L-1)^{th}$ line segment. 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.	

Shape	PTS input	Drawn Shape
M Polylines with (*L*-1) Segments	2*L*-by-*N* matrix 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. The block produces an error message if the number of rows is less than two or not a multiple of two.	
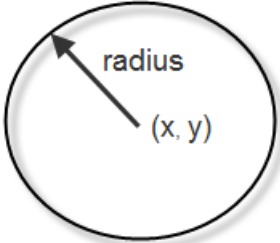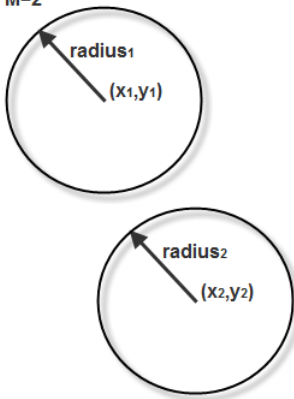
Polygon

You can draw one or more polygons.

Shape	PTS input	Drawn Shape
Single Polygon with L line segments	Row vector of size $2L$, where L is the number of vertices, with format, $[x_1 \ y_1 \ x_2 \ y_2 \ \ldots \ x_L \ y_L]$ where • 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 $(L\text{-}1)^{\text{th}}$ line segment and the beginning of the Lth line segment. The block connects $[x_1 \ y_1]$ to $[x_L \ y_L]$ to complete the polygon. The block produces an error if the number of rows is negative or not a multiple of two.	
M Polygons with the largest number of line segments in any line being L	M-by-$2L$ matrix 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. The block produces an error message if the number of rows is less than two or is not a multiple of two.	

Circle

You can draw one or more circles.

Shape	PTS input	Drawn Shape
Single Circle	Three-element row vector `[x y radius]` where • x and y are coordinates for the center of the circle. • radius is the radius of the circle, which must be greater than 0.	
M Circles	*M-by-3* matrix $$\begin{bmatrix} x_1 & y_1 & radius_1 \\ x_2 & y_2 & radius_2 \\ \vdots & \vdots & \vdots \\ x_M & y_M & radius_M \end{bmatrix}$$ where each row of the matrix corresponds to a different circle and is of the same form as the vector for a single circle.	

6

Registration and Stereo Vision

Detect Edges in Images

This example shows how to find the edges of rice grains in an intensity image. It finds 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`
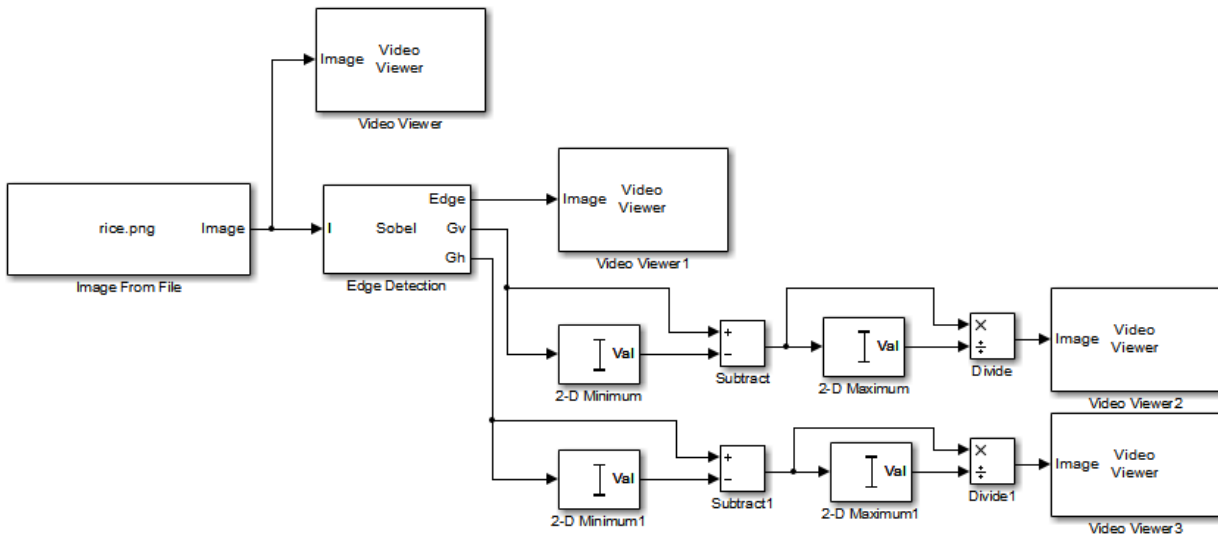
Set block parameters.

Block	Parameter setting
Image From File	• **File name** to `rice.png`.
	• **Output data type** to `single`.

Block	Parameter setting
Edge Detection	Use the Edge Detection block to find the edges in the image. • **Output type** = `Binary image and gradient components` • Select the **Edge thinning** check box.
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

Run your model and view edge detection results.

The Video Viewer window displays the original image.
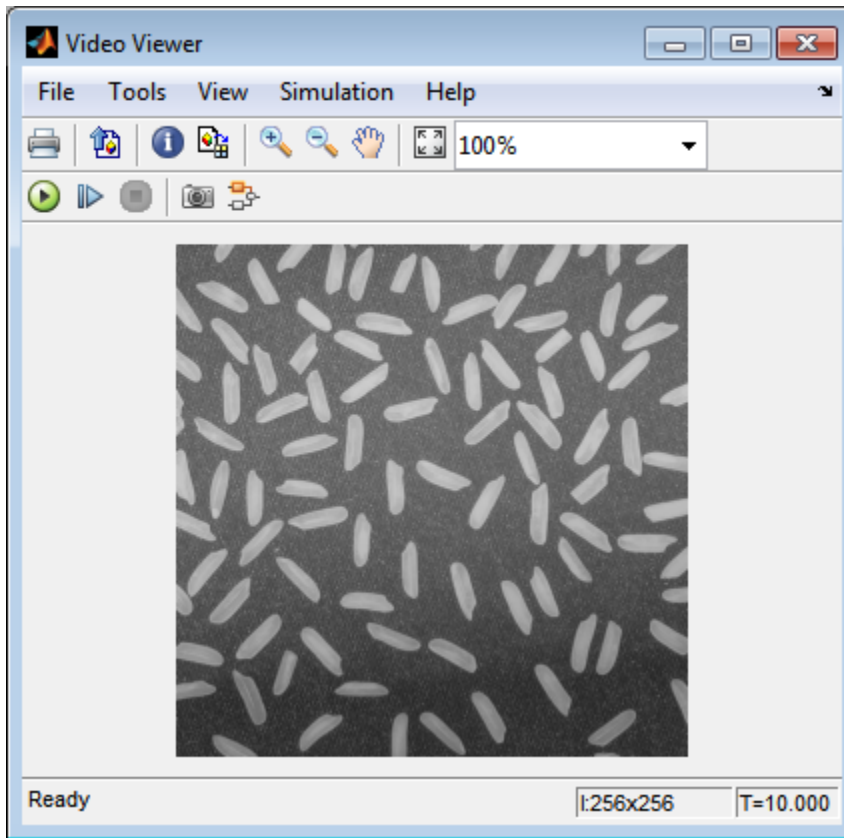
The Video Viewer1 window displays the edges of the rice grains in white and the background in black.
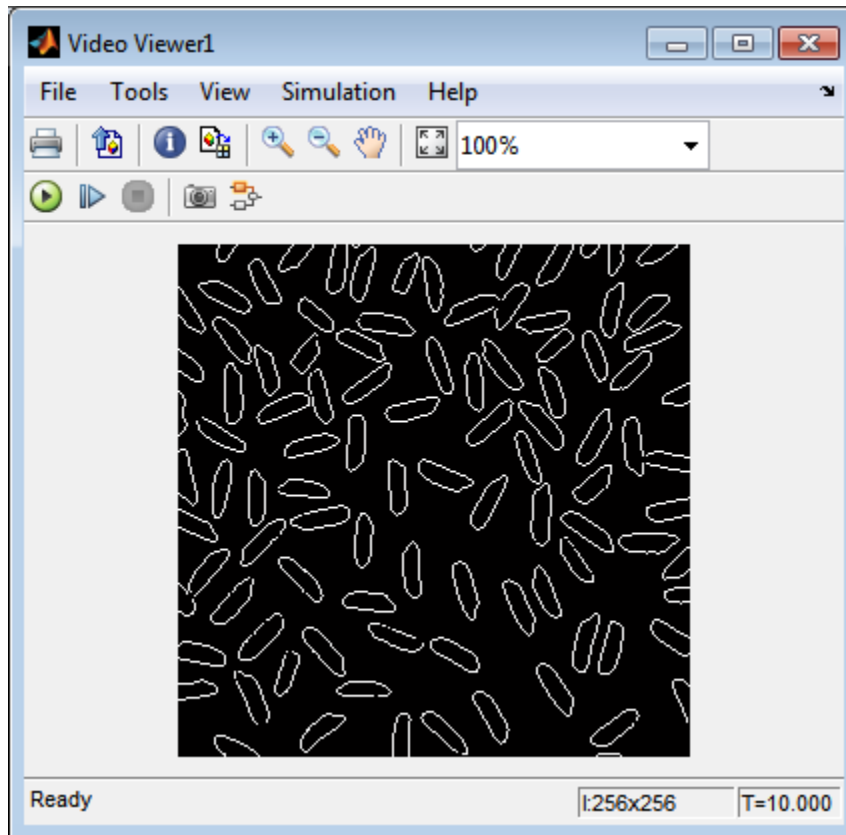
The Video Viewer2 window displays the intensity image of the vertical gradient components of the image. You can see that the vertical edges of the rice grains are darker and more well defined than the horizontal edges.
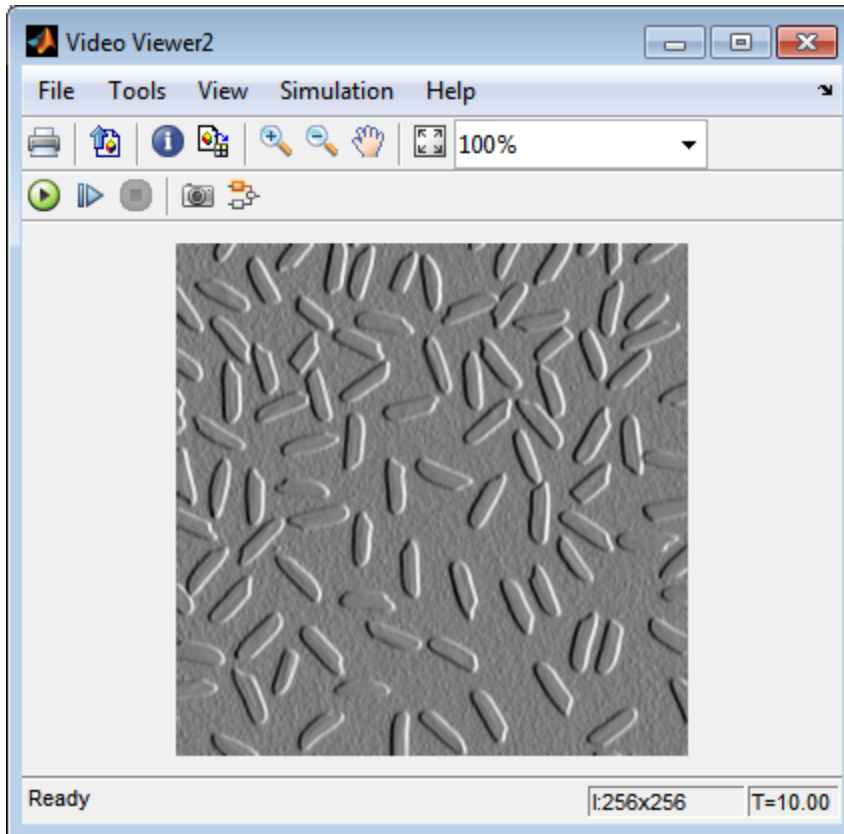
The Video Viewer3 window displays the intensity image of the horizontal gradient components of the image. In this image, the horizontal edges of the rice grains are more well defined.
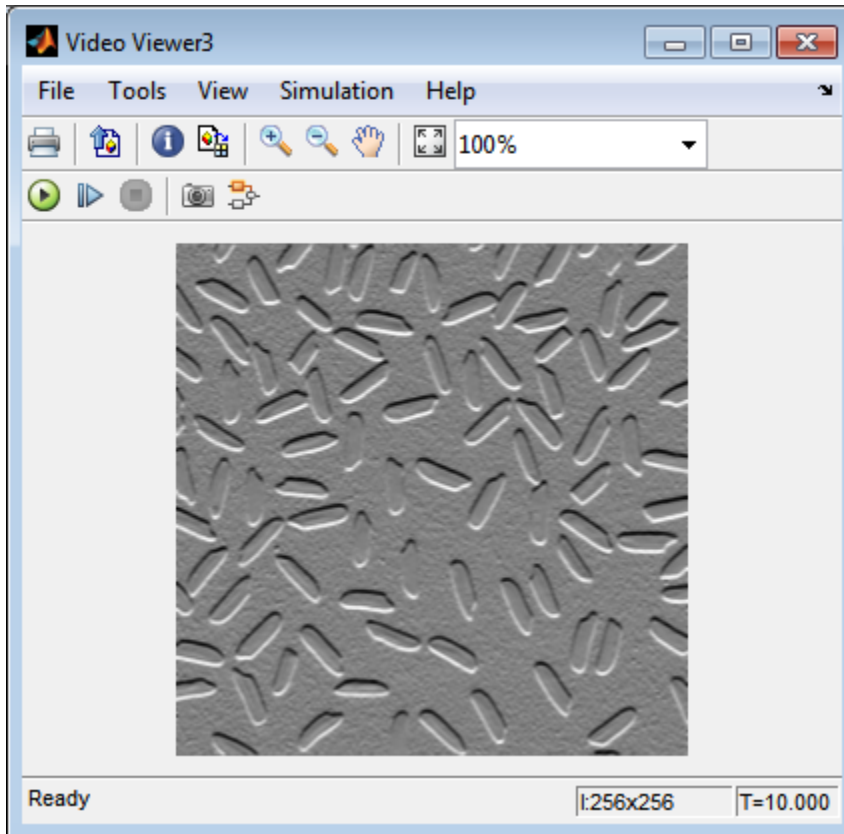
The Edge Detection block convolves the input matrix with the Sobel kernel. This calculates the gradient components of the image that correspond to the horizontal and vertical edge responses. The block outputs these components at the **Gh** and **Gv** ports, respectively. Then the block performs a thresholding operation on the gradient components to find the binary image. The binary image is a matrix filled with 1s and 0s. The nonzero elements of this matrix correspond to the edge pixels and the zero elements correspond to the background pixels. The block outputs the binary image at the **Edge** port.

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

Run the model faster by double-clicking the Edge Detection block and clear the **Edge thinning** check box.

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.
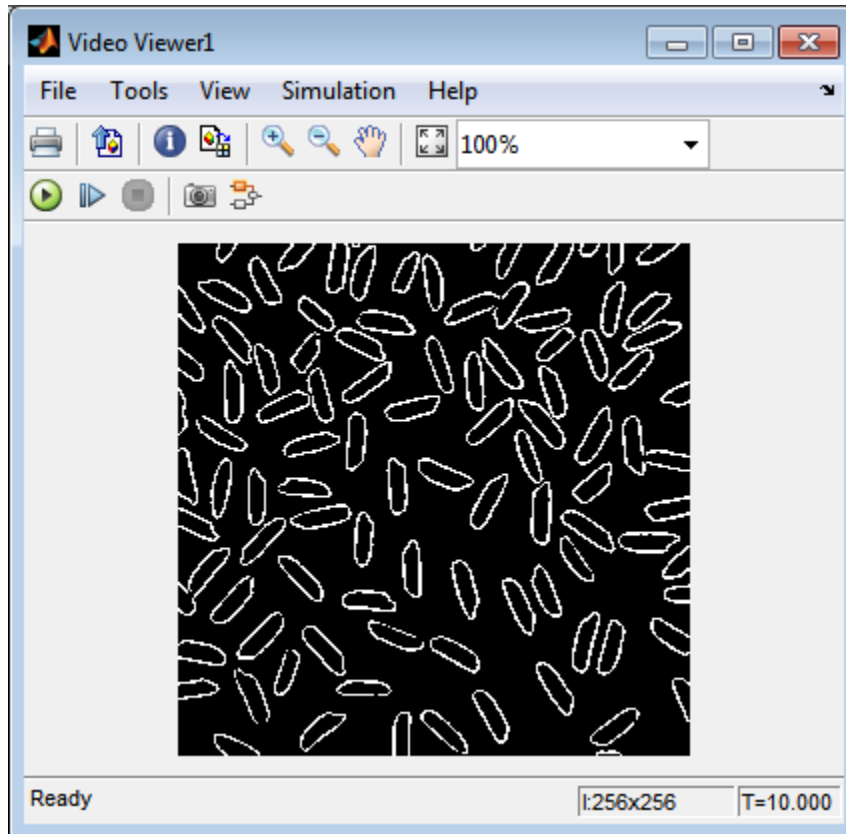
Close the model.

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

Detect Lines in Images

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

You can open the example model by typing at the MATLAB command line.

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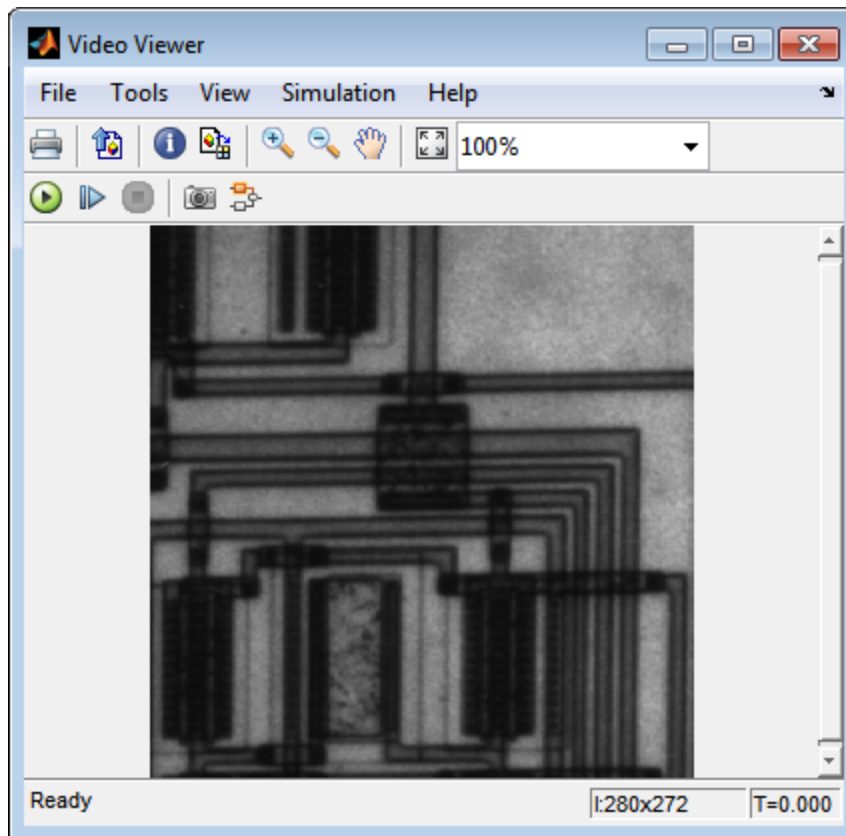
The Video Viewer blocks display the original image, the image with all edges found, and the image with the longest line annotated.

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
Rotation Correction	`videorotationcorrection`	`viphough`

Setting Block Parameters

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

Block	Parameter setting
Selector, Selector1	The Selector blocks separate the indices of the rho and theta values, which the Find Local Maxima block outputs at the **Idx** port. The rho and theta values correspond to the maximum value in the Hough matrix. The Selector blocks parameters are set as follows: • **Number of input dimensions:** 1 • **Index mode** = One-based • **Index Option** = Index vector (port) • **Input port size** = 2
Selector2, Selector3	The 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 Selector blocks are set as follows: • **Number of input dimensions:** 2 • **Index mode** = One-based • **Index Option** = Index vector (port)
Hough Lines	The Hough Lines block determines where the longest line intersects the edges of the original image. • **Sine value computation method** = Trigonometric function

Block	Parameter setting
Draw Shapes	The Draw Shapes block draws a white line over the longest line on the original image. The coordinates are set to superimpose a line on the original image. The block parameters are set as follows: • **Shape** = Lines • **Border color** = White

Configuration Parameters

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

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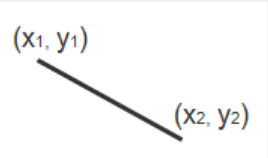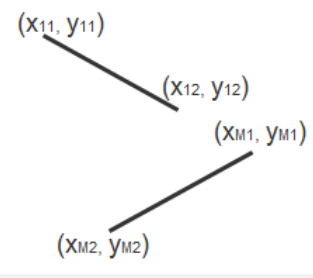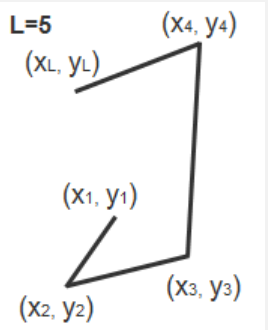- **Solver** pane, **Solver** = Discrete (no continuous states)
- **Solver** pane, **Fixed-step size (fundamental sample time):** = 0.2

Fisheye Calibration Basics

Camera calibration is the process of computing the extrinsic and intrinsic parameters of a camera. Once you calibrate a camera, you can use the image information to recover 3-D information from 2-D images. You can also undistort images taken with a fisheye camera.

Fisheye cameras are used in odometry and to solve the simultaneous localization and mapping (SLAM) problems visually. Other applications include, surveillance systems, GoPro, virtual reality (VR) to capture 360 degree field of view (fov), and stitching algorithms. These cameras use a complex series of lenses to enlarge the camera's field of view, enabling it to capture wide panoramic or hemispherical images. However, the lenses achieve this extremely wide angle view by distorting the lines of perspective in the images

Fisheye image   Undistorted fisheye image

Because of the extreme distortion a fisheye lens produces, the pinhole model cannot model a fisheye camera.
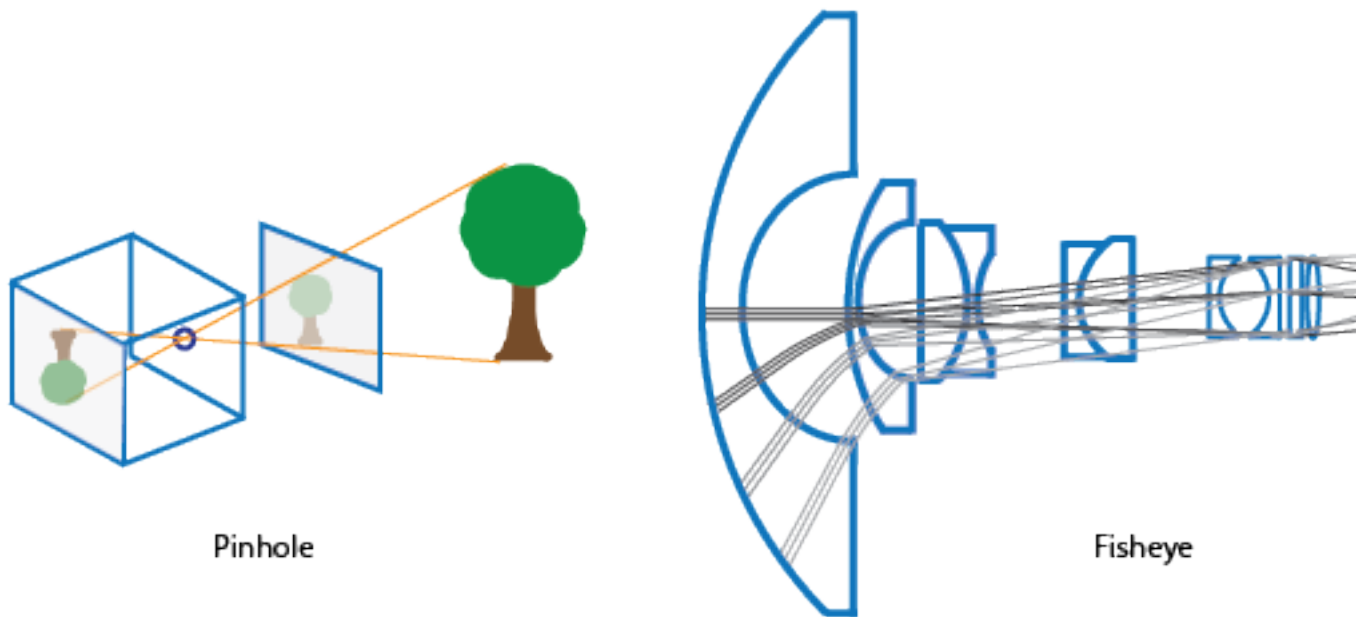
Fisheye Camera Model

The Computer Vision System Toolbox™ calibration algorithm uses the fisheye camera model proposed by Scaramuzza[1]. The model uses an omnidirectional camera model. The process treats the imaging system as a compact system. In order to relate a 3-D world point on to a 2-D image, you must obtain the camera extrinsic and intrinsic parameters. World points are transformed to camera coordinates using the extrinsics parameters. The camera coordinates are mapped into the image plane using the intrinsics parameters.
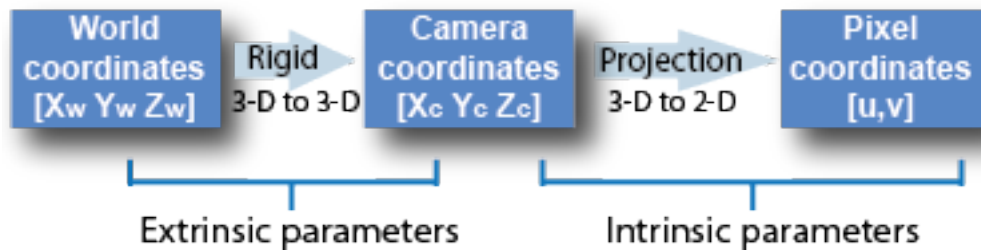
Extrinsic Parameters

The extrinsic parameters consist of a rotation, R, and a translation, t. The origin of the camera's coordinate system is at its optical center and its x- and y- axis define the image plane.
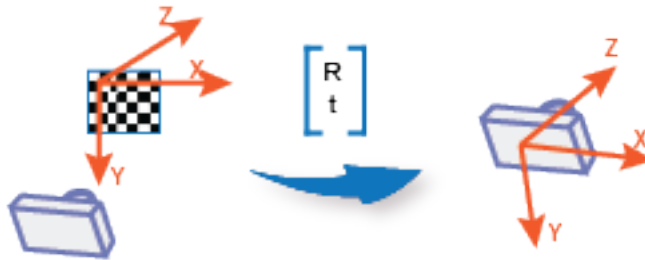
The transformation from world points to camera points is:

$$\begin{pmatrix} X_C \\ Y_C \\ Z_C \end{pmatrix} = R \begin{pmatrix} X_W \\ Y_W \\ Z_W \end{pmatrix} + T$$

Camera points   Rotation   World points   Translation

Intrinsic Parameters

For the fisheye camera model, the intrinsic parameters include the polynomial mapping coefficients of the projection function. The alignment coefficients are related to sensor alignment and the transformation from the sensor plane to a pixel location in the camera image plane.

The following equation maps an image point into its corresponding 3-D vector.

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = \lambda \begin{pmatrix} u \\ v \\ a_0 + a_2\rho^2 + a_3\rho^3 + a_4\rho^4 \end{pmatrix}$$

- (u, v) are the ideal image projections of the real-world points.

- λ represents a scalar factor.

- a_0, a_2, a_3, a_4 are polynomial coefficents described by the Scaramuzza model, where $a_1 = 0$.

- ρ is a function of (u,v) and depends only on the distance of a point from the image center: $\rho = \sqrt{u^2 + v^2}$.

The intrinsic parameters also account for stretching and distortion. The stretch matrix compensates for the sensor-to-lens misalignment, and the distortion vector adjusts the (0,0) location of the image plane.
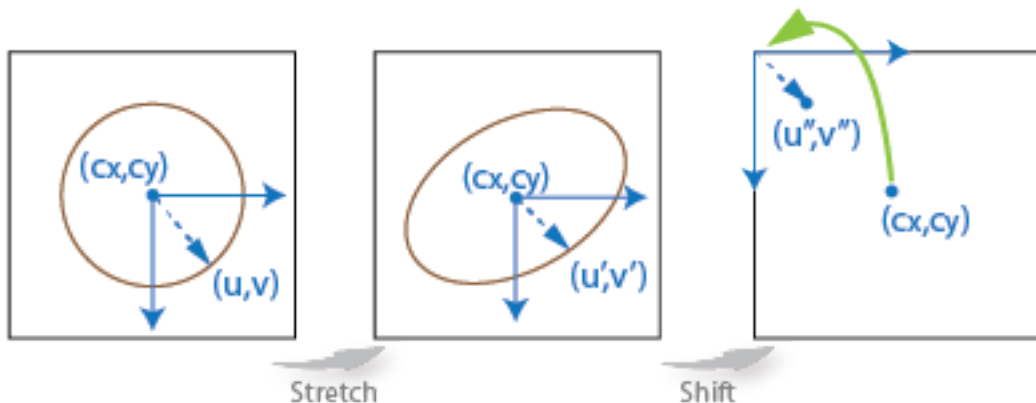
The following equation relates the real distorted coordinates (u'',v'') to the ideal distorted coordinates (u,v).

$$\begin{pmatrix} u'' \\ v'' \end{pmatrix} = \begin{pmatrix} c & d \\ e & 1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \end{pmatrix}$$

Image pixels   Stretch matrix   Hypothetical image plane   Distortion center

Fisheye Camera Calibration in MATLAB

To remove lens distortion from a fisheye image, you can detect a checkerboard calibration pattern and then calibrate the camera. You can find the checkerboard points using the `detectCheckerboardPoints` and `generateCheckerboardPoints` functions. The `estimateFisheyeParameters` function uses the detected points and returns the `fisheyeParameters` object that contains the intrinsic and extrinsic parameters of a fisheye camera. You can use the `fisheyeCalibrationErrors` to check the accuracy of the calibration.

Correct Fisheye Image for Lens Distortion

Remove lens distortion from a fisheye image by detecting a checkboard calibration pattern and calibrating the camera. Then, display the results.

Gather a set of checkerboard calibration images.

```
images = imageDatastore(fullfile(toolboxdir('vision'),'visiondata', ...
    'calibration','gopro'));
```

Detect the calibration pattern from the images.

```
[imagePoints,boardSize] = detectCheckerboardPoints(images.Files);
```

Generate world coordinates for the corners of the checkerboard squares.

```
squareSize = 29; % millimeters
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Estimate the fisheye camera calibration parameters based on the image and world points. Use the first image to get the image size.

```
I = readimage(images,1);
imageSize = [size(I,1) size(I,2)];
params = estimateFisheyeParameters(imagePoints,worldPoints,imageSize);
```

Remove lens distortion from the first image I and display the results.

```
J1 = undistortFisheyeImage(I,params.Intrinsics);
figure
imshowpair(I,J1,'montage')
title('Original Image (left) vs. Corrected Image (right)')

J2 = undistortFisheyeImage(I,params.Intrinsics,'OutputView','full');
figure
imshow(J2)
title('Full Output View')
```


Original Image (left) vs. Corrected Image (right)

Full Output View
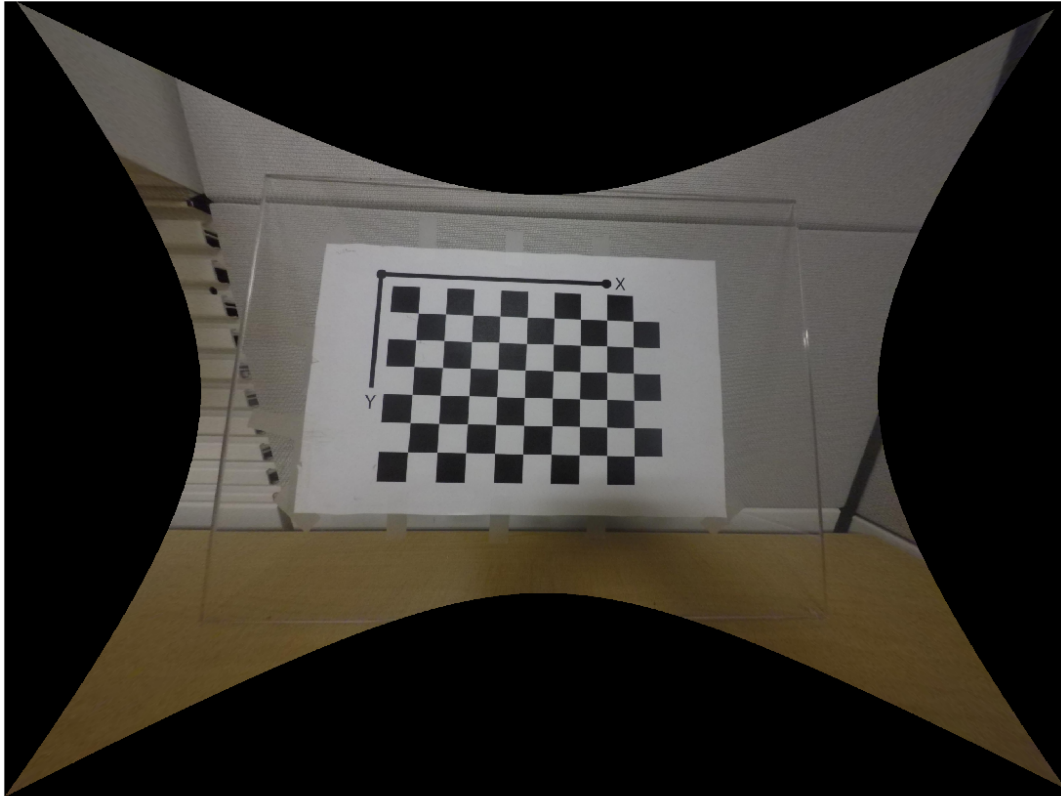
References

[1] Scaramuzza, D., A. Martinelli, and R. Siegwart. "A Toolbox for Easy Calibrating Omnidirectional Cameras." *Proceedings to IEEE International Conference on Intelligent Robots and Systems, (IROS)*. Beijing, China, October 7–15, 2006.

See Also

estimateFisheyeParameters | fisheyeCalibrationErrors |
fisheyeIntrinsics | fisheyeIntrinsicsEstimationErrors |
fisheyeParameters | undistortFisheyeImage | undistortFisheyePoints

Related Examples

- "Monocular Visual Odometry"

Single Camera Calibrator App

In this section...

Camera Calibrator Overview

You can use the **Camera Calibrator** app to estimate camera intrinsics, extrinsics, and lens distortion parameters. You can use these camera parameters for various computer vision applications. These applications include removing the effects of lens distortion from an image, measuring planar objects, or reconstructing 3-D scenes from multiple cameras.

The suite of calibration functions used by the **Camera Calibrator** app provide the workflow for camera calibration. You can use these functions directly in theMATLAB workspace. For a list of functions, see "Single Camera Calibration".

Single Camera Calibration
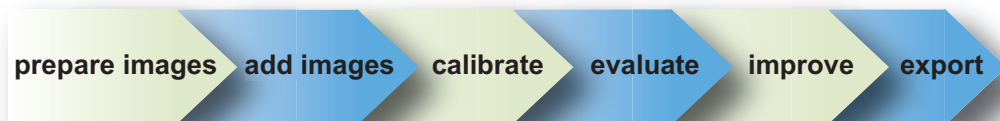
Follow this workflow to calibrate your camera using the app:

1. Prepare images, camera, and calibration pattern.

2. Add images and select standard or fisheye camera model.

3. Calibrate the camera.

4. Evaluate calibration accuracy.

5. Adjust parameters to improve accuracy (if necessary).

6. Export the parameters object.

In some cases, the default values work well, and you do not need to make any improvements before exporting parameters. You can also make improvements using the camera calibration functions directly in the MATLAB workspace. For a list of functions, see "Single Camera Calibration".

Open the Camera Calibrator

- MATLAB Toolstrip: On the **Apps** tab, in the **Image Processing and Computer Vision** section, click the **Camera Calibrator** icon.

- MATLAB command prompt: Enter `cameraCalibrator`

Prepare the Pattern, Camera, and Images

To better the results, use between 10 and 20 images of the calibration pattern. The calibrator requires at least three images. Use uncompressed images or lossless compression formats such as PNG. The calibration pattern and the camera setup must satisfy a set of requirements to work with the calibrator. For greater calibration accuracy, follow these instructions for preparing the pattern, setting up the camera, and capturing the images.

Note The Camera Calibrator app supports only checkerboard patterns. If you are using a different type of calibration pattern, you can still calibrate your camera using the `estimateCameraParameters` function. Using a different type of pattern requires that you supply your own code to detect the pattern points in the image.

Prepare the Checkerboard Pattern

The **Camera Calibrator** app uses a checkerboard pattern. A checkerboard pattern is a convenient calibration target. If you want to use a different pattern to extract key points,

you can use the camera calibration MATLAB functions directly. See "Single Camera Calibration" for the list of functions.

You can print (from MATLAB) and use the checkerboard pattern provided. The checkerboard pattern you use must not be square. One side must contain an even number of squares and the other side must contain an odd number of squares. Therefore, the pattern contains two black corners along one side and two white corners on the opposite side. This criteria enables the app to determine the orientation of the pattern. The calibrator assigns the longer side to be the *x*-direction.

To prepare the checkerboard pattern:

1 Attach the checkerboard printout to a flat surface. Imperfections on the surface can affect the accuracy of the calibration.

2 Measure one side of the checkerboard square. You need this measurement for calibration. The size of the squares can vary depending on printer settings.

Size of checkerboard square

3   To improve the detection speed, set up the pattern with as little background clutter as possible.

Camera Setup

To calibrate your camera, follow these rules:

- Keep the pattern in focus, but do not use autofocus.
- If you change zoom settings between images, the focal length changes.

Capture Images

For better results, use at least 10 to 20 images of the calibration pattern. The calibrator requires at least three images. Use uncompressed images or images in lossless compression formats such as PNG. For greater calibration accuracy:

- Capture the images of the pattern at a distance roughly equal to the distance from your camera to the objects of interest. For example, if you plan to measure objects from 2 meters, keep your pattern approximately 2 meters from the camera.
- Place the checkerboard at an angle less than 45 degrees relative to the camera plane.

- Do not modify the images, (for example, do not crop them).
- Do not use autofocus or change the zoom settings between images.
- Capture the images of a checkerboard pattern at different orientations relative to the camera.

- Capture a variety of images of the pattern so that you have accounted for as much of the image frame as possible. Lens distortion increases radially from the center of the image and sometimes is not uniform across the image frame. To capture this lens distortion, the pattern must appear close to the edges of the captured images.

The Calibrator works with a range of checkerboard square sizes. As a general rule, your checkerboard should fill at least 20% of the captured image. For example, the preceding images were taken with a checkerboard square size of 108 mm, as the following montage shows:
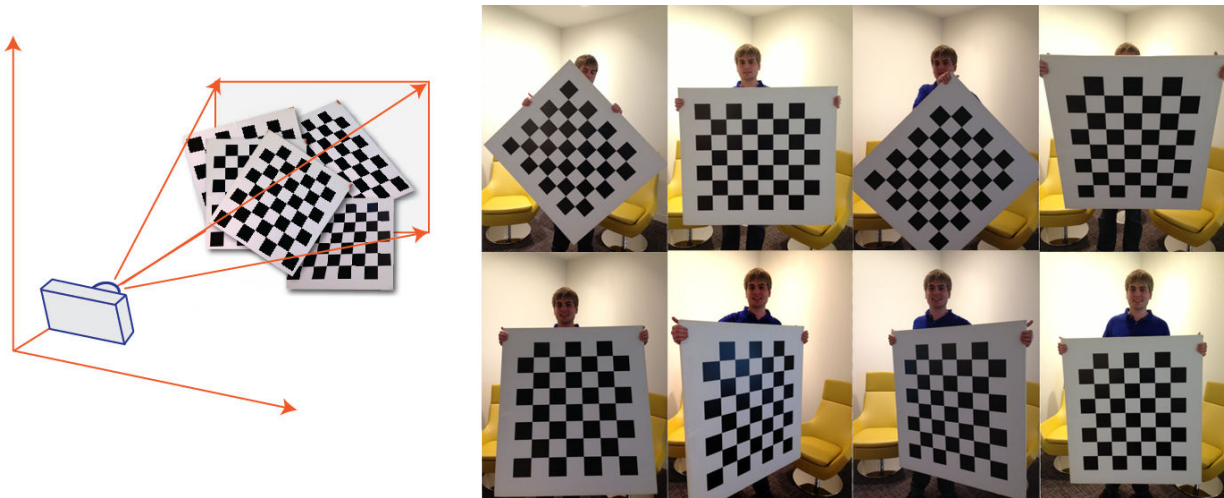
Add Images and Select Camera Model

To begin calibration, you must add images. You can add saved images from a folder or add images directly from a camera. The calibrator analyzes the images to ensure they meet the calibrator requirements. The calibrator then detects the points on the checkerboard.

Add Images from File

On the **Calibration** tab, in the **File** section, click Add images, and then select `From file`. You can add images from multiple folders by clicking Add images for each folder.

Acquire Live Images

To begin calibration, you must add images. You can acquire live images from a webcam using the MATLAB Webcam support. To use this feature, you must install MATLAB Support Package for USB Webcams installed to use this feature. See "Install the MATLAB Support Package for USB Webcams" (Image Acquisition Toolbox) for information on installing the support package. To add live images, follow these steps.

1   On the **Calibration** tab, in the **File** section, click **Add Images**, then select `From camera`.

    This action opens the **Camera** tab opens. If you have only one webcam connected to your system, it is selected by default and a live preview window opens. If you have

multiple cameras connected and want to use one different from the default, select that specific camera in the **Camera** list.

2 Set properties for the camera to control the image (optional). Click the **Camera Properties** to open a menu of the properties for the selected camera. This list varies depending on your device.

Use the sliders or drop-down list to change any available property settings. The Preview window updates dynamically when you change a setting. When you are done setting properties, click anywhere outside of the menu box to dismiss the properties list.

3 Enter a location for the acquired image files in the **Save Location** box by typing the path to the folder or using the **Browse** button. You must have permission to write to the folder you select.

4 Set the capture parameters.

- To set the number of seconds between image captures, use the **Capture Interval** box or slider. The default is 5 seconds, the minimum is 1 second, and the maximum is 60 seconds.

- To set the number of image captures, use the **Number of images to capture** box or slider. The default is 20 images, the minimum is 2 images, and the maximum is 100 images.

In the default configuration, a total of 20 images are captured, one every 5 seconds.

5 The Preview window shows the live images streamed as RGB data. After you adjust any device properties and capture settings, use the Preview window as a guide to line up the camera to acquire the checkerboard pattern image you want to capture.

6 Click the **Capture** button. The number of images you set are captured and the thumbnails of the snapshots appear in the **Data Browser** pane. They are automatically named incrementally and are captured as `.png` files.

You can optionally stop the image capture before the designated number of images are captured by clicking **Stop Capture**.

When you are capturing images of a checkerboard, after the designated number of images are captured, a Checkerboard Square Size dialog box displays. Specify the size of the checkerboard square, then click **OK**.
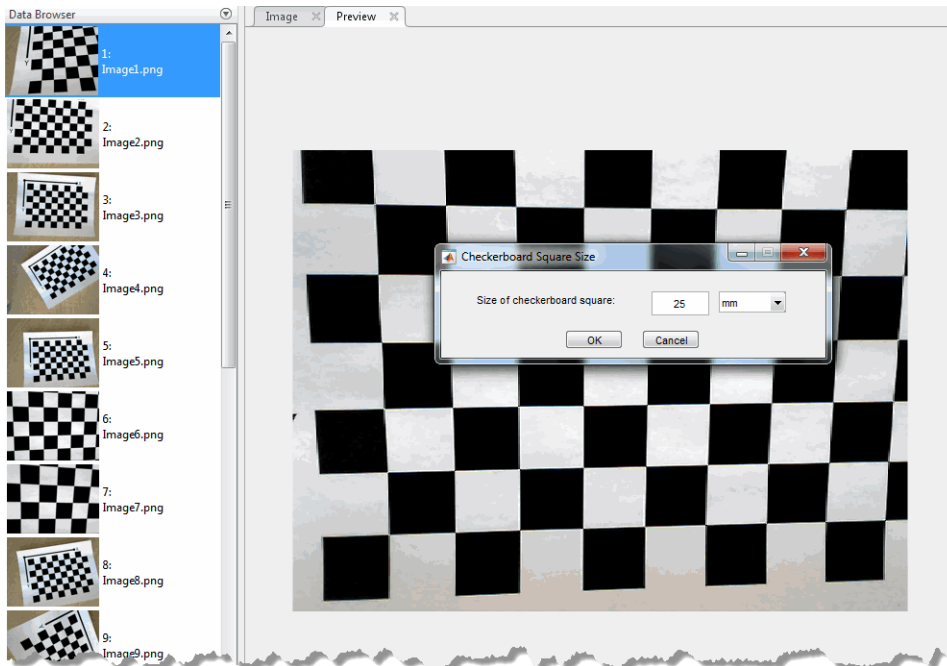
The detection results are then calculated and displayed. For example:
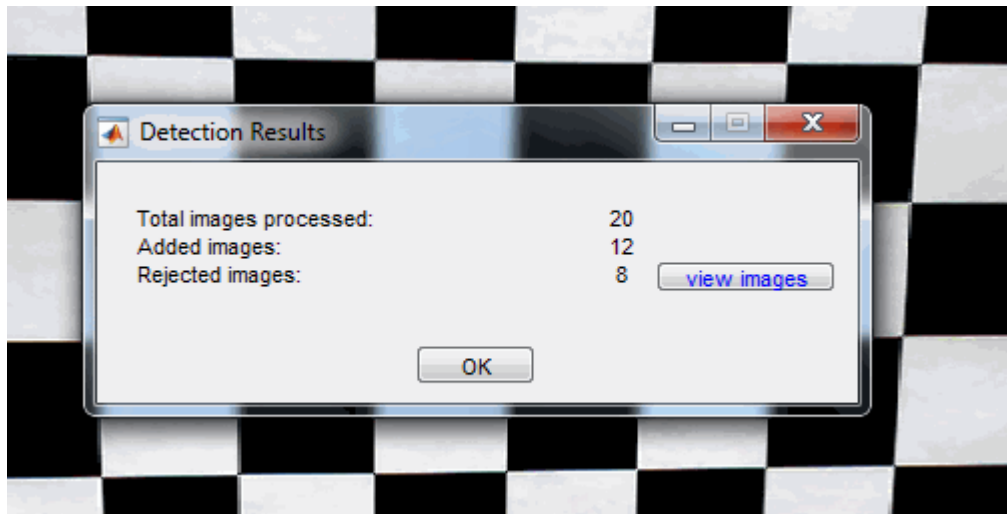
7 Click **OK** to dismiss the Detection Results dialog box.

8 When you have finished acquiring live images, click **Close Image Capture** to close the **Camera** tab.

Analyze Images

After you add the images, the Checkerboard Square Size dialog box appears. Specify size of the checkerboard square by entering the length of one side of a square from the checkerboard pattern.

The calibrator attempts to detect a checkerboard in each of the added images, displaying an Analyzing Images progress bar window, indicating detection progress. If any of the images are rejected, the Detection Results dialog box appears, which contains diagnostic information. The results indicate how many total images were processed, and of those processed, how many were accepted, rejected, or skipped. The calibrator skips duplicate images.

To view the rejected images, click **View images**. The calibrator rejects duplicate images. It also rejects images where the entire checkerboard could not be detected. Possible reasons for no detection are a blurry image or an extreme angle of the pattern. Detection takes longer with larger images and with patterns that contain a large number of squares.

View Images and Detected Points

The **Data Browser** pane displays a list of images with IDs. These images contain a detected pattern. To view an image, select it from the **Data Browser** pane.
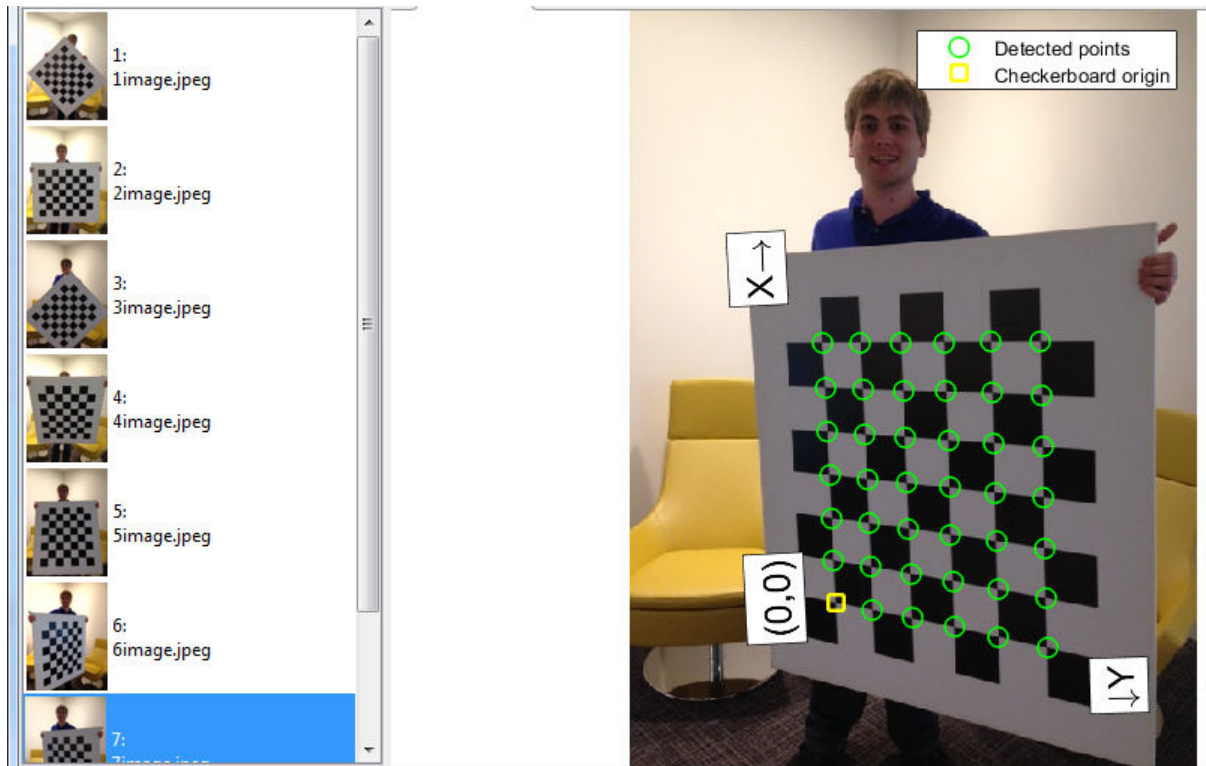
The **Image** window displays the selected checkerboard image with green circles to indicate detected points. You can verify that the corners were detected correctly using the zoom controls. The yellow square indicates the (0,0) origin. The X and Y arrows indicate the checkerboard axes orientation.

Calibrate

Once you are satisfied with the accepted images, click the **Calibrate** button on the **Calibration** tab. The default calibration settings assume the minimum set of camera parameters. Start by running the calibration with the default settings. After evaluating

the results, you can try to improve calibration accuracy by adjusting the settings and adding or removing images and then calibrating again. If you switch between standard and fisheye camera model, you must recalibrate.

Select Camera Model

You can select either a standard or fisheye camera model on the **Calibration** tab, in the **Camera Model** section, select **Standard** or **Fisheye**.

You can switch camera models at any point in the session. You must calibrate again after any changes you make to the app's settings. Click **Options** to access settings and optimizations for either camera model.

Standard Model Options

When the camera has severe lens distortion, the app can fail to compute the initial values for the camera intrinsics. If you have the manufacturer's specifications for your camera and know the pixel size, focal length, or lens characteristics, you can manually set initial guesses for camera intrinsics and radial distortion. To set initial guesses, click **Options > Optimization Options**.

- Select the top checkbox and then enter a 3-by-3 matrix to specify initial intrinsics. If you do not specify an initial guess, the function computes the initial intrinsic matrix using linear least squares.
- Select the bottom checkbox and then enter a 2- or 3-element vector to specify the initial radial distortion. If you do not provide a value, the function uses `0` as the initial value for all the coefficients.

Fisheye Model Options

In the **Camera Model** section, with **Fisheye** selected, click **Options**. Select **Estimate Alignment** to enable estimation of the axes alignment when the optical axis of the fisheye lens is not perpendicular to the image plane.

Calibration Algorithm

See "Fisheye Calibration Basics" on page 6-13 for the fisheye camera model calibration algorithm.

The standard camera model calibration algorithm assumes a pinhole camera model:

$$w[x \quad y \quad 1] = [X \quad Y \quad Z \quad 1]\begin{bmatrix} R \\ t \end{bmatrix} K$$

- (X,Y,Z): world coordinates of a point.
- (x,y): image coordinates of the corresponding image point in pixels.
- w: arbitrary homogeneous coordinates scale factor.
- K: camera intrinsic matrix, defined as.

$$\begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

The coordinates $(c_x\ c_y)$ represent the optical center (the principal point), in pixels. When the x- and y-axes are exactly perpendicular, the skew parameter, s, equals 0. The matrix elements are defined as:
$f_x = F * s_x$
$f_y = F * s_y$
F is the focal length in world units, typically expressed in millimeters.
$[s_x, s_y]$ are the number of pixels per world unit in the x and y direction respectively.
f_x and f_y are expressed in pixels.

- R: matrix representing the 3-D rotation of the camera .
- t: translation of the camera relative to the world coordinate system.

The camera calibration algorithm estimates the values of the intrinsic parameters, the extrinsic parameters, and the distortion coefficients. Camera calibration involves these steps:

1 Solve for the intrinsics and extrinsics in closed form, assuming that lens distortion is zero. [1]

2 Estimate all parameters simultaneously, including the distortion coefficients, using nonlinear least-squares minimization (Levenberg–Marquardt algorithm). Use the closed-form solution from the preceding step as the initial estimate of the intrinsics and extrinsics. Set the initial estimate of the distortion coefficients to zero. [1][2]

Evaluate Calibration Results

You can evaluate calibration accuracy by examining the reprojection errors, examining the camera extrinsics, or viewing the undistorted image. For best calibration results, use all three methods of evaluation.
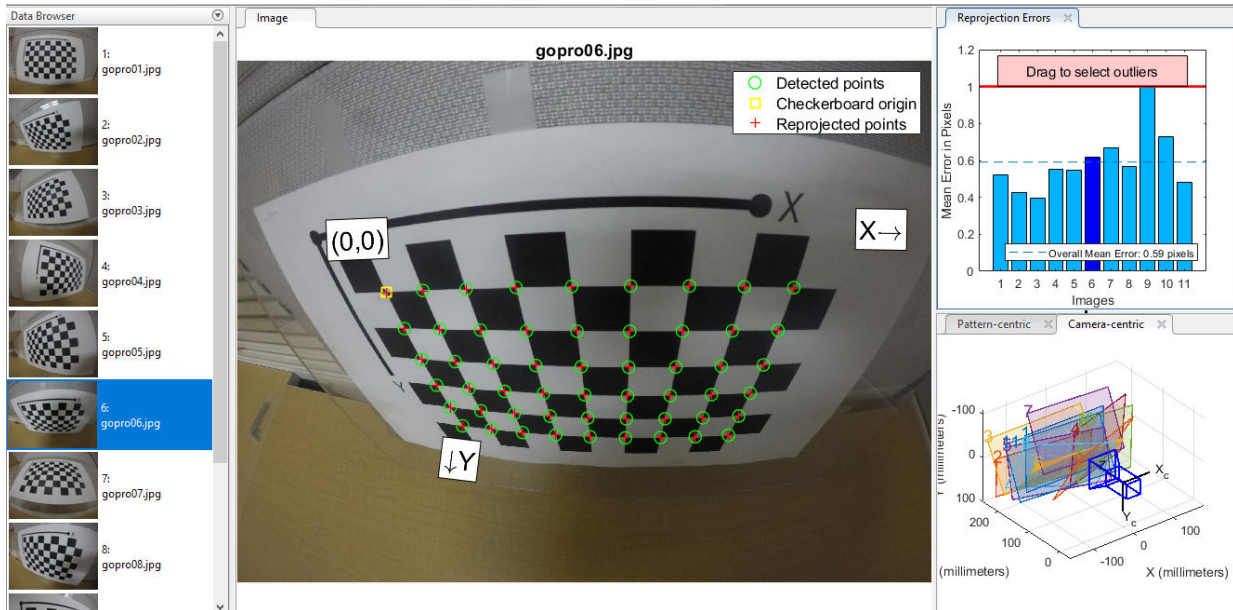
Examine Reprojection Errors

The reprojection errors are the distances, in pixels, between the detected and the reprojected points. The **Camera Calibrator** app calculates reprojection errors by projecting the checkerboard points from world coordinates, defined by the checkerboard, into image coordinates. The app then compares the reprojected points to the corresponding detected points. As a general rule, mean reprojection errors of less than one pixel are acceptable.
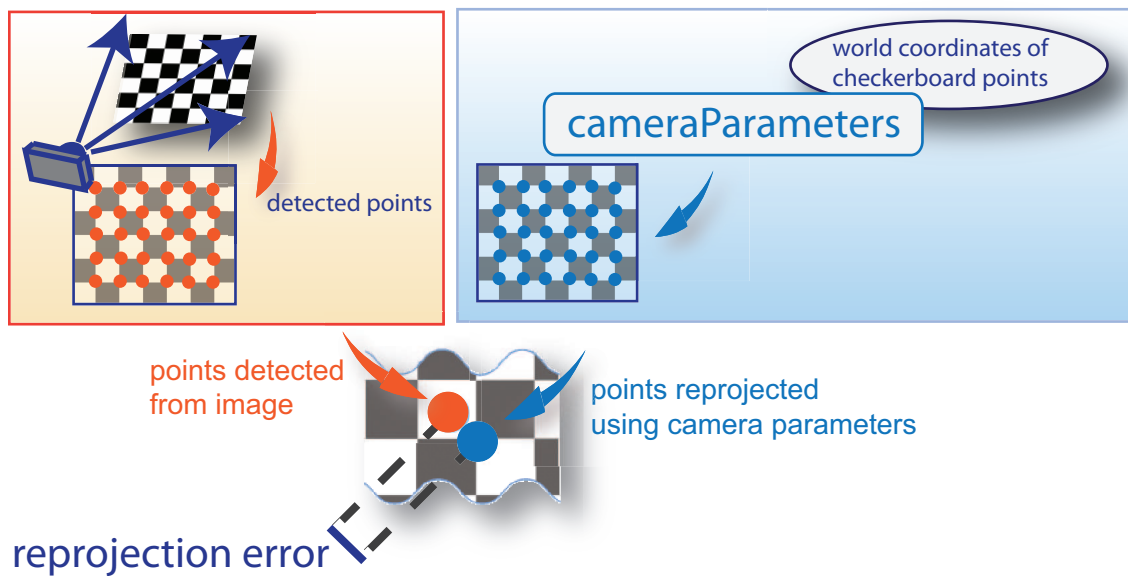
The **Camera Calibrator** app displays, in pixels, the reprojection errors as a bar graph. The graph helps you to identify which images that adversely contribute to the calibration. Select the bar graph entry and remove the image from the list of images in the **Data Browser** pane.

Reprojection Errors Bar Graph
The bar graph displays the mean reprojection error per image, along with the overall mean error. The bar labels correspond to the image IDs. The highlighted bars correspond to the selected images.

Select an image in one of these ways:

- Click a corresponding bar in the graph.
- Select an image from the list of images in the **Data Browser** pane.
- Adjust the overall mean error. Click and slide the red line up or down to select outlier images.

Examine Extrinsic Parameter Visualization

The 3-D extrinsic parameters plot provides a camera-centric view of the patterns and a pattern-centric view of the camera. The camera-centric view is helpful if the camera was stationary when the images were captured. The pattern-centric view is helpful if the pattern was stationary. You can click the cursor and hold down the mouse button with the rotate icon to rotate the figure. Click a checkerboard (or camera) to select it. The highlighted data in the visualizations correspond to the selected image in the list. Examine the relative positions of the pattern and the camera to determine if they match what you expect. For example, a pattern that appears behind the camera indicates a calibration error.
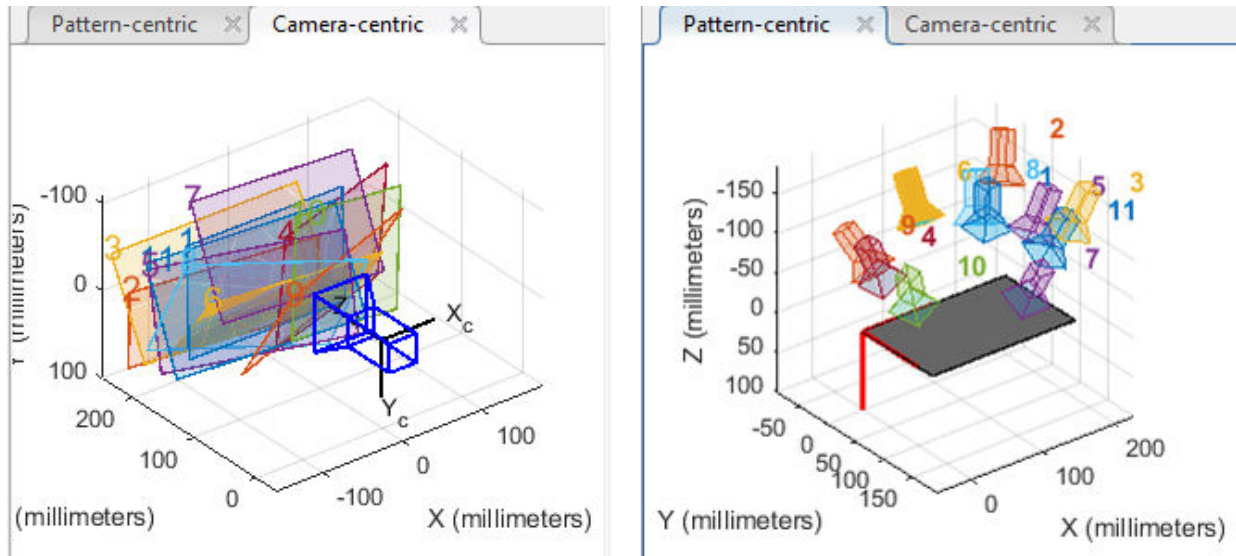
View Undistorted Image

To view the effects of removing lens distortion, click **Show Undistorted** in the **View** section of the **Calibration** tab. If the calibration was accurate, the distorted lines in the image become straight.
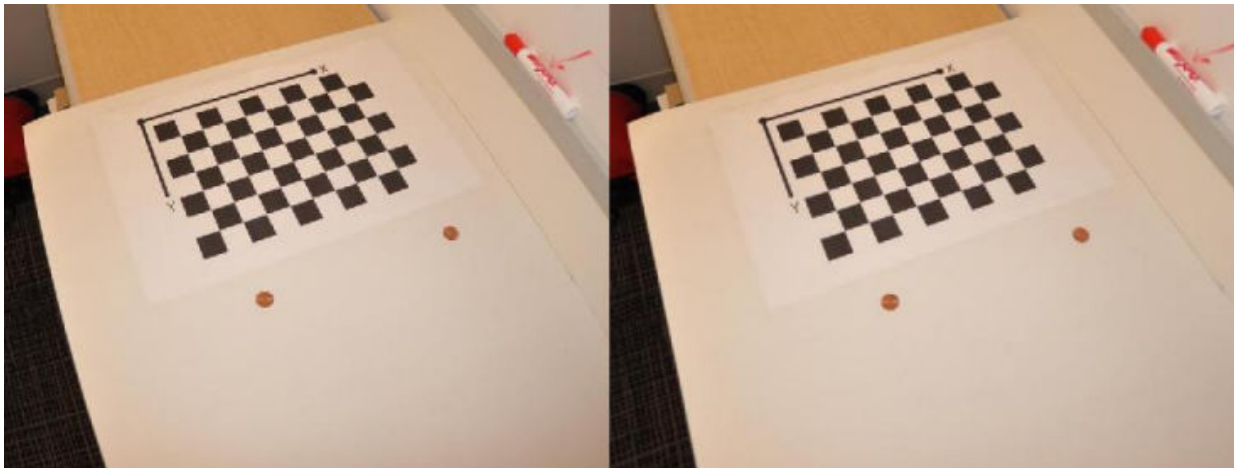
Checking the undistorted images is important even if the reprojection errors are low. For example, if the pattern covers only a small percentage of the image, the distortion estimation might be incorrect, even though the calibration resulted in few reprojection errors. The following image shows an example of this type of incorrect estimation for a single camera calibration.

While viewing the undistorted images, you can examine the fisheye images more closely by selecting **Fisheye Scale** in the **View** section of the **Calibration** tab. Use the slider in the Scale Factor window to adjust the scale of the image.
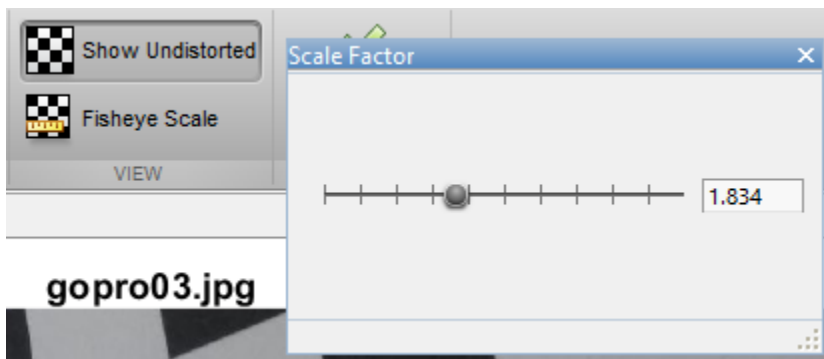
Improve Calibration

To improve the calibration, you can remove high-error images, add more images, or modify the calibrator settings.

Add or Remove Images

Consider adding more images if:

- You have less than 10 images.
- The patterns do not cover enough of the image frame.
- The patterns do not have enough variation in orientation with respect to the camera.

Consider removing images if the images:

- The images have a high mean reprojection error.
- The images are blurry.
- The images contain a checkerboard at an angle greater than 45 degrees relative to the camera plane.
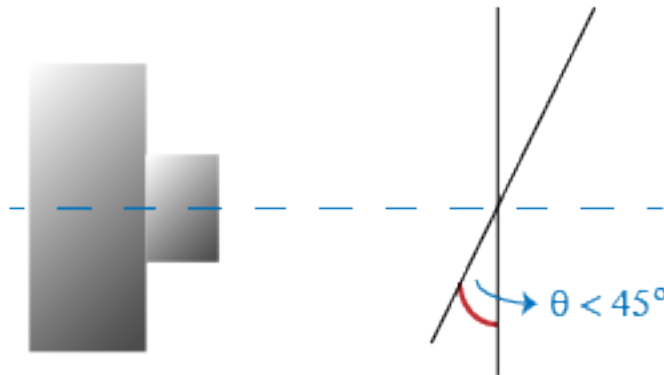
- The images contain incorrectly detected checkerboard points.

Standard Model: Change the Number of Radial Distortion Coefficients

You can specify two or three radial distortion coefficients. On the **Calibrations** tab, in the **Camera Model** section, with **Standard** selected, click **Options**. Select the **Radial Distortion** as either **2 Coefficients** or **3 Coefficients**. Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion.

Negative radial distortion
"pincushion"   No distortion   Positive radial distortion
                                                                   "barrel"

The radial distortion coefficients model this type of distortion. The distorted points are denoted as ($x_{\text{distorted}}$, $y_{\text{distorted}}$):

$$x_{\text{distorted}} = x(1 + k_1*r^2 + k_2*r^4 + k_3*r^6)$$

$$y_{\text{distorted}} = y(1 + k_1*r^2 + k_2*r^4 + k_3*r^6)$$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- k_1, k_2, and k_3 — Radial distortion coefficients of the lens.
- r^2: $x^2 + y^2$

Typically, two coefficients are sufficient for calibration. For severe distortion, such as in wide-angle lenses, you can select 3 coefficients to include k_3.

The undistorted pixel locations are in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

Standard Model: Compute Skew

When you select the **Compute Skew** check box, the calibrator estimates the image axes skew. Some camera sensors contain imperfections that cause the x- and y-axes of the image to not be perpendicular. You can model this defect using a skew parameter. If you do not select the check box, the image axes are assumed to be perpendicular, which is the case for most modern cameras.

Standard Model: Compute Tangential Distortion

Tangential distortion occurs when the lens and the image plane are not parallel. The tangential distortion coefficients model this type of distortion.

The distorted points are denoted as ($x_{\text{distorted}}$, $y_{\text{distorted}}$):

$$x_{\text{distorted}} = x + [2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2)]$$

$$y_{\text{distorted}} = y + [p_1 * (r^2 + 2 *y^2) + 2 * p_2 * x * y]$$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- p_1 and p_2 — Tangential distortion coefficients of the lens.
- r^2: $x^2 + y^2$

When you select the **Compute Tangential Distortion** check box, the calibrator estimates the tangential distortion coefficients. Otherwise, the calibrator sets the tangential distortion coefficients to zero.

Fisheye Model: Estimate Alignment

In the **Camera Model** section, with **Fisheye** selected, click **Options**. Select **Estimate Alignment** to enable estimation of the axes alignment when the optical axis of the fisheye lens is not perpendicular to the image plane.

Export Camera Parameters

When you are satisfied with calibration accuracy, click **Export Camera Parameters**. You can either save and export the camera parameters to an object by selecting **Export Camera Parameters** or generate the camera parameters as a MATLAB script.

Export Camera Parameters

Select **Export Camera Parameters > Export Parameters to Workspace** to create a `cameraParameters` object in your workspace. The object contains the intrinsic and extrinsic parameters of the camera and the distortion coefficients. You can use this object for various computer vision tasks, such as image undistortion, measuring planar objects, and 3-D reconstruction. See "Measuring Planar Objects with a Calibrated Camera". You can optionally export the `cameraCalibrationErrors` object, which contains the standard errors of estimated camera parameters, by selecting the **Export estimation errors** check box.

Generate MATLAB Script

Select **Export Camera Parameters > Generate MATLAB script** to save your camera parameters to a MATLAB script, enabling you to reproduce the steps from your calibration session.

References

[1] Zhang, Z. "A Flexible New Technique for Camera Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, Number. 11, 2000, pp. 1330–1334.

[2] Heikkila, J. and O. Silven. "A Four-step Camera Calibration Procedure with Implicit Image Correction." *IEEE International Conference on Computer Vision and Pattern Recognition*. 1997.

[3] Scaramuzza, D., A. Martinelli, and R. Siegwart. "A Toolbox for Easy Calibrating Omindirectional Cameras." *Proceedings to IEEE International Conference on Intelligent Robots and Systems (IROS 2006)*. Beijing, China, October 7–15, 2006.

[4] Urban, S., J. Leitloff, and S. Hinz. "Improved Wide-Angle, Fisheye and Omnidirecitonal Camera Calibration." *ISPRS Journal of Photogrammetry and Remove Sensing*. Vol. 108, 2015, pp.72–79.

See Also

Camera Calibrator | **Stereo Camera Calibrator** | cameraParameters | detectCheckerboardPoints | estimateCameraParameters | generateCheckerboardPoints | showExtrinsics | showReprojectionErrors | stereoParameters | undistortImage

Related Examples

- "Evaluating the Accuracy of Single Camera Calibration"
- "Measuring Planar Objects with a Calibrated Camera"
- "Structure From Motion From Two Views"
- "Structure From Motion From Multiple Views"
- "Depth Estimation From Stereo Video"
- "3-D Point Cloud Registration and Stitching"
- "Uncalibrated Stereo Image Rectification"
- Checkerboard pattern

More About

- "Stereo Camera Calibrator App" on page 6-43
- "Coordinate Systems"

External Websites

- Camera Calibration with MATLAB

Stereo Camera Calibrator App

Stereo Camera Calibrator Overview

You can use the **Stereo Camera Calibrator** app to calibrate a stereo camera, which you can then use to recover depth from images. A stereo system consists of two cameras: camera 1 and camera 2. The app can either estimate or import the parameters of individual cameras. The app also calculates the position and orientation of camera 2, relative to camera 1.

The **Stereo Camera Calibrator** app produces an object containing the stereo camera parameters. You can use this object to

- Rectify stereo images using the `rectifyStereoImages` function.
- Reconstruct the 3-*D* scene using the `reconstructScene` function.
- Compute 3-D locations corresponding to matching pairs of image points using the `triangulate` function.

The suite of calibration functions used by the **Stereo Camera Calibrator** app provide the workflow for stereo system calibration. You can these functions directly in the MATLAB workspace. For a list of calibration functions, see "Single Camera Calibration".

Note You can use the Camera Calibrator app with cameras up to a field of view (FOV) of 95 degrees.

Stereo Camera Calibration
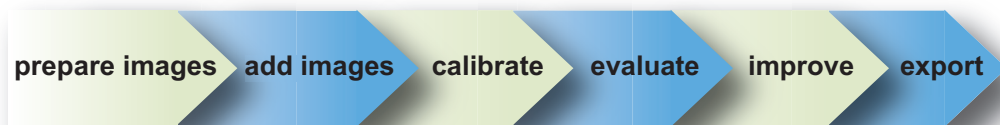
Follow this workflow to calibrate your stereo camera using the app:

1 Prepare images, camera, and calibration pattern.

2 Add image pairs.

3 Calibrate the stereo camera.

4 Evaluate calibration accuracy.

5 Adjust parameters to improve accuracy (if necessary).

6 Export the parameters object.

7 In some cases, the default values work well, and you do not need to make any improvements before exporting parameters. You can also make improvements using the camera calibration functions directly in the MATLAB workspace. For a list of functions, see "Single Camera Calibration".

Open the Stereo Camera Calibrator

- MATLAB Toolstrip: On the **Apps** tab, in the **Image Processing and Computer Vision** section, click the **Stereo Camera Calibrator** icon.

- MATLAB command prompt: Enter `stereoCameraCalibrator`

Prepare Pattern, Camera, and Images

To better the results, use between 10 and 20 images of the calibration pattern. The calibrator requires at least three images. Use uncompressed images or lossless compression formats such as PNG. The calibration pattern and the camera setup must satisfy a set of requirements to work with the calibrator. For greater calibration accuracy, follow these instructions for preparing the pattern, setting up the camera, and capturing the images.

Prepare the Checkerboard Pattern

The **Camera Calibrator** app uses a checkerboard pattern. A checkerboard pattern is a convenient calibration target. If you want to use a different pattern to extract key points, you can use the camera calibration MATLAB functions directly. See "Single Camera Calibration" for the list of functions.

You can print (from MATLAB) and use the checkerboard pattern provided. The checkerboard pattern you use must not be square. One side must contain an even number of squares and the other side must contain an odd number of squares. Therefore, the pattern contains two black corners along one side and two white corners on the opposite side. This criteria enables the app to determine the orientation of the pattern. The calibrator assigns the longer side to be the *x*-direction.

To prepare the checkerboard pattern:

1 Attach the checkerboard printout to a flat surface. Imperfections on the surface can affect the accuracy of the calibration.

2 Measure one side of the checkerboard square. You need this measurement for calibration. The size of the squares can vary depending on printer settings.

Size of checkerboard square

3 To improve the detection speed, set up the pattern with as little background clutter as possible.

Camera Setup

To calibrate your camera, follow these rules:

- Keep the pattern in focus, but do not use autofocus.
- If you change zoom settings between images, the focal length changes.

Capture Images

For best results, use at least 10 to 20 images of the calibration pattern. The calibrator requires at least three images. Use uncompressed images or images in lossless compression formats such as PNG. For greater calibration accuracy:

- Capture the images of the pattern at a distance roughly equal to the distance from your camera to the objects of interest. For example, if you plan to measure objects from 2 meters, keep your pattern approximately 2 meters from the camera.
- Place the checkerboard at an angle less than 45 degrees relative to the camera plane.

- Do not modify the images, (for example, do not crop them).
- Do not use autofocus or change the zoom settings between images.
- Capture the images of a checkerboard pattern at different orientations relative to the camera.
- Capture a variety of images of the pattern so that you have accounted for as much of the image frame as possible. Lens distortion increases radially from the center of the image and sometimes is not uniform across the image frame. To capture this lens distortion, the pattern must appear close to the edges of the captured images.

- Make sure the checkerboard pattern is fully visible in both images of each stereo pair.
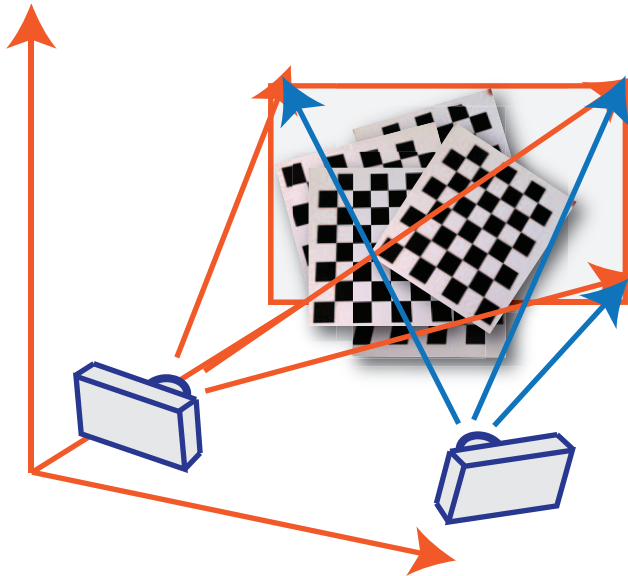
- Keep the pattern stationary for each image pair. Any motion of the pattern between taking image 1 and image 2 of the pair negatively affects the calibration.

- Create a stereo display, or anaglyph, by positioning the two cameras approximately 55 mm apart. This distance represents the average distance between human eyes.

- For greater reconstruction accuracy at longer distances, position your cameras farther apart.

Add Image Pairs

To begin calibration, click Add images, specifically two sets of stereo images of the checkerboard,one set from each camera.

Load Images

You can add images from multiple folders by clicking Add images in the **File** section of the **Calibration** tab. Select the location for the images corresponding to camera 1 using the **Browse** button, then do the same for camera 2. **Specify Size of checkerboard square** by entering the length of one side of a square from the checkerboard pattern.

Size of checkerboard square
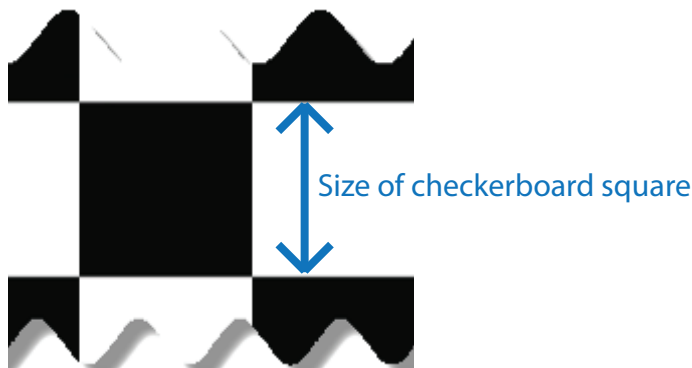
Analyze Images

The calibrator attempts to detect a checkerboard in each of the added images, displaying an Analyzing Images progress bar window, indicating detection progress. If any of the images are rejected, the Detection Results dialog box appears, which contains diagnostic information. The results indicate how many total images were processed, and of those processed, how many were accepted, rejected, or skipped. The calibrator skips duplicate images.

To view the rejected images, click **View images**. The calibrator rejects duplicate images. It also rejects images where the entire checkerboard could not be detected. Possible reasons for no detection are a blurry image or an extreme angle of the pattern. Detection takes longer with larger images and with patterns that contain a large number of squares.

View Images and Detected Points

The **Data Browser** pane displays a list of image pairs with IDs. These image pairs contain a detected pattern. To view an image, select it from the **Data Browser** pane.
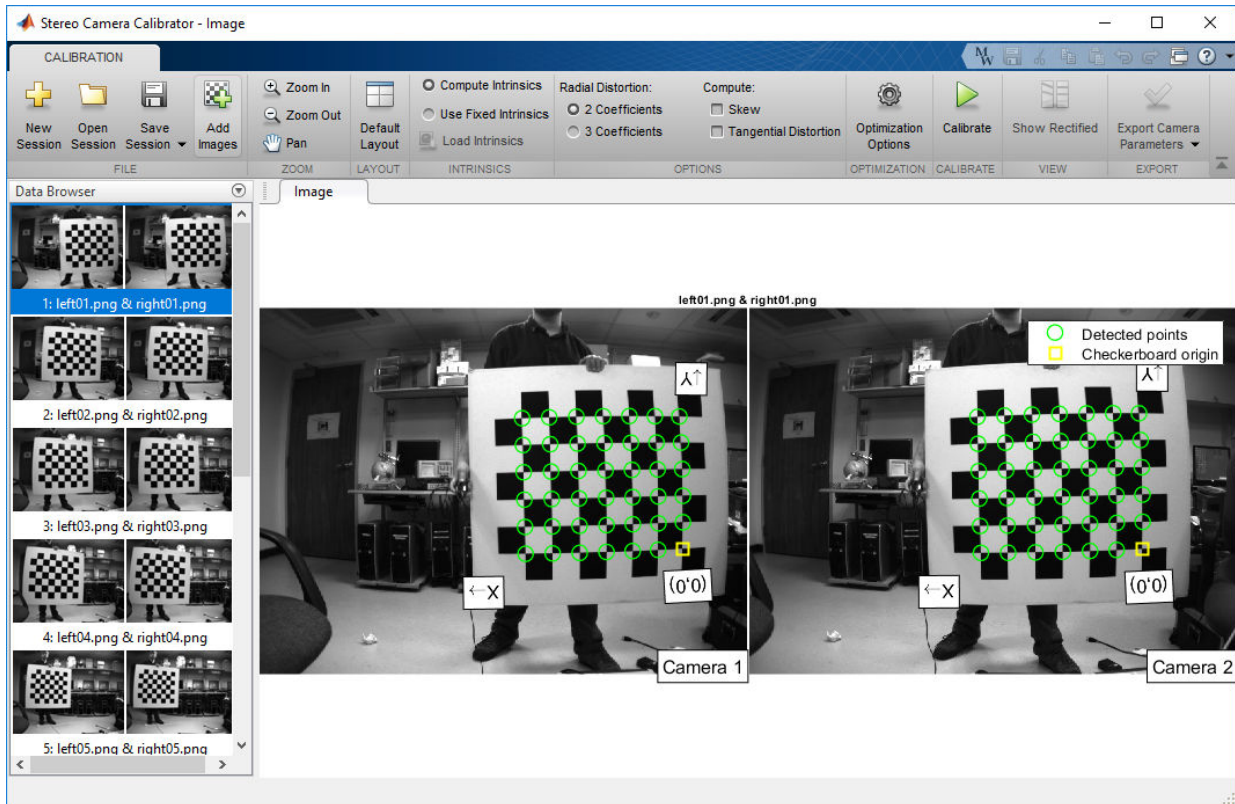
The **Image** pane displays the selected checkerboard image pair with green circles to indicate detected points. You can verify that the corners were detected correctly using the zoom controls. The yellow square indicates the (0,0) origin. The X and Y arrows indicate the checkerboard axes orientation.

Intrinsics

You can choose for the app to compute camera intrinsics or you can load pre-computed fixed intrinsics. To load intrinsics into the app, select Use Fixed Intrinsics in the Intrinsics section of the **Calibration** tab. The **Radial Distortion** and **Compute** options in the **Options** section are disabled when you load intrinsics.

To load intrinsics as variables from your workspace, click **Load Intrinsics**. For example, if the `wideBaselineStereo` struct contains the intrinsics for both cameras.

```
ld = load('wideBaselineStereo');
int1 = ld.intrinsics1
int2 = ld.intrinsics2
```

Then, click **Load Intrinsics** to specify these variables in the dialog box, as shown.
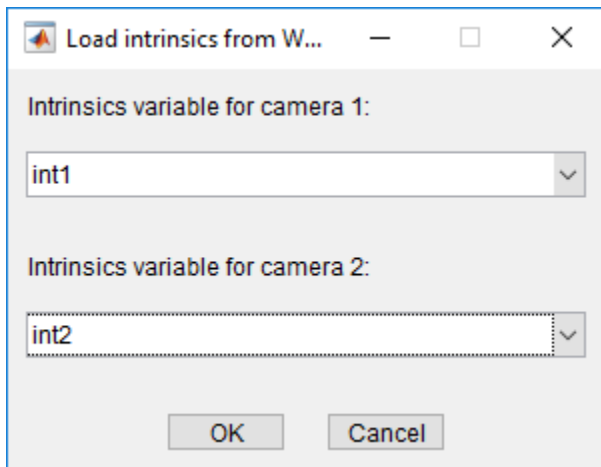
Calibrate

Once you are satisfied with the accepted image pairs, click the **Calibrate** button on the **Calibration** tab. The default calibration settings assume the minimum set of camera parameters. Start by running the calibration with the default settings. After evaluating the results, you can try to improve calibration accuracy by adjusting the settings and adding or removing images, and then calibrate again.

Optimization

When the camera has severe lens distortion, the app can fail to compute the initial values for the camera intrinsics. If you have the manufacturer's specifications for your camera and know the pixel size, focal length, or lens characteristics, you can manually set initial guesses for camera intrinsics and radial distortion. To set initial guesses, click **Options > Optimization Options**.

Note These options are not available for preloaded intrinsics.

- Select the top checkbox and then enter a 3-by-3 matrix to specify initial intrinsics. If you do not specify an initial guess, the function computes the initial intrinsic matrix using linear least squares.

- Select the bottom checkbox and then enter a 2- or 3-element vector to specify the initial radial distortion. If you do not provide a value, the function uses 0 as the initial value for all the coefficients.

Evaluate Calibration Results

You can evaluate calibration accuracy by examining the reprojection errors, examining the camera extrinsics, or viewing the undistorted image. For best calibration results, use all three methods of evaluation.

Examine Reprojection Errors

The reprojection errors are the distances, in pixels, between the detected and the reprojected points. The **Stereo Camera Calibrator** app calculates reprojection errors by projecting the checkerboard points from world coordinates, defined by the checkerboard, into image coordinates. The app then compares the reprojected points to the corresponding detected points. As a general rule, mean reprojection errors of less than one pixel are acceptable.
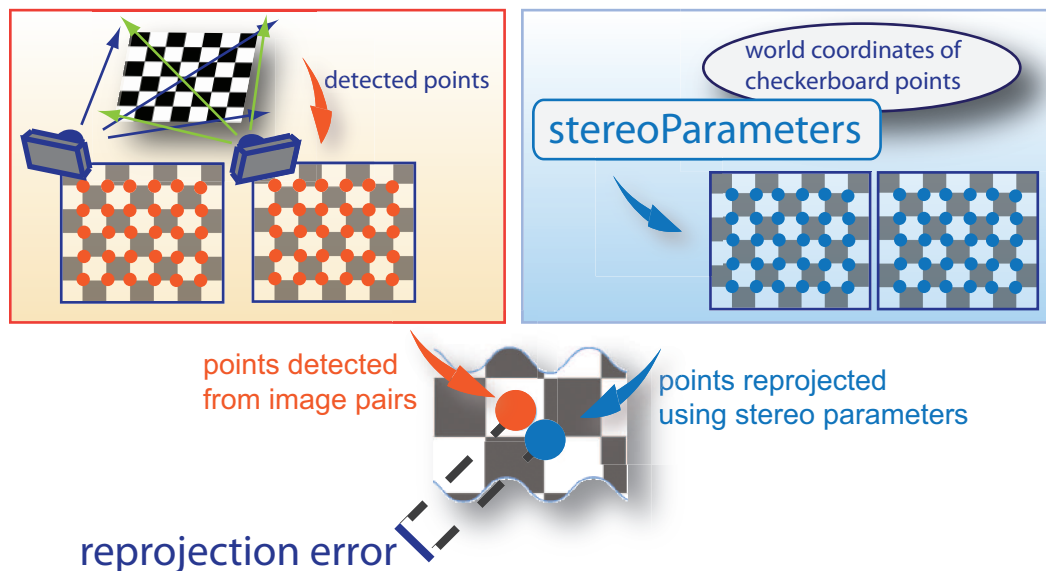
The **Stereo Calibration App** app displays, in pixels, the reprojection errors as a bar graph. The graph helps you to identify which images that adversely contribute to the calibration. Select the bar graph entry and remove the image from the list of images in the **Data Browser** pane.

Reprojection Errors Bar Graph
The bar graph displays the mean reprojection error per image, along with the overall mean error. The bar labels correspond to the image IDs. The highlighted bars correspond to the selected image pair.
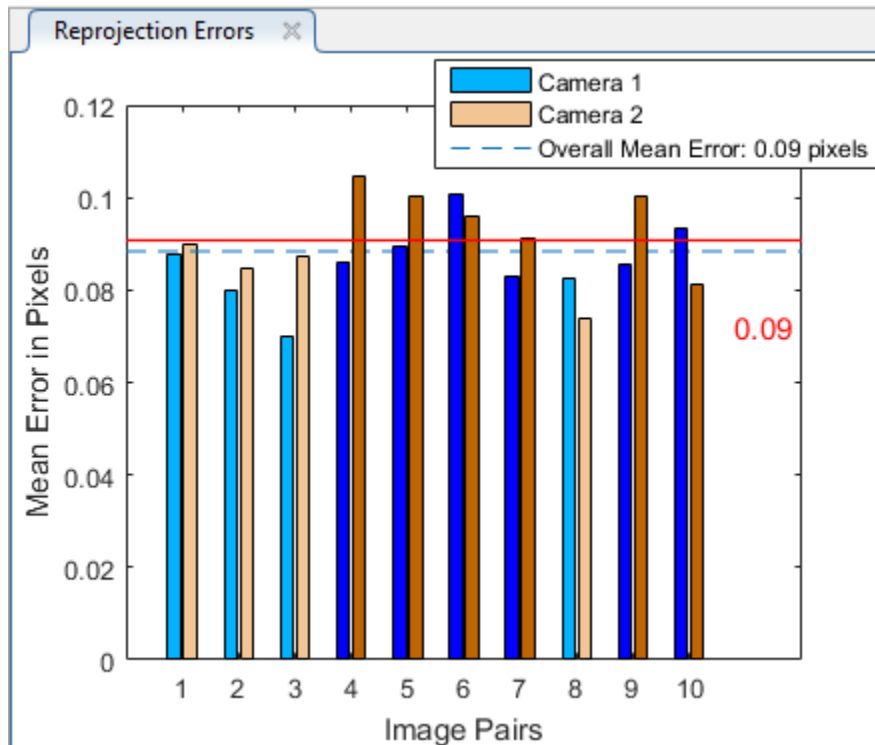
Select an image pair in one of these ways:

- Clicking the corresponding bar in the graph.
- Select the image pair from the list in the **Data Browser** pane.
- Adjust the overall mean error. Click and slide the red line up or down to select outlier images.

Examine Extrinsic Parameter Visualization

The 3-D extrinsic parameters plot provides a camera-centric view of the patterns and a pattern-centric view of the camera. The camera-centric view is helpful if the camera was stationary when the images were captured. The pattern-centric view is helpful if the pattern was stationary. You can click the cursor and hold down the mouse button with the rotate icon to rotate the figure. Click a checkerboard (or camera) to select it. The highlighted data in the visualizations correspond to the selected image in the list. Examine the relative positions of the pattern and the camera to determine if they match

6-55

what you expect. For example, a pattern that appears behind the camera indicates a calibration error.

Show Rectified Images

To view the effects of stereo rectification, click **Show Rectified** in the **View** section of the **Calibration** tab. If the calibration was accurate, the images become undistorted and row-aligned.
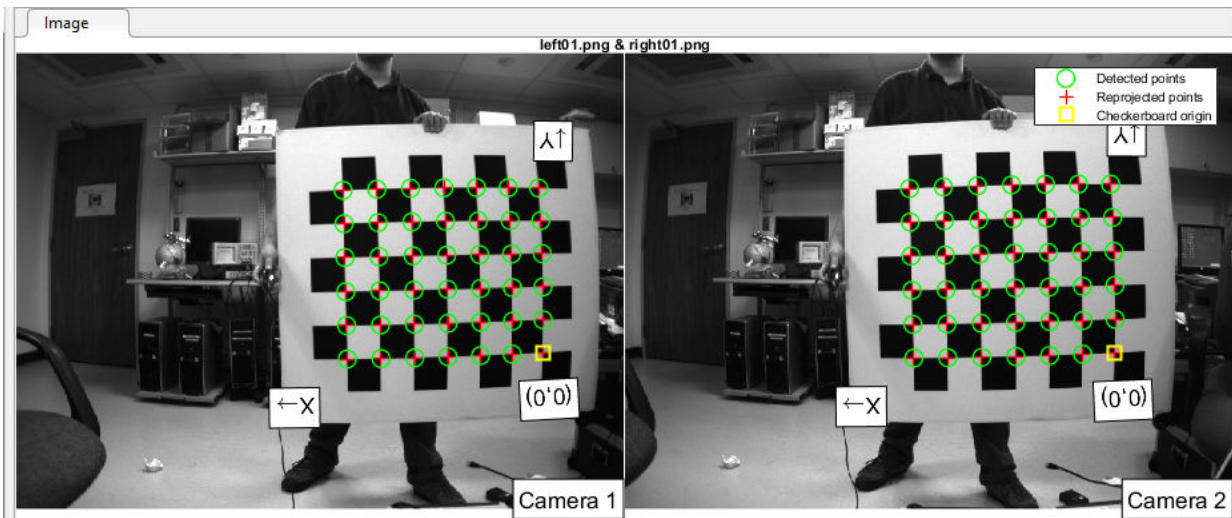
Check the rectified images is important even if the reprojection errors are low. For example, if the pattern covers only a small percentage of the image, the distortion estimation might be incorrect, even though the calibration resulted in few reprojection errors.The following image shows an example of this type of incorrect estimation for a single camera calibration.

Improve Calibration

To improve the calibration, you can remove high-error image pairs, add more image pairs, or modify the calibrator settings.

Add or Remove Images

Consider adding more images if:

- You have less than 10 images.
- The patterns do not cover enough of the image frame.
- The patterns do not have enough variation in orientation with respect to the camera.

Consider removing images if the images:

- The images have a high mean reprojection error.
- The images are blurry.
- The images contain a checkerboard at an angle greater than 45 degrees relative to the camera plane.
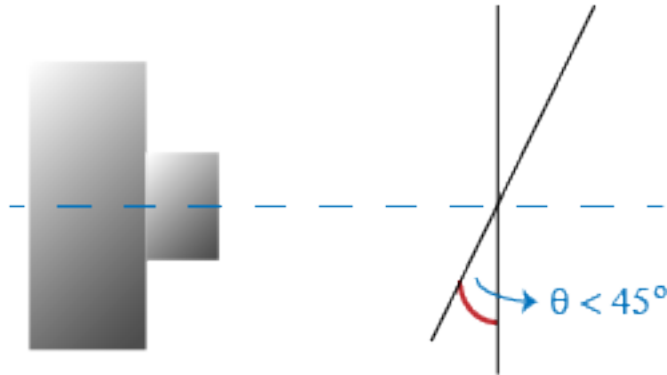
• The images contain incorrectly detected checkerboard points.

Change the Number of Radial Distortion Coefficients

You can specify 2 or 3 radial distortion coefficients by selecting the corresponding radio button from the **Options** section. Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion.

Negative radial distortion   No distortion   Positive radial distortion
"pincushion"   "barrel"

The radial distortion coefficients model this type of distortion. The distorted points are denoted as $(x_{\text{distorted}}, y_{\text{distorted}})$:

$x_{\text{distorted}} = x(1 + k_1 \ast r^2 + k_2 \ast r^4 + k_3 \ast r^6)$

$y_{\text{distorted}} = y(1 + k_1 \ast r^2 + k_2 \ast r^4 + k_3 \ast r^6)$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- k_1, k_2, and k_3 — Radial distortion coefficients of the lens.
- r^2: $x^2 + y^2$

Typically, two coefficients are sufficient for calibration. For severe distortion, such as in wide-angle lenses, you can select 3 coefficients to include k_3.

Compute Skew

When you select the **Compute Skew** check box, the calibrator estimates the image axes skew. Some camera sensors contain imperfections that cause the x- and y-axes of the image to not be perpendicular. You can model this defect using a skew parameter. If you do not select the check box, the image axes are assumed to be perpendicular, which is the case for most modern cameras.

Compute Tangential Distortion

Tangential distortion occurs when the lens and the image plane are not parallel. The tangential distortion coefficients model this type of distortion.

The distorted points are denoted as ($x_{\text{distorted}}$, $y_{\text{distorted}}$):

$$x_{\text{distorted}} = x + [2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2)]$$

$$y_{\text{distorted}} = y + [p_1 * (r^2 + 2 *y^2) + 2 * p_2 * x * y]$$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- p_1 and p_2 — Tangential distortion coefficients of the lens.
- r^2: $x^2 + y^2$

When you select the **Compute Tangential Distortion** check box, the calibrator estimates the tangential distortion coefficients. Otherwise, the calibrator sets the tangential distortion coefficients to zero.

Export Camera Parameters

When you are satisfied with calibration accuracy, click **Export Camera Parameters**. You can either save and export the camera parameters to an object by selecting **Export Camera Parameters** or generate the camera parameters as a MATLAB script.

Export Camera Parameters

Select **Export Camera Parameters > Export Parameters to Workspace** to create a `stereoParameters` object in your workspace. The object contains the intrinsic and extrinsic parameters of the camera and the distortion coefficients. You can use this object for various computer vision tasks, such as image undistortion, measuring planar objects, and 3-D reconstruction. See "Measuring Planar Objects with a Calibrated Camera". You can optionally export the `stereoCalibrationErrors` object, which contains the standard errors of estimated stereo camera parameters, by selecting the **Export estimation errors** check box.

Generate MATLAB Script

Select **Export Camera Parameters > Generate MATLAB script** to save your camera parameters to a MATLAB script, enabling you to reproduce the steps from your calibration session.

References

[1] Zhang, Z. "A Flexible New Technique for Camera Calibration". *IEEE Transactions on Pattern Analysis and Machine Intelligence*.Vol. 22, No. 11, 2000, pp. 1330–1334.

[2] Heikkila, J, and O. Silven. "A Four-step Camera Calibration Procedure with Implicit Image Correction." *IEEE International Conference on Computer Vision and Pattern Recognition*. 1997.

See Also
Camera Calibrator | **Stereo Camera Calibrator** | cameraParameters | detectCheckerboardPoints | estimateCameraParameters | generateCheckerboardPoints | showExtrinsics | showReprojectionErrors | stereoParameters | undistortImage

Related Examples
- "Evaluating the Accuracy of Single Camera Calibration"
- "Measuring Planar Objects with a Calibrated Camera"
- "Structure From Motion From Two Views"
- "Structure From Motion From Multiple Views"
- "Depth Estimation From Stereo Video"
- "3-D Point Cloud Registration and Stitching"
- "Uncalibrated Stereo Image Rectification"
- Checkerboard pattern

More About
- "Single Camera Calibrator App" on page 6-21
- "Coordinate Systems"

External Websites
- Camera Calibration with MATLAB

What Is Camera Calibration?

Geometric camera calibration, also referred to as camera resectioning, estimates the parameters of a lens and image sensor of an image or video camera. You can use these parameters to correct for lens distortion, measure the size of an object in world units, or determine the location of the camera in the scene. These tasks are used in applications such as machine vision to detect and measure objects. They are also used in robotics, for navigation systems, and 3-D scene reconstruction.

Examples of what you can do after calibrating your camera:

Before

After

Remove Lens Distortion
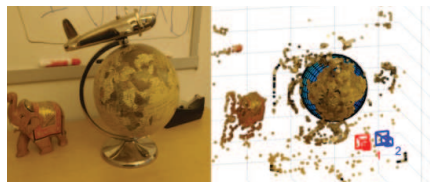
Estimate 3-D Structure from Camera Motion
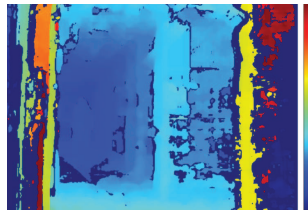
Estimate Depth Using a Stereo Camera

Measure Planar Objects

Camera parameters include intrinsics, extrinsics, and distortion coefficients. To estimate the camera parameters, you need to have 3-D world points and their corresponding 2-D image points. You can get these correspondences using multiple images of a calibration pattern, such as a checkerboard. Using the correspondences, you can solve for the camera parameters. After you calibrate a camera, to evaluate the accuracy of the estimated parameters, you can:

- Plot the relative locations of the camera and the calibration pattern
- Calculate the reprojection errors.
- Calculate the parameter estimation errors.

Use the **Camera Calibrator** to perform camera calibration and evaluate the accuracy of the estimated parameters.

Camera Model

The Computer Vision System Toolbox calibration algorithm uses the camera model proposed by Jean-Yves Bouguet [3]. The model includes:

- The pinhole camera model [1].
- Lens distortion [2].

The pinhole camera model does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent a real camera, the full camera model used by the algorithm includes the radial and tangential lens distortion.

Pinhole Camera Model

A pinhole camera is a simple camera without a lens and with a single small aperture. Light rays pass through the aperture and project an inverted image on the opposite side of the camera. Think of the virtual image plane as being in front of the camera and containing the upright image of the scene.
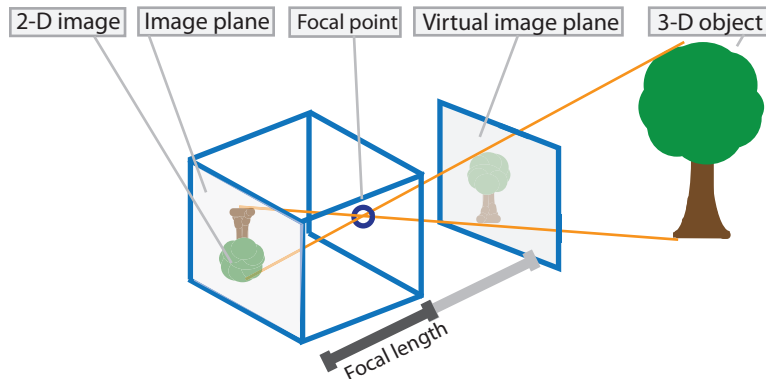
The pinhole camera parameters are represented in a 4-by-3 matrix called the camera matrix. This matrix maps the 3-D world scene into the image plane. The calibration algorithm calculates the camera matrix using the extrinsic and intrinsic parameters. The extrinsic parameters represent the location of the camera in the 3-D scene. The intrinsic parameters represent the optical center and focal length of the camera.

$$w \, [x \ y \ 1] = [X \ Y \ Z \ 1] \, P$$

Scale factor *Image points* *World points*

$$P = \left[\begin{array}{c} R \\ t \end{array} \right] K$$

Camera matrix *Intrinsic matrix*
Extrinsics
Rotation and translation

The world points are transformed to camera coordinates using the extrinsics parameters. The camera coordinates are mapped into the image plane using the intrinsics parameters.
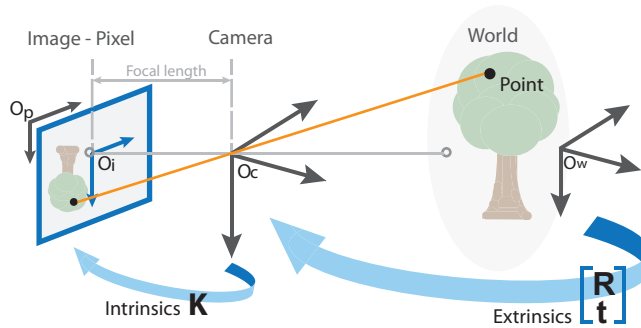
Camera Calibration Parameters

The calibration algorithm calculates the camera matrix using the extrinsic and intrinsic parameters. The extrinsic parameters represent a rigid transformation from 3-D world coordinate system to the 3-D camera's coordinate system. The intrinsic parameters represent a projective transformation from the 3-D camera's coordinates into the 2-D image coordinates.
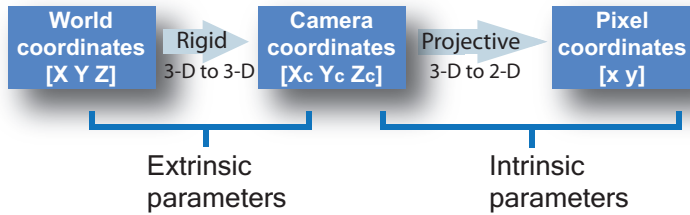
Extrinsic Parameters

The extrinsic parameters consist of a rotation, *R*, and a translation, *t*. The origin of the camera's coordinate system is at its optical center and its *x*- and *y*-axis define the image plane.
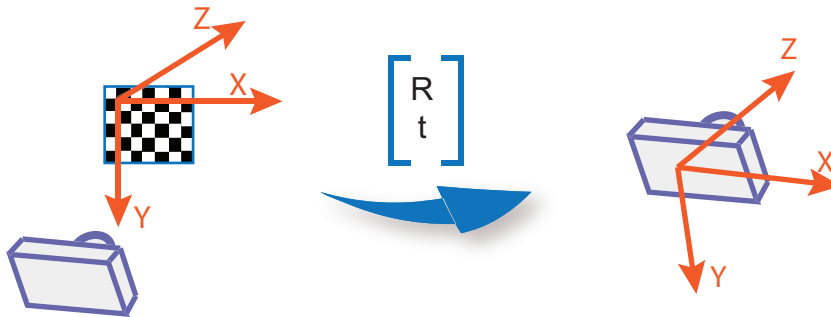
Intrinsic Parameters

The intrinsic parameters include the focal length, the optical center, also known as the principal point, and the skew coefficient. The camera intrinsic matrix, *K*, is defined as:

$$\begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

The pixel skew is defined as:

Skew

$\begin{bmatrix} c_x & c_y \end{bmatrix}$ — Optical center (the principal point), in pixels.
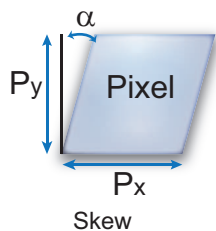
$\begin{pmatrix} f_x, & f_y \end{pmatrix}$ — Focal length in pixels.

$f_x = F / p_x$

$f_y = F / p_y$

F — Focal length in world units, typically expressed in millimeters.

$\begin{pmatrix} p_x, & p_y \end{pmatrix}$ — Size of the pixel in world units.

s — Skew coefficient, which is non-zero if the image axes are not perpendicular.

$s = f_y \tan \alpha$

Distortion in Camera Calibration

The camera matrix does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent a real camera, the camera model includes the radial and tangential lens distortion.

Radial Distortion

Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion.

Negative radial distortion "pincushion"   No distortion   Positive radial distortion "barrel"

The radial distortion coefficients model this type of distortion. The distorted points are denoted as ($x_{distorted}$, $y_{distorted}$):

$x_{distorted} = x(1 + k_1{*}r^2 + k_2{*}r^4 + k_3{*}r^6)$

$y_{distorted} = y(1 + k_1{*}r^2 + k_2{*}r^4 + k_3{*}r^6)$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- k_1, k_2, and k_3 — Radial distortion coefficients of the lens.
- r^2: $x^2 + y^2$

Typically, two coefficients are sufficient for calibration. For severe distortion, such as in wide-angle lenses, you can select 3 coefficients to include k_3.

Tangential Distortion

Tangential distortion occurs when the lens and the image plane are not parallel. The tangential distortion coefficients model this type of distortion.

The distorted points are denoted as ($x_{\text{distorted}}$, $y_{\text{distorted}}$):

$x_{\text{distorted}} = x + [2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2)]$

$y_{\text{distorted}} = y + [p_1 * (r^2 + 2 * y^2) + 2 * p_2 * x * y]$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- p_1 and p_2 — Tangential distortion coefficients of the lens.
- r^2: $x^2 + y^2$

References

[1] Zhang, Z. "A Flexible New Technique for Camera Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.

[2] Heikkila, J., and O. Silven. "A Four-step Camera Calibration Procedure with Implicit Image Correction." *IEEE International Conference on Computer Vision and Pattern Recognition*.1997.

[3] Bouguet, J. Y. "Camera Calibration Toolbox for Matlab." Computational Vision at the California Institute of Technology. Camera Calibration Toolbox for MATLAB.

[4] Bradski, G., and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol, CA: O'Reilly, 2008.

See Also

"Single Camera Calibrator App" on page 6-21 | **Camera Calibrator**

Related Examples

- "Evaluating the Accuracy of Single Camera Calibration"
- "Measuring Planar Objects with a Calibrated Camera"
- "Structure From Motion From Two Views"

Structure from Motion

In this section...
"Structure from Motion from Two Views" on page 6-70
"Structure from Motion from Multiple Views" on page 6-72

Structure from motion (SfM) is the process of estimating the 3-D structure of a scene from a set of 2-D images. SfM is used in many applications, such as 3-D scanning and augmented reality.

SfM can be computed in many different ways. The way in which you approach the problem depends on different factors, such as the number and type of cameras used, and whether the images are ordered. If the images are taken with a single calibrated camera, then the 3-D structure and camera motion can only be recovered up to scale. up to scale means that you can rescale the structure and the magnitude of the camera motion and still maintain observations. For example, if you put a camera close to an object, you can see the same image as when you enlarge the object and move the camera far away. If you want to compute the actual scale of the structure and motion in world units, you need additional information, such as:

- The size of an object in the scene
- Information from another sensor, for example, an odometer.

Structure from Motion from Two Views

For the simple case of structure from two stationary cameras or one moving camera, one view must be considered camera 1 and the other one camera 2. In this scenario, the algorithm assumes that camera 1 is at the origin and its optical axis lies along the z-axis.
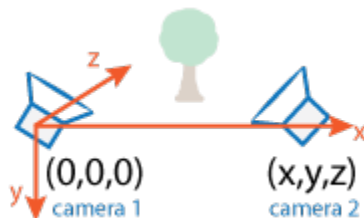
1　SfM requires point correspondences between images. Find corresponding points either by matching features or tracking points from image 1 to image 2. Feature

tracking techniques, such as Kanade-Lucas-Tomasi (KLT) algorithm, work well when the cameras are close together. As cameras move further apart, the KLT algorithm breaks down, and feature matching can be used instead.

camera 1   camera 2
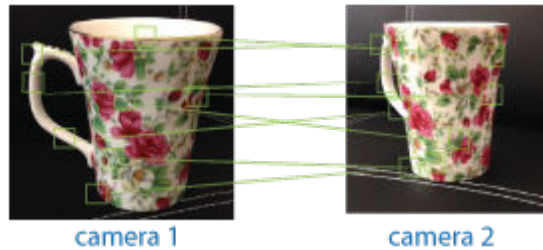
Distance Between Cameras (Baseline)	Method for Finding Point Correspondences	Example
Wide	Match features using `matchFeatures`	"Find Image Rotation and Scale Using Automated Feature Matching"
Narrow	Track features using `vision.PointTracker`	"Face Detection and Tracking Using the KLT Algorithm"

2 To find the pose of the second camera relative to the first camera, you must compute the fundamental matrix. Use the corresponding points found in the previous step for the computation. The fundamental matrix describes the epipolar geometry of the two cameras. It relates a point in one camera to an epipolar line in the other camera. Use the `estimateFundamentalMatrix` function to estimate the fundamental matrix.
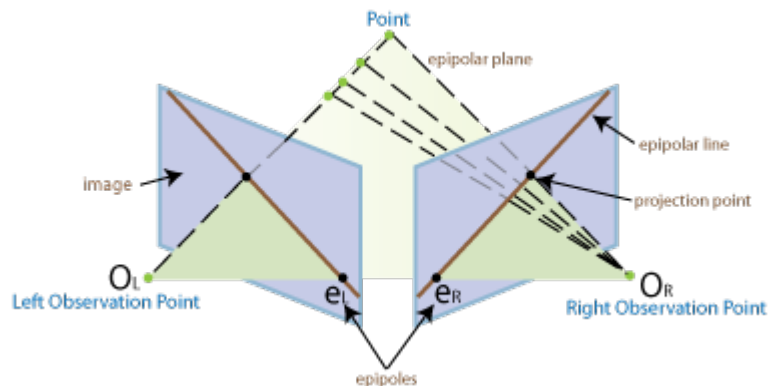
3   Input the fundamental matrix to the `relativeCameraPose` function.
    `relativeCameraPose` returns the orientation and the location of the second camera
    in the coordinate system of the first camera. The location can only be computed up to
    scale, so the distance between two cameras is set to 1. In other words, the distance
    between the cameras is defined to be 1 unit.

4   Determine the 3-D locations of the matched points using `triangulate`. Because the
    pose is up to scale, when you compute the structure, it has the right shape but not
    the actual size.

    The `triangulate` function takes two camera matrices, which you can compute using
    `cameraMatrix`.

5   Use `pcshow` to display the reconstruction, and use `plotCamera` to visualize the
    camera poses.

To recover the scale of the reconstruction, you need additional information. One method
to recover the scale is to detect an object of a known size in the scene. The "Structure
From Motion From Two Views" example shows how to recover scale by detecting a
sphere of a known size in the point cloud of the scene.

Structure from Motion from Multiple Views

For most applications, such as robotics and autonomous driving, SfM uses more than two
views.
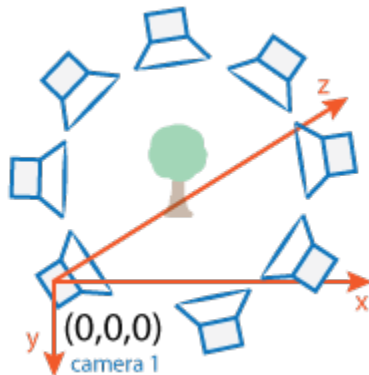
The approach used for SfM from two views can be extended for multiple views. The set of
multiple views used for SfM can be ordered or unordered. The approach taken here
assumes an ordered sequence of views. SfM from multiple views requires point

correspondences across multiple images, called tracks. A typical approach is to compute the tracks from pairwise point correspondences. You can use `viewSet` to manage the pairwise correspondences and find the tracks. Each track corresponds to a 3-D point in the scene. To compute 3-D points from the tracks, use `triangulateMultiview`.
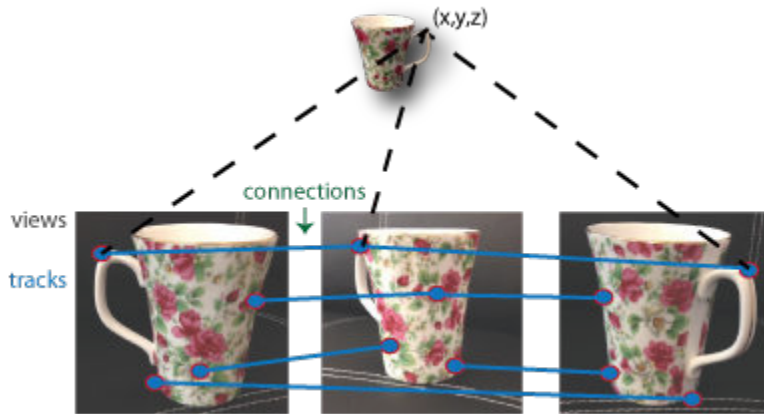
Using the approach in SfM from two views, you can find the pose of camera 2 relative to camera 1. To extend this approach to the multiple view case, find the pose of camera 3 relative to camera 2, and so on. The relative poses must be transformed into a common coordinate system. Typically, all camera poses are computed relative to camera 1 so that all poses are in the same coordinate system. You can use `viewSet` to manage camera poses. The `viewSet` object stores the views and connections between the views.
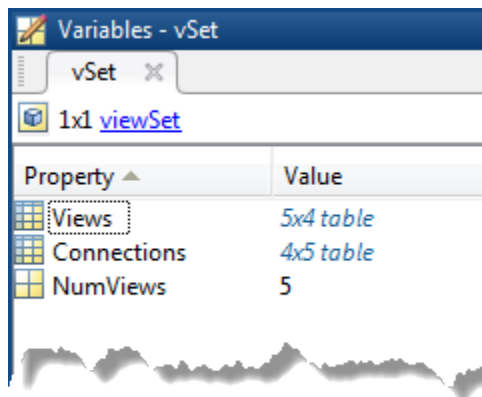
Every camera pose estimation from one view to the next contains errors. The errors arise from imprecise point localization in images, and from noisy matches and imprecise calibration. These errors accumulate as the number of views increases, an effect known as drift. One way to reduce the drift, is to refine camera poses and 3-D point locations. The nonlinear optimization algorithm, called bundle adjustment, implemented by the `bundleAdjustment` function, can be used for the refinement.
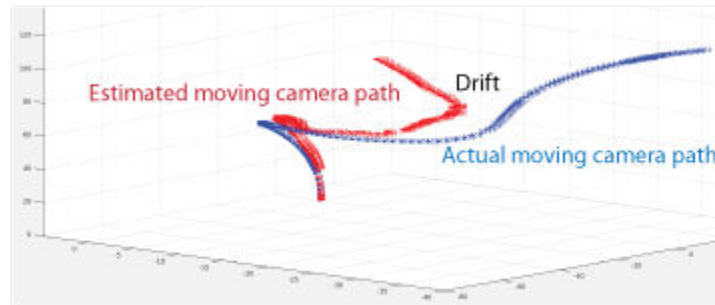
The "Structure From Motion From Multiple Views" example shows how to reconstruct a 3-D scene from a sequence of 2-D views. The example uses the **Camera Calibrator** app to calibrate the camera that takes the views. It uses a `viewSet` object to store and manage the data associated with each view.

See Also

Camera Calibrator | **Stereo Camera Calibrator** | bundleAdjustment | cameraMatrix | estimateFundamentalMatrix | matchFeatures | pointTrack | relativeCameraPose | triangulateMultiview | viewSet | vision.PointTracker

Related Examples

- "Structure From Motion From Two Views"
- "Structure From Motion From Multiple Views"

7

Object Detection

Semantic Segmentation Basics

Segmentation is essential for image analysis tasks. Semantic segmentation describes the process of associating each pixel of an image with a class label, (such as *flower*, *person*, *road*, *sky*, *ocean*, or *car*).
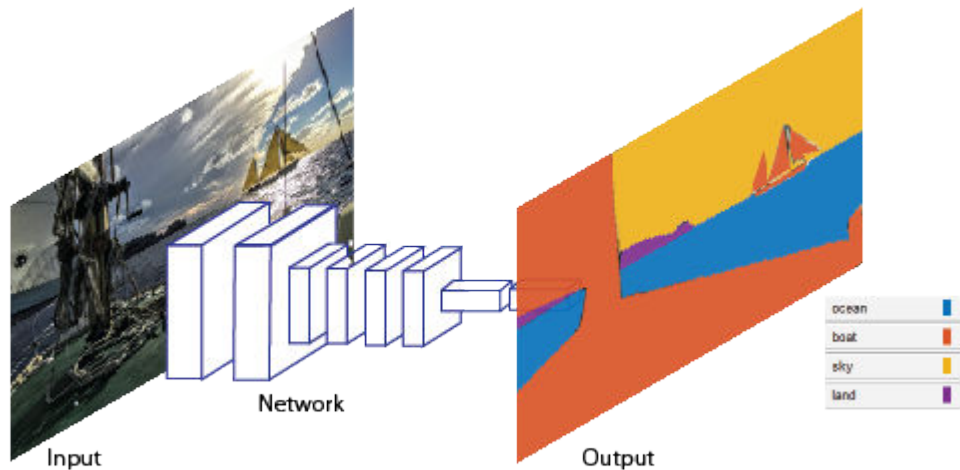
Applications for semantic segmentation include:

- Autonomous driving
- Industrial inspection
- Classification of terrain visible in satellite imagery
- Medical imaging analysis

Train a Semantic Segmentation Network

The steps for training a semantic segmentation network are as follows:

1. "Analyze Training Data for Semantic Segmentation" on page 7-5

2. "Create a Semantic Segmentation Network" on page 7-9

3. "Train A Semantic Segmentation Network" on page 7-13

Label Training Data for Semantic Segmentation

You can use the **Image Labeler** app to interactively label pixels and export the label data for training. The app can also be used to label rectangular regions of interest (ROIs) and scene labels for image classification.
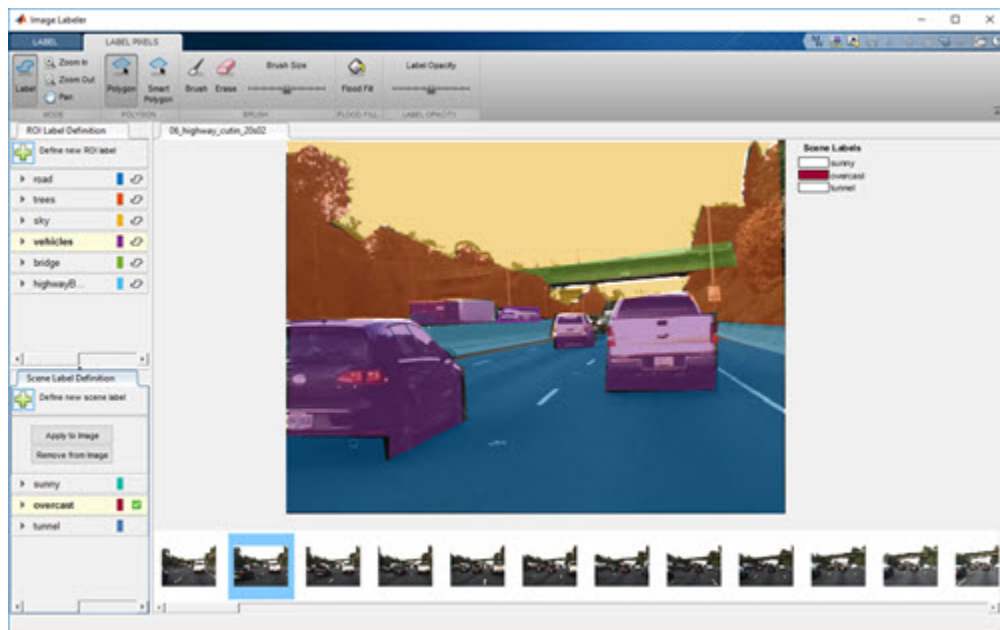
See Also
Image Labeler | evaluateSemanticSegmentation | fcnLayers | pixelClassificationLayer | pixelLabelDatastore | pixelLabelImageDatastore | segnetLayers | semanticSegmentationMetrics | semanticseg

Related Examples

- "Semantic Segmentation Using Deep Learning"
- "Label Pixels for Semantic Segmentation" on page 7-43

More About

- "Deep Learning Basics" (Neural Network Toolbox)

Semantic Segmentation Examples

Analyze Training Data for Semantic Segmentation

To train a semantic segmentation network you need a collection of images and its corresponding collection of pixel labeled images. A pixel labeled image is an image where every pixel value represents the categorical label of that pixel.

The following code loads a small set of images and their corresponding pixel labeled images:

```
dataDir = fullfile(toolboxdir('vision'),'visiondata');
imDir = fullfile(dataDir,'building');
pxDir = fullfile(dataDir,'buildingPixelLabels');
```

Load the image data using an `imageDatastore`. An image datastore can efficiently represent a large collection of images because images are only read into memory when needed.

```
imds = imageDatastore(imDir);
```

Read and display the first image.

```
I = readimage(imds,1);
figure
imshow(I)
```

Load the pixel label images using a `pixelLabelDatastore` to define the mapping between label IDs and categorical names. In the dataset used here, the labels are "sky", "grass", "building", and "sidewalk". The label IDs for these classes are 1, 2, 3, 4, respectively.

Define the class names.

```
classNames = ["sky" "grass" "building" "sidewalk"];
```

Define the label ID for each class name.

```
pixelLabelID = [1 2 3 4];
```

Create a `pixelLabelDatastore`.

```
pxds = pixelLabelDatastore(pxDir,classNames,pixelLabelID);
```

Read the first pixel label image.

```
C = readimage(pxds,1);
```

The output `C` is a categorical matrix where `C(i,j)` is the categorical label of pixel `I(i,j)`.

```
C(5,5)
```

```
ans = categorical
     sky
```

Overlay the pixel labels on the image to see how different parts of the image are labeled.

```
B = labeloverlay(I,C);
figure
imshow(B)
```
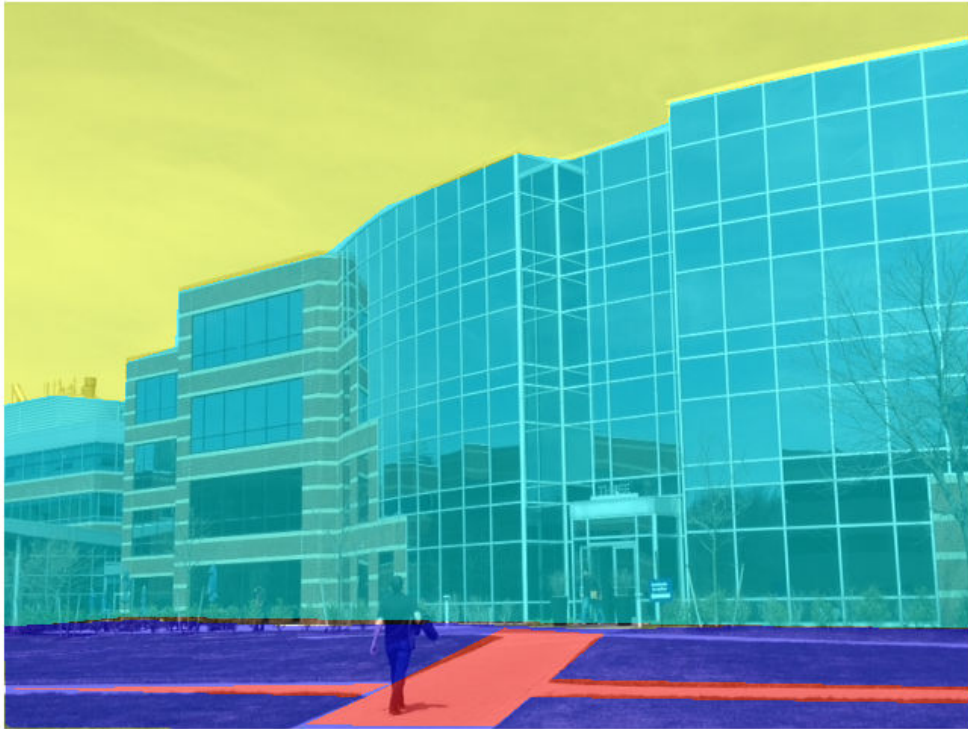
The categorical output format simplifies tasks that require doing things by class names. For instance, you can create a binary mask of just the building:

```
buildingMask = C == 'building';

figure
imshowpair(I, buildingMask,'montage')
```

Create a Semantic Segmentation Network

Create a simple semantic segmentation network and learn about common layers found in many semantic segmentation networks. A common pattern in semantic segmentation networks requires the downsampling of an image between convolutional and ReLU layers, and then upsample the output to match the input size. This operation is analogous to the standard scale-space analysis using image pyramids. During this process however, a network performs the operations using non-linear filters optimized for a specific set of classes that you want to segment.

Create An Image Input Layer

A semantic segmentation network starts with an `imageInputLayer`, which defines the smallest image size the network can process. Most semantic segmentation networks are fully convolutional, which means they can process images that are larger than the specified input size. Here, an image size of [32 32 3] is used for the network to process 64x64 RGB images.

```
inputSize = [32 32 3];
imgLayer = imageInputLayer(inputSize)

imgLayer =
  ImageInputLayer with properties:
```

```
            Name: ''
       InputSize: [32 32 3]

   Hyperparameters
    DataAugmentation: 'none'
       Normalization: 'zerocenter'
```

Create Downsampling Network

Start with the convolution and ReLU layers. The convolution layer padding is selected such that the output size of the convolution layer is the same as the input size. This makes it easier to construct a network because the input and output sizes between most layers remain the same as you progress through the network.

```
filterSize = 3;
numFilters = 32;
conv = convolution2dLayer(filterSize,numFilters,'Padding',1);
relu = reluLayer();
```

The downsampling is performed using a max pooling layer. Create a max pooling layer to downsample the input by a factor of 2 by setting the 'Stride' parameter to 2.

```
poolSize = 2;
maxPoolDownsample2x = maxPooling2dLayer(poolSize,'Stride',2);
```

Stack the convolution, ReLU, and max pooling layers to create a network that downsamples its input by a factor of 4.

```
downsamplingLayers = [
    conv
    relu
    maxPoolDownsample2x
    conv
    relu
    maxPoolDownsample2x
    ]

downsamplingLayers =
  6x1 Layer array with layers:

     1   ''   Convolution   32 3x3 convolutions with stride [1  1] and padding [1  1  1
     2   ''   ReLU          ReLU
     3   ''   Max Pooling   2x2 max pooling with stride [2  2] and padding [0  0  0  0]
     4   ''   Convolution   32 3x3 convolutions with stride [1  1] and padding [1  1  1
```

```
    5   ''   ReLU          ReLU
    6   ''   Max Pooling   2x2 max pooling with stride [2  2] and padding [0  0  0  0]
```

Create Upsampling Network

The upsampling is done using the tranposed convolution layer (also commonly referred to as "deconv" or "deconvolution" layer). When a transposed convolution is used for upsampling, it performs the upsampling and the filtering at the same time.

Create a transposed convolution layer to upsample by 2.

```
filterSize = 4;
transposedConvUpsample2x = transposedConv2dLayer(4,numFilters,'Stride',2,'Cropping',1);
```

The 'Cropping' parameter is set to 1 to make the output size equal twice the input size.

Stack the transposed convolution and relu layers. An input to this set of layers is upsampled by 4.

```
upsamplingLayers = [
    transposedConvUpsample2x
    relu
    transposedConvUpsample2x
    relu
    ]

upsamplingLayers =
  4x1 Layer array with layers:

    1   ''   Transposed Convolution   32 4x4 transposed convolutions with stride [2  2
    2   ''   ReLU                     ReLU
    3   ''   Transposed Convolution   32 4x4 transposed convolutions with stride [2  2
    4   ''   ReLU                     ReLU
```

Create A Pixel Classification Layer

The final set of layers are responsible for making pixel classifications. These final layers process an input that has the same spatial dimensions (height and width) as the input image. However, the number of channels (third dimension) is larger and is equal to number of filters in the last transposed convolution layer. This third dimension needs to be squeezed down to the number of classes we wish to segment. This can be done using a 1-by-1 convolution layer whose number of filters equal the number of classes, e.g. 3.

Create a convolution layer to combine the third dimension of the input feature maps down to the number of classes.

```
numClasses = 3;
conv1x1 = convolution2dLayer(1,numClasses);
```

Following this 1-by-1 convolution layer are the softmax and pixel classification layers.
These two layers combine to predict the categorical label for each image pixel.

```
finalLayers = [
    conv1x1
    softmaxLayer()
    pixelClassificationLayer()
    ]

finalLayers =
  3x1 Layer array with layers:

    1   ''   Convolution              3 1x1 convolutions with stride [1  1] and pa
    2   ''   Softmax                  softmax
    3   ''   Pixel Classification Layer   Cross-entropy loss
```

Stack All Layers

Stack all the layers to complete the semantic segmentation network.

```
net = [
    imgLayer
    downsamplingLayers
    upsamplingLayers
    finalLayers
    ]

net =
  14x1 Layer array with layers:

    1    ''   Image Input              32x32x3 images with 'zerocenter' normalizat:
    2    ''   Convolution              32 3x3 convolutions with stride [1  1] and p
    3    ''   ReLU                     ReLU
    4    ''   Max Pooling              2x2 max pooling with stride [2  2] and padd:
    5    ''   Convolution              32 3x3 convolutions with stride [1  1] and p
    6    ''   ReLU                     ReLU
    7    ''   Max Pooling              2x2 max pooling with stride [2  2] and padd:
    8    ''   Transposed Convolution   32 4x4 transposed convolutions with stride |
    9    ''   ReLU                     ReLU
   10    ''   Transposed Convolution   32 4x4 transposed convolutions with stride |
   11    ''   ReLU                     ReLU
   12    ''   Convolution              3 1x1 convolutions with stride [1  1] and pa
```

```
13   ''   Softmax                    softmax
14   ''   Pixel Classification Layer  Cross-entropy loss
```

This network is ready to be trained using `trainNetwork` from Neural Network Toolbox™.

Train A Semantic Segmentation Network

Load the training data.

```
dataSetDir = fullfile(toolboxdir('vision'),'visiondata','triangleImages');
imageDir = fullfile(dataSetDir,'trainingImages');
labelDir = fullfile(dataSetDir,'trainingLabels');
```

Create an image datastore for the images.

```
imds = imageDatastore(imageDir);
```

Create a `pixelLabelDatastore` for the ground truth pixel labels.

```
classNames = ["triangle","background"];
labelIDs   = [255 0];
pxds = pixelLabelDatastore(labelDir,classNames,labelIDs);
```

Visualize training images and ground truth pixel labels.

```
I = read(imds);
C = read(pxds);

I = imresize(I,5);
L = imresize(uint8(C),5);
imshowpair(I,L,'montage')
```

Create a semantic segmentation network. This network uses a simple semantic segmentation network based on a downsampling and upsampling design.

```
numFilters = 64;
filterSize = 3;
numClasses = 2;
layers = [
    imageInputLayer([32 32 1])
    convolution2dLayer(filterSize,numFilters,'Padding',1)
    reluLayer()
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(filterSize,numFilters,'Padding',1)
    reluLayer()
    transposedConv2dLayer(4,numFilters,'Stride',2,'Cropping',1);
    convolution2dLayer(1,numClasses);
    softmaxLayer()
    pixelClassificationLayer()
    ]
```

```
layers =
  10x1 Layer array with layers:

    1   ''    Image Input              32x32x1 images with 'zerocenter' normalizat:
    2   ''    Convolution              64 3x3 convolutions with stride [1  1] and p
    3   ''    ReLU                     ReLU
    4   ''    Max Pooling              2x2 max pooling with stride [2  2] and padd:
```

```
 5   ''   Convolution                64 3x3 convolutions with stride [1  1] and p
 6   ''   ReLU                       ReLU
 7   ''   Transposed Convolution     64 4x4 transposed convolutions with stride [
 8   ''   Convolution                2 1x1 convolutions with stride [1  1] and pa
 9   ''   Softmax                    softmax
10   ''   Pixel Classification Layer Cross-entropy loss
```

Setup training options.

```
opts = trainingOptions('sgdm', ...
    'InitialLearnRate',1e-3, ...
    'MaxEpochs',100, ...
    'MiniBatchSize',64);
```

Create a pixel label image datastore that contains training data.

```
trainingData = pixelLabelImageDatastore(imds,pxds);
```

Train the network.

```
net = trainNetwork(trainingData,layers,opts);
```

```
Training on single GPU.
Initializing image normalization.
```

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:00	56.29%	0.6931	0.001
17	50	00:00:04	95.07%	0.5546	0.001
34	100	00:00:09	95.38%	0.4408	0.001
50	150	00:00:13	94.53%	0.3785	0.001
67	200	00:00:18	95.07%	0.3278	0.001
84	250	00:00:22	95.38%	0.2923	0.001
100	300	00:00:27	94.53%	0.2820	0.001

Read and display a test image.

```
testImage = imread('triangleTest.jpg');
imshow(testImage)
```
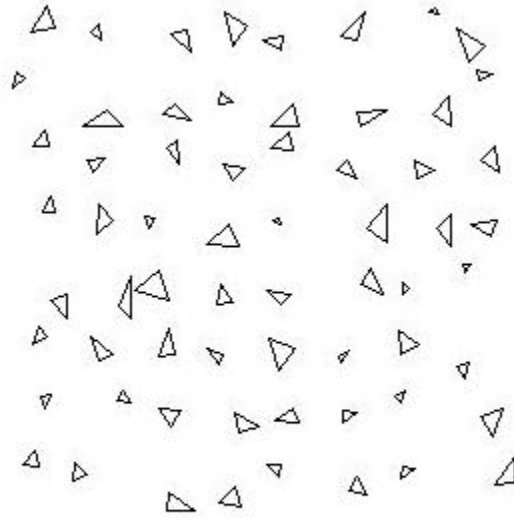
Segment the test image and display the results.

```
C = semanticseg(testImage,net);
B = labeloverlay(testImage,C);
imshow(B)
```
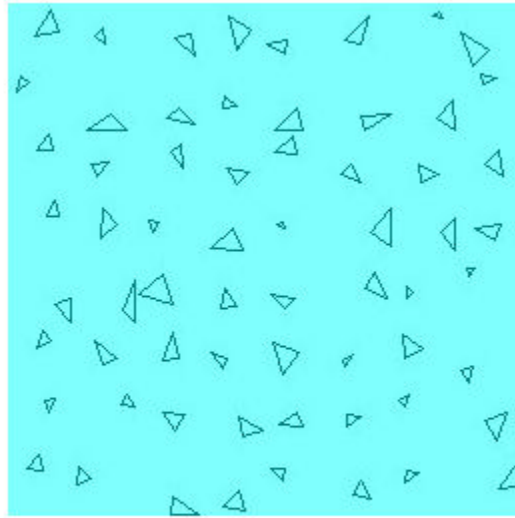
Improve the results

The network failed to segment the triangles and classified every pixel as "background". The training appeared to be going well with training accuracies greater than 90%. However, the network only learned to classify the background class. To understand why this happened, you can count the occurrence of each pixel label across the dataset.

```
tbl = countEachLabel(trainingData)
```

```
tbl=2×3 table
        Name         PixelCount     ImagePixelCount
    _____    _____     _____

    'triangle'           10326         2.048e+05
    'background'     1.9447e+05         2.048e+05
```

The majority of pixel labels are for the background. The poor results are due to the class imbalance. Class imbalance biases the learning process in favor of the dominant class.

That's why every pixel is classified as "background". To fix this, use class weighting to balance the classes. There are several methods for computing class weights. One common method is inverse frequency weighting where the class weights are the inverse of the class frequencies. This increases weight given to under-represented classes.

```
totalNumberOfPixels = sum(tbl.PixelCount);
frequency = tbl.PixelCount / totalNumberOfPixels;
classWeights = 1./frequency
```

classWeights = *2×1*

```
    19.8334
     1.0531
```

Class weights can be specified using the `pixelClassificationLayer`. Update the last layer to use a `pixelClassificationLayer` with inverse class weights.

```
layers(end) = pixelClassificationLayer('ClassNames',tbl.Name,'ClassWeights',classWeight
```

Train network again.

```
net = trainNetwork(trainingData,layers,opts);
```

```
Training on single GPU.
Initializing image normalization.
```

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:00	57.17%	0.6945	0.001
17	50	00:00:04	78.32%	0.6677	0.001
34	100	00:00:09	80.11%	0.4283	0.001
50	150	00:00:14	85.69%	0.3923	0.001
67	200	00:00:19	87.64%	0.3395	0.001
84	250	00:00:24	89.53%	0.3038	0.001
100	300	00:00:28	90.40%	0.2855	0.001

Try to segment the test image again.

```
C = semanticseg(testImage,net);
B = labeloverlay(testImage,C);
imshow(B)
```
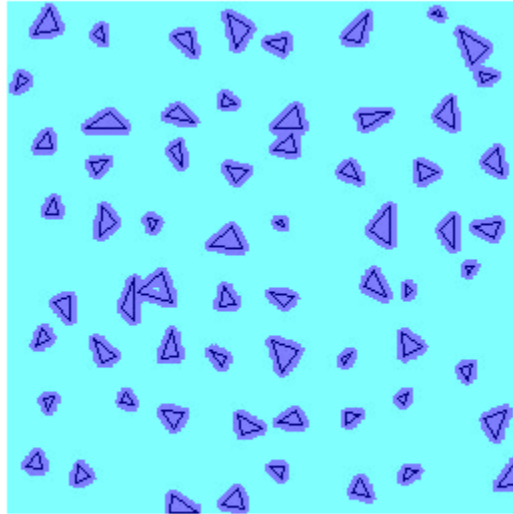
Using class weighting to balance the classes produced a better segmentation result. Additional steps to improve the results include increasing the number of epochs used for training, adding more training data, or modifying the network.

Evaluate and inspect the results of semantic segmentation

Import a test data set, run a pre-trained semantic segmentation network, and evaluate and inspect semantic segmentation quality metrics for the predicted results.

Import a data set

The `triangleImages` data set has 100 test images with ground truth labels. Define the location of the data set.

```
dataSetDir = fullfile(toolboxdir('vision'), 'visiondata', 'triangleImages');
```

Define the location of the test images.

```
testImagesDir = fullfile(dataSetDir, 'testImages');
```

Create an `imageDatastore` object holding the test images.

```
imds = imageDatastore(testImagesDir);
```

Define the location of the ground truth labels.

```
testLabelsDir = fullfile(dataSetDir, 'testLabels');
```

Define the class names and their associated label IDs. The label IDs are the pixel values used in the image files to represent each class.

```
classNames = ["triangle" "background"];
labelIDs   = [255 0];
```

Create a `pixelLabelDatastore` object holding the ground truth pixel labels for the test images.

```
pxdsTruth = pixelLabelDatastore(testLabelsDir, classNames, labelIDs);
```

Run a semantic segmentation classifier

Load a semantic segmentation network that has been trained on the training images of `triangleImages`.

```
net = load('triangleSegmentationNetwork.mat');
net = net.net;
```

Run the network on the test images. Predicted labels are written to disk in a temporary directory and returned as a `pixelLabelDatastore` object.

```
pxdsResults = semanticseg(imds, net, "WriteLocation", tempdir);
```

```
Running semantic segmentation network
-------------------------------------
* Processing 100 images.
* Progress: 100.00%
```

Evaluate the quality of the prediction

The predicted labels are compared to the ground truth labels. While the semantic segmentation metrics are being computed, progress is printed to the Command Window.

```
metrics = evaluateSemanticSegmentation(pxdsResults, pxdsTruth);
```

```
Evaluating semantic segmentation results
-------------------------------------[=============================================
Elapsed time: 00:00:01
Estimated time remaining: 00:00:00
* Finalizing... Done.
* Data set metrics:
```

GlobalAccuracy	MeanAccuracy	MeanIoU	WeightedIoU	MeanBFScore
0.90624	0.95085	0.61588	0.87529	0.40652

Inspect class metrics

Display the classification accuracy, the intersection over union (IoU), and the boundary F-1 score for each class in the data set.

```
metrics.ClassMetrics
```

ans=*2×3 table*

	Accuracy	IoU	MeanBFScore
triangle	1	0.33005	0.028664
background	0.9017	0.9017	0.78438

Display the confusion matrix

Display the confusion matrix in the Command Window.

```
metrics.ConfusionMatrix
```

ans=*2×2 table*

	triangle	background
triangle	4730	0
background	9601	88069

Visualize the normalized confusion matrix as a heat map in a figure window.

```
normConfMatData = metrics.NormalizedConfusionMatrix.Variables;
figure
```

```
h = heatmap(classNames, classNames, 100 * normConfMatData);
h.XLabel = 'Predicted Class';
h.YLabel = 'True Class';
h.Title  = 'Normalized Confusion Matrix (%)';
```
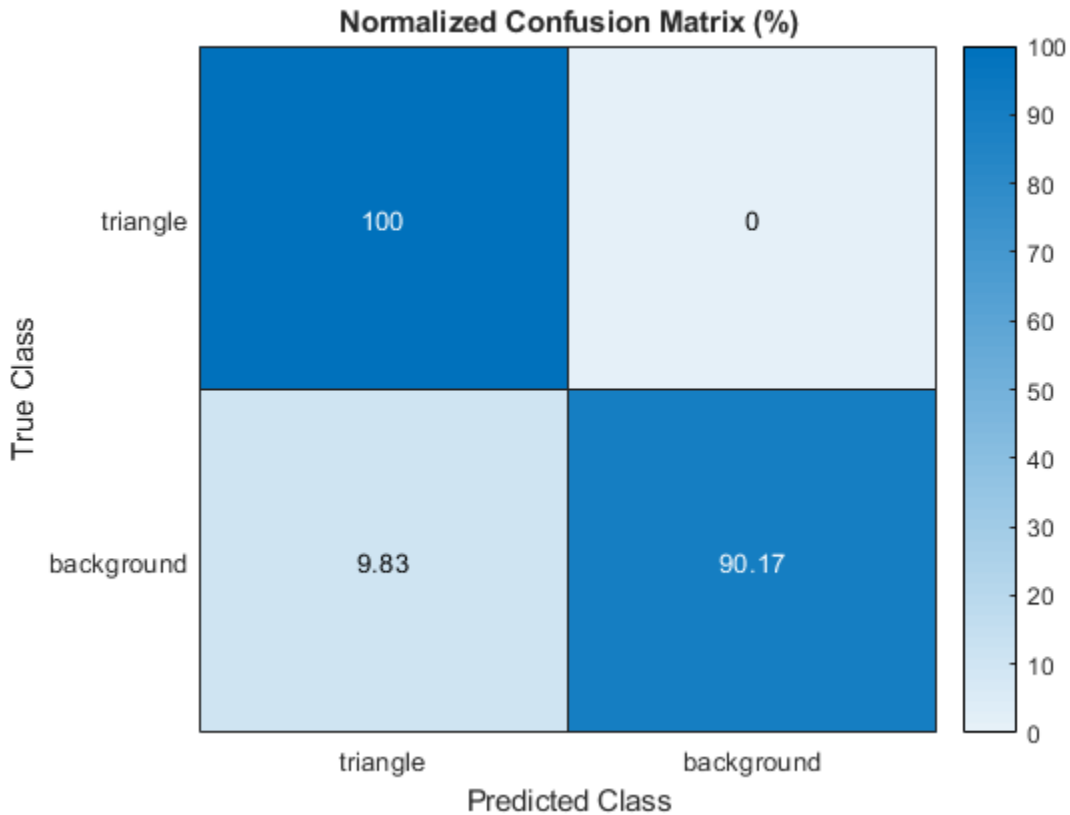

Inspect an image metric

Visualize the histogram of the per-image intersection over union (IoU).

```
imageIoU = metrics.ImageMetrics.MeanIoU;
figure
histogram(imageIoU)
title('Image Mean IoU')
```

Image Mean IoU
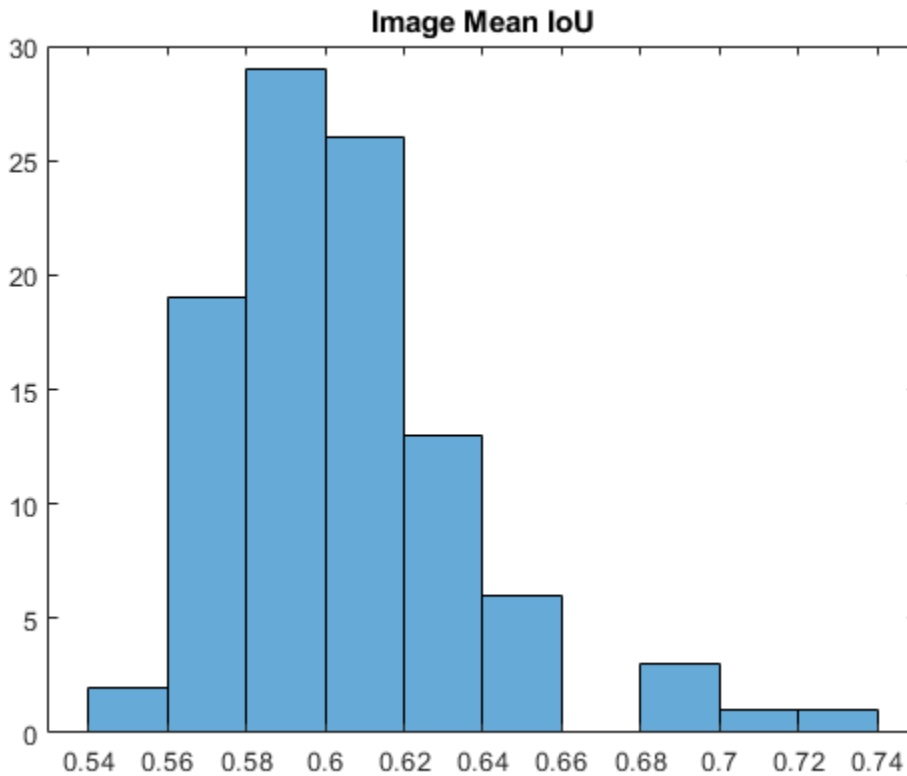
Find the test image with the lowest IoU.

```
[minIoU, worstImageIndex] = min(imageIoU);
minIoU = minIoU(1);
worstImageIndex = worstImageIndex(1);
```

Read the test image with the worst IoU, its ground truth labels, and its predicted labels for comparison.

```
worstTestImage = readimage(imds, worstImageIndex);
worstTrueLabels = readimage(pxdsTruth, worstImageIndex);
worstPredictedLabels = readimage(pxdsResults, worstImageIndex);
```

Convert the label images to images that can be displayed in a figure window.

```
worstTrueLabelImage = im2uint8(worstTrueLabels == classNames(1));
worstPredictedLabelImage = im2uint8(worstPredictedLabels == classNames(1));
```

Display the worst test image, the ground truth, and the prediction.

```
worstMontage = cat(4,worstTestImage, worstTrueLabelImage, worstPredictedLabelImage);
worstMontage = imresize(worstMontage, 4, "nearest");
figure
montage(worstMontage, 'Size', [1 3])
title(['Test image vs. Truth vs. Prediction. IoU = ' num2str(minIoU)])
```


Test image vs. Truth vs. Prediction. IoU = 0.55519
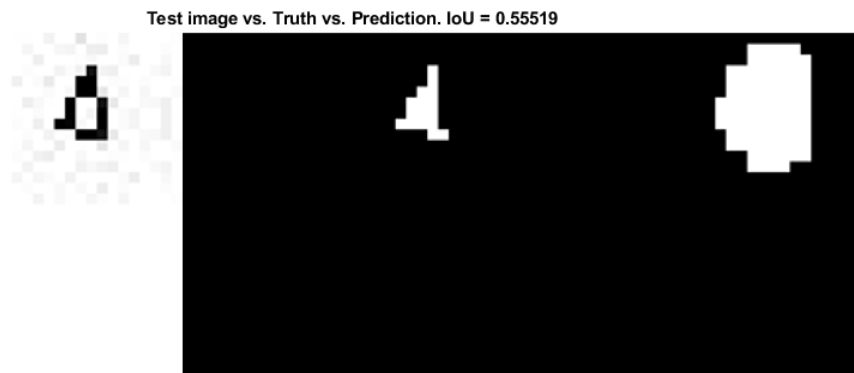
Similarly, find the test image with the highest IoU.

```
[maxIoU, bestImageIndex] = max(imageIoU);
maxIoU = maxIoU(1);
bestImageIndex = bestImageIndex(1);
```

Repeat the previous steps to read, convert, and display the test image with the best IoU with its ground truth and predicted labels.

```
bestTestImage = readimage(imds, bestImageIndex);
bestTrueLabels = readimage(pxdsTruth, bestImageIndex);
bestPredictedLabels = readimage(pxdsResults, bestImageIndex);

bestTrueLabelImage = im2uint8(bestTrueLabels == classNames(1));
bestPredictedLabelImage = im2uint8(bestPredictedLabels == classNames(1));

bestMontage = cat(4,bestTestImage, bestTrueLabelImage, bestPredictedLabelImage);
```

```
bestMontage = imresize(bestMontage, 4, "nearest");
figure
montage(bestMontage, 'Size', [1 3])
title(['Test image vs. Truth vs. Prediction. IoU = ' num2str(maxIoU)])
```


Test image vs. Truth vs. Prediction. IoU = 0.72502
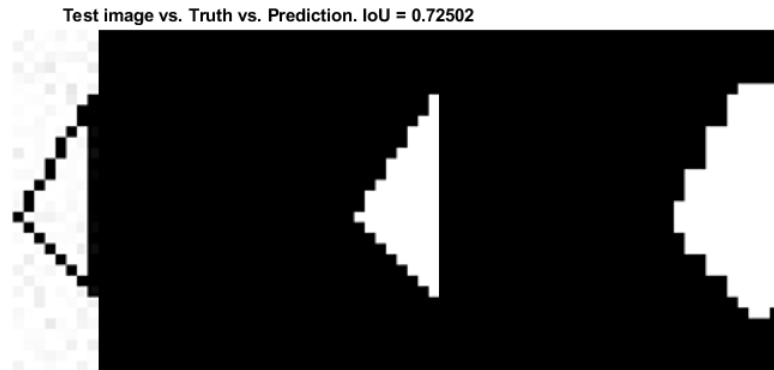
Specify metrics to evaluate

Optionally, list the metric(s) you would like to evaluate using the 'Metrics' parameter.

Define the metrics to compute.

```
evaluationMetrics = ["accuracy" "iou"];
```

Compute these metrics for the triangleImages test data set.

```
metrics = evaluateSemanticSegmentation(pxdsResults, pxdsTruth, "Metrics", evaluationMet
```

```
Evaluating semantic segmentation result[==================================================
Elapsed time: 00:00:00
Estimated time remaining: 00:00:00
* Finalizing... Done.
* Data set metrics:

    MeanAccuracy    MeanIoU
    _____    _____

      0.95085       0.61588
```

Display the chosen metrics for each class.

```
metrics.ClassMetrics
```

ans=*2×2 table*

	Accuracy	IoU
triangle	1	0.33005
background	0.9017	0.9017

Import Pixel Labeled Dataset For Semantic Segmentation

This example shows you how to import a pixel labeled dataset for semantic segmentation networks.

A pixel labeled dataset is a collection of images and a corresponding set of ground truth pixel labels used for training semantic segmentation networks. There are many public datasets that provide annotated images with per-pixel labels. To illustrate the steps for importing these types of datasets, the example uses the CamVid dataset from the University of Cambridge [1].

The CamVid dataset is a collection of images containing street level views obtained while driving. The dataset provides pixel-level labels for 32 semantic classes including car, pedestrian, and road. The steps shown to import CamVid can be used to import other pixel labeled datasets.

Download CamVid Dataset

Download the CamVid image data from the following URLs:

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsF
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledAppro

outputFolder = fullfile(tempdir, 'CamVid');
imageDir = fullfile(outputFolder,'images');
labelDir = fullfile(outputFolder,'labels');

if ~exist(outputFolder, 'dir')
    disp('Downloading 557 MB CamVid data set...');

    unzip(imageURL, imageDir);
```

```
    unzip(labelURL, labelDir);
end
```

Note: Download time of the data depends on your internet connection. The commands used above will block MATLAB® until the download is complete. Alternatively, you can use your web browser to first download the dataset to your local disk. To use the file you downloaded from the web, change the `outputFolder` variable above to the location of the downloaded file.

CamVid Pixel Labels

The CamVid data set encodes the pixel labels as RGB images, where each class is represented by an RGB color. Here are the classes the dataset defines along with their RGB encodings.

```
classNames = [ ...
    "Animal", ...
    "Archway", ...
    "Bicyclist", ...
    "Bridge", ...
    "Building", ...
    "Car", ...
    "CartLuggagePram", ...
    "Child", ...
    "Column_Pole", ...
    "Fence", ...
    "LaneMkgsDriv", ...
    "LaneMkgsNonDriv", ...
    "Misc_Text", ...
    "MotorcycleScooter", ...
    "OtherMoving", ...
    "ParkingBlock", ...
    "Pedestrian", ...
    "Road", ...
    "RoadShoulder", ...
    "Sidewalk", ...
    "SignSymbol", ...
    "Sky", ...
    "SUVPickupTruck", ...
    "TrafficCone", ...
    "TrafficLight", ...
    "Train", ...
    "Tree", ...
    "Truck_Bus", ...
```

```
    "Tunnel", ...
    "VegetationMisc", ...
    "Wall"];
```

Define the mapping between label indices and class names such that `className(k)` corresponds to `labelIDs(k,:)`.

```
labelIDs = [ ...
    064 128 064; ... % "Animal"
    192 000 128; ... % "Archway"
    000 128 192; ... % "Bicyclist"
    000 128 064; ... % "Bridge"
    128 000 000; ... % "Building"
    064 000 128; ... % "Car"
    064 000 192; ... % "CartLuggagePram"
    192 128 064; ... % "Child"
    192 192 128; ... % "Column_Pole"
    064 064 128; ... % "Fence"
    128 000 192; ... % "LaneMkgsDriv"
    192 000 064; ... % "LaneMkgsNonDriv"
    128 128 064; ... % "Misc_Text"
    192 000 192; ... % "MotorcycleScooter"
    128 064 064; ... % "OtherMoving"
    064 192 128; ... % "ParkingBlock"
    064 064 000; ... % "Pedestrian"
    128 064 128; ... % "Road"
    128 128 192; ... % "RoadShoulder"
    000 000 192; ... % "Sidewalk"
    192 128 128; ... % "SignSymbol"
    128 128 128; ... % "Sky"
    064 128 192; ... % "SUVPickupTruck"
    000 000 064; ... % "TrafficCone"
    000 064 064; ... % "TrafficLight"
    192 064 128; ... % "Train"
    128 128 000; ... % "Tree"
    192 128 192; ... % "Truck_Bus"
    064 000 064; ... % "Tunnel"
    192 192 000; ... % "VegetationMisc"
    064 192 000];    % "Wall"
```

Note that other datasets have different formats of encoding data. For example, the PASCAL VOC [2] dataset uses numeric label IDs between 0 and 21 to encode their class labels.

Visualize the pixel labels for one of the CamVid images.

```matlab
labels = imread(fullfile(labelDir,'0001TP_006690_L.png'));
figure
imshow(labels)

% Add colorbar to show class to color mapping.
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterp
colormap(labelIDs./255)
```

Load CamVid Data

A pixel labeled dataset can be loaded using an `imageDatastore` and a `pixelLabelDatastore`.

Create an `imageDatastore` to load the CamVid images.

```
imds = imageDatastore(fullfile(imageDir,'701_StillsRaw_full'));
```

Create a `pixelLabelDatastore` to load the CamVid pixel labels.

```
pxds = pixelLabelDatastore(labelDir,classNames,labelIDs);
```

Read the 10th image and corresponding pixel label image.

```
I = readimage(imds,10);
C = readimage(pxds,10);
```

The pixel label image is returned as a categorical array where `C(i,j)` is the categorical label assigned to pixel `I(i,j)`. Display the pixel label image on top of the image.

```
B = labeloverlay(I,C,'Colormap',labelIDs./255);
figure
imshow(B)

% Add a colorbar
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterp
colormap(labelIDs./255)
```

Undefined or Void Labels

It is common for pixel labeled datasets to include "undefined" or "void" labels. These are used to designate pixels that were not labeled. For example, in CamVid, the label ID [0 0 0] is used to designate the "void" class. Training algorithms and evaluation algorithms are not expected to include these labels in any computations.

The "void" class need not be explicitly named when using `pixelLabelDatastore`. Any label ID that is not mapped to a class name is automatically labeled "undefined" and is excluded from computations. To see the undefined pixels, use `isundefined` to create a mask and then display it on top of the image.

```
undefinedPixels = isundefined(C);
B = labeloverlay(I,undefinedPixels);
figure
imshow(B)
title('Undefined Pixel Labels')
```


Undefined Pixel Labels

Combine Classes

When working with public datasets, you may need to combine some of the classes to better suit your application. For example, you may want to train a semantic segmentation network that segments a scene into 4 classes: road, sky, vehicle, pedestrian, and

background. To do this with the CamVid dataset, group the label IDs defined above to fit the new classes. First, define the new class names.

```
newClassNames = ["road","sky","vehicle","pedestrian","background"];
```

Next, group label IDs using a cell array of M-by-3 matrices.

```
groupedLabelIDs = {
    % road
    [
    128 064 128; ... % "Road"
    128 000 192; ... % "LaneMkgsDriv"
    192 000 064; ... % "LaneMkgsNonDriv"
    000 000 192; ... % "Sidewalk"
    064 192 128; ... % "ParkingBlock"
    128 128 192; ... % "RoadShoulder"
    ]

    % "sky"
    [
    128 128 128; ... % "Sky"
    ]

    % "vehicle"
    [
    064 000 128; ... % "Car"
    064 128 192; ... % "SUVPickupTruck"
    192 128 192; ... % "Truck_Bus"
    192 064 128; ... % "Train"
    000 128 192; ... % "Bicyclist"
    192 000 192; ... % "MotorcycleScooter"
    128 064 064; ... % "OtherMoving"
    ]

    % "pedestrian"
    [
    064 064 000; ... % "Pedestrian"
    192 128 064; ... % "Child"
    064 000 192; ... % "CartLuggagePram"
    064 128 064; ... % "Animal"
    ]

    % "background"
    [
    128 128 000; ... % "Tree"
```

```
        192 192 000; ... % "VegetationMisc"
        192 128 128; ... % "SignSymbol"
        128 128 064; ... % "Misc_Text"
        000 064 064; ... % "TrafficLight"
        064 064 128; ... % "Fence"
        192 192 128; ... % "Column_Pole"
        000 000 064; ... % "TrafficCone"
        000 128 064; ... % "Bridge"
        128 000 000; ... % "Building"
        064 192 000; ... % "Wall"
        064 000 064; ... % "Tunnel"
        192 000 128; ... % "Archway"
        ]
        };
```

Create a `pixelLabelDatastore` using the new class and label IDs.

```
pxds = pixelLabelDatastore(labelDir,newClassNames,groupedLabelIDs);
```

Read the 10th pixel label image and display it on top of the image.

```
C = readimage(pxds,10);
cmap = jet(numel(newClassNames));
B = labeloverlay(I,C,'Colormap',cmap);
figure
imshow(B)

% add colorbar
N = numel(newClassNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(newClassNames),'Ticks',ticks,'TickLength',0,'TickLabelInt
colormap(cmap)
```
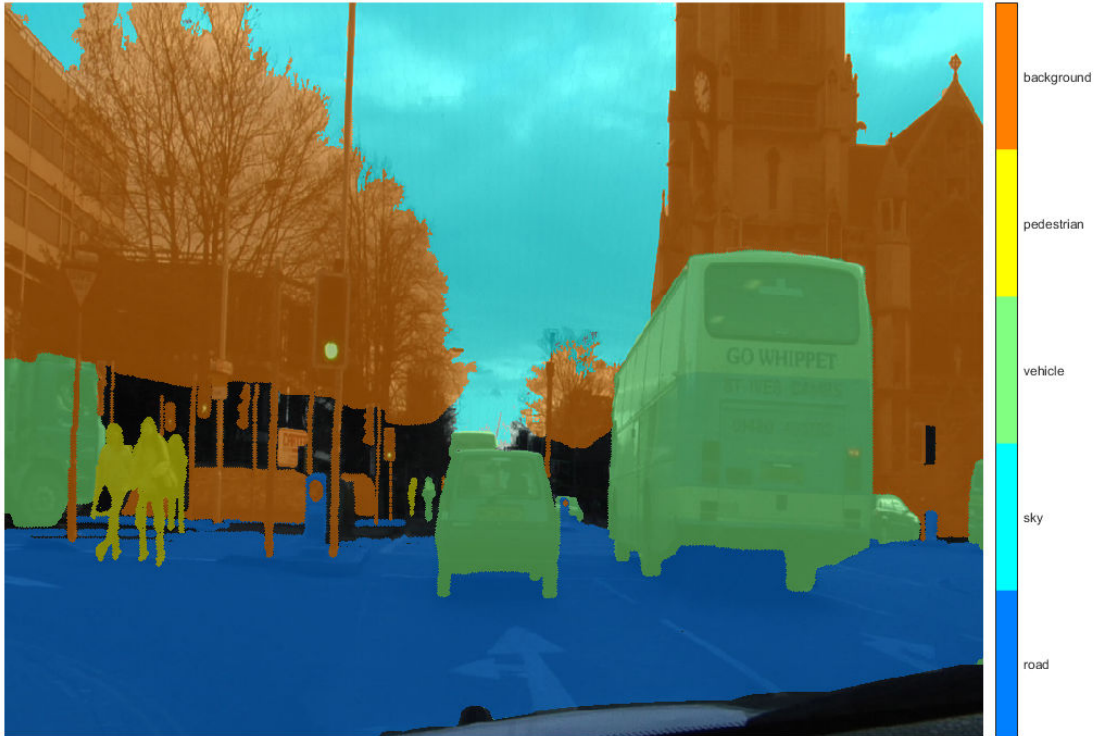
The pixelLabelDatastore with the new class names can now be used to train a network for the 4 classes without having to modify the original CamVid pixel labels.

References

[1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic object classes in video: A high-definition ground truth database." *Pattern Recognition Letters* 30.2 (2009): 88-97.

[2] Everingham, M., et al. "The PASCAL visual object classes challenge 2012 results." *See http://www. pascal-network. org/challenges/VOC/voc2012/workshop/index. html*. Vol. 5. 2012.

Train an Object Detector from Ground Truth Data

You can use the **Image Labeler** and Computer Vision System Toolbox objects and functions to train an object detector from your ground truth data. Use the app to interactively label ground truth data in an image collection. Then use the `objectDetectorTrainingData` function to create training data from the ground truth data.
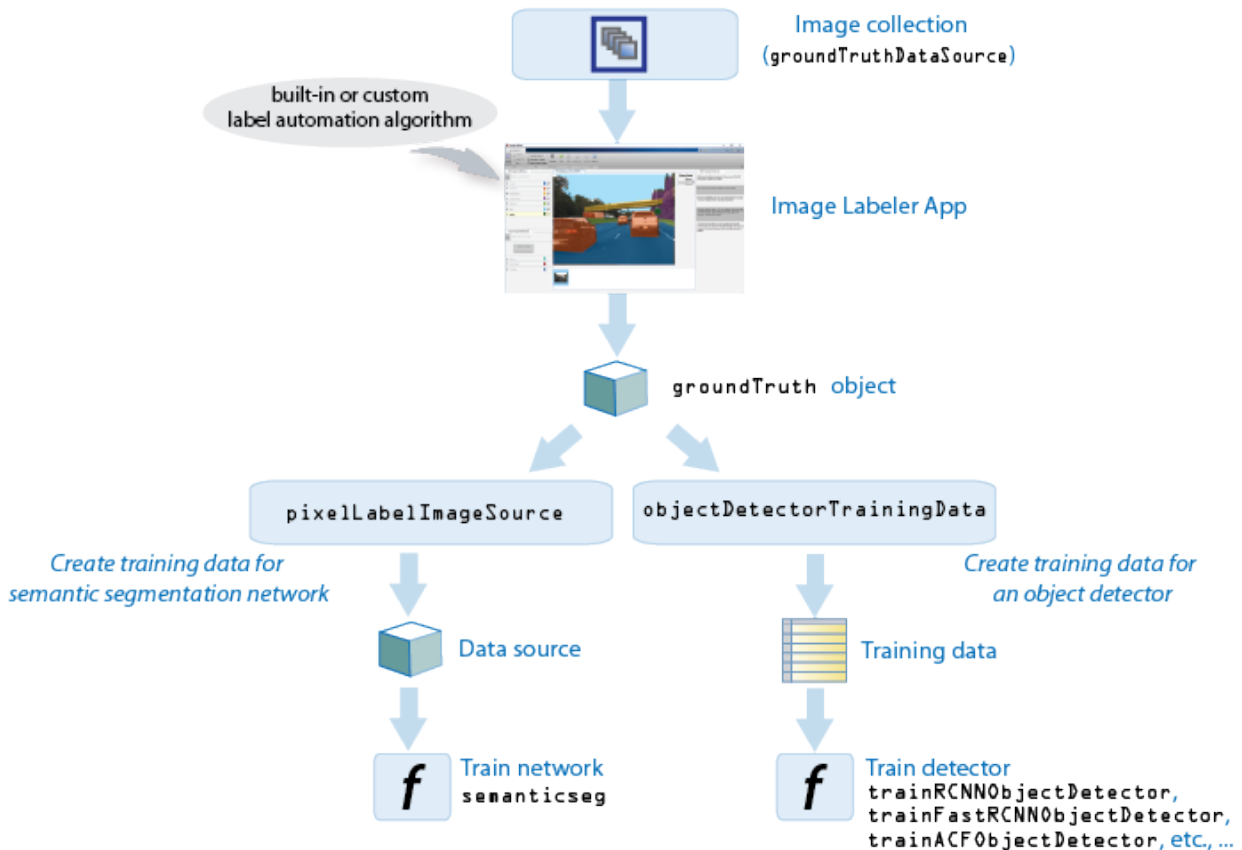
1. **Load data for labeling**: Load an image collection from a file or `ImageDatastore` object into the **Image Labeler** app.

2 **Label data and select an automation algorithm**: Create ROI and scene labels. See "Define Ground Truth for Image Collections" on page 7-54. You can choose from one of the built-in algorithms or create your own custom algorithm to label objects in your data. To learn how to create your own automation algorithm, see "Create Automation Algorithm for Image Labeling" on page 7-39.

3 **Export labels**: After labeling your data, you can export the labels to the workspace or save them to a file. The labels are exported as a `groundTruth` object. Label the entire set of image collections to obtain an array of `groundTruth` objects.

4 **Create training data**: You can use the `objectDetectorTrainingData` function to create training data from the `groundTruth` objects, to use for an object detector.

5 **Train an object detector**: Use one of several training "Object Detectors" with the training data.

See Also

Apps
Image Labeler

Functions
`groundTruth` | `groundTruthDataSource` | `objectDetectorTrainingData` | `trainACFObjectDetector` | `trainFasterRCNNObjectDetector` | `trainRCNNObjectDetector` | `trainRCNNObjectDetector`

More About

- "Define Ground Truth for Image Collections" on page 7-54
- "Create Automation Algorithm for Image Labeling" on page 7-39

Create Automation Algorithm for Image Labeling

The **Image Labeler** app enables you to label ground truth data in a collection of images. You can use an automation algorithm to automatically label your data by creating and importing a custom automation algorithm.

Create Custom Label Automation Algorithm for the Image Labeler

The `vision.labeler.AutomationAlgorithm` class enables you to define a custom label automation algorithm in the **Image Labeler** app. You can use the class to define the interface used by the app to run an automation algorithm.

To define and use a custom automation algorithm with your loaded data source:

1 **Create the automation folder**: Create a `+vision/+labeler/` folder within a folder that is on the MATLAB path. For example, if the folder `/local/MyProject` is on the MATLAB path, then create the `+vision/+labeler/` folder hierarchy as follows:

```
projectFolder = fullfile('local','MyProject');
automationFolder = fullfile('+vision','+labeler');
mkdir(projectFolder,automationFolder)
```

2 **Define a class that inherits from the AutomationAlgorithm class**: At the MATLAB command prompt, enter `imageLabeler` to open the app. Then click **Select Algorithm** > **Add Algorithm** > **Create new algorithm** to open the `vision.labeler.AutomationAlgorithm` class template. Define your algorithm by following the instructions in the header and comments in the class.

3 **Save the file**: Save the file to the `+vision/+labeler` package folder to use your custom algorithm from within the app. To add a folder to the path, use the `addpath` function.

4 **Refresh the algorithm list**: To start using your custom algorithm, refresh the algorithm list for it to display in the list of algorithms. In the app, click **Select Algorithm** > **Refresh list** in the app.

Import a Custom Algorithm into the Labeler

Alternatively, to import your custom algorithm, click **Select Algorithm** > **Add Algorithm** > **Import Algorithm** and then refresh the list.

Custom Algorithm Execution

The properties and methods in your automation algorithm class define how the class interacts with the **Automate** button in the labeler app.

When you click **Automate**, the app checks each label definition in the ROI and Scene label definition list by using the checkLabelDefinition method defined in your custom algorithm. Label definitions that return true are retained for automation. Label definitions that return false are disabled and not included. Use the checkLabelDefinition method to choose a subset of label definitions that are valid for your custom algorithm. For example, if your custom algorithm is a semantic segmentation algorithm, use this method to return false for label definitions that are not of type PixelLabel.
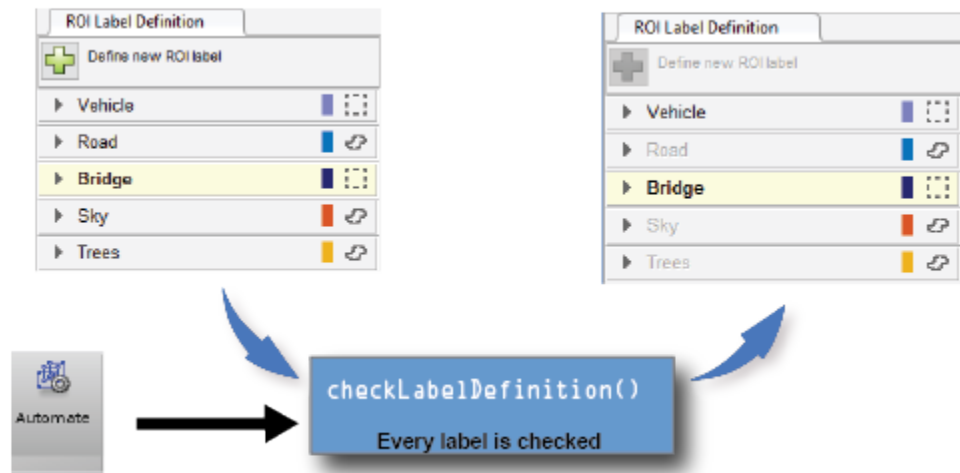
After you select the algorithm. Then click **Automate**, click **Settings**, which enables you to modify custom app settings. To control the **Settings** options, use the settingsDialog method.

When you first run the algorithm, the app calls the `checkSetup` method to check if it is ready for execution. If the method returns `true`, the app calls the `initialize` method and then the `run` method on every image selected for automation. Then, the app calls the `terminate` method.

Use the `checkSetup` method to check whether all conditions needed for your custom algorithm are set up correctly. For example, before running the algorithm, check that the scene contains at least one ROI label before running the algorithm. Use the `initialize` method to initialize the state for your custom algorithm by using the image. Use the `run` method to implement the core of the algorithm that computes and returns labels for each image. Use the `terminate` method to clean up or terminate the state after the algorithm runs.
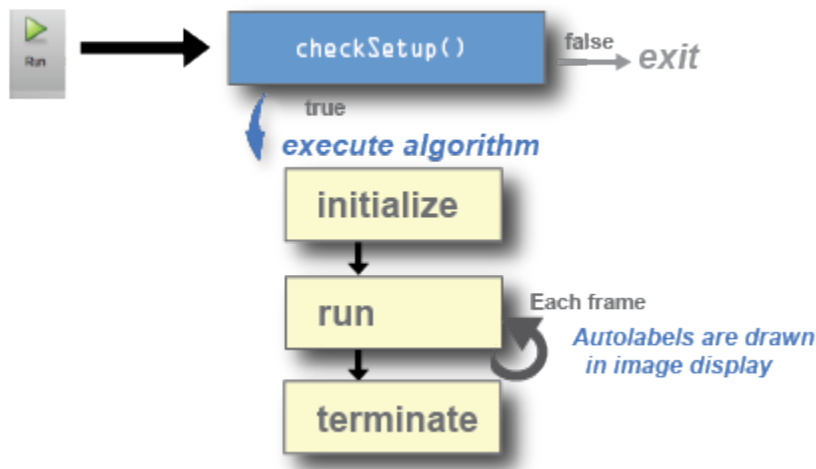
See Also

Image Labeler | `groundTruth` | `groundTruthDataSource` | `vision.labeler.AutomationAlgorithm`

Related Examples

*   "Automate Ground Truth Labeling of Lane Boundaries" (Automated Driving System Toolbox)

More About

- "Define Ground Truth for Image Collections" on page 7-54
- "Train an Object Detector from Ground Truth Data" on page 7-37
- "Create Automation Algorithm for Image Labeling" on page 7-39

Label Pixels for Semantic Segmentation

The **Image Labeler** app enables you to assign pixel labels manually. Each pixel can have at most one pixel label. The labels are used to create ground truth data for training semantic segmentation algorithms.

Start Pixel Labeling

Begin by loading an image into **Image Labeler** and defining pixel ROI labels. For more details, see "Define Ground Truth for Image Collections" on page 7-54.

Select a pixel label definition from the **ROI Label Definition** pane. A **Label Pixels** tab opens, containing tools to label pixels manually using polygons, brushes, or flood fill. You can use the labeling tools in any order. This tab also has controls to adjust the display of the image by zooming and panning and to adjust the opacity of the labels.

This example uses two general strategies to label pixels in the highway image:

- First use the semi-automated tools, such as **Flood Fill** and **Smart Polygon**. Then, refine the labels using tools that offer more direct control, such as **Polygon** and **Brush**.
- First label distant objects with a rough estimation of object borders. Then, label nearer objects with more precise object borders.
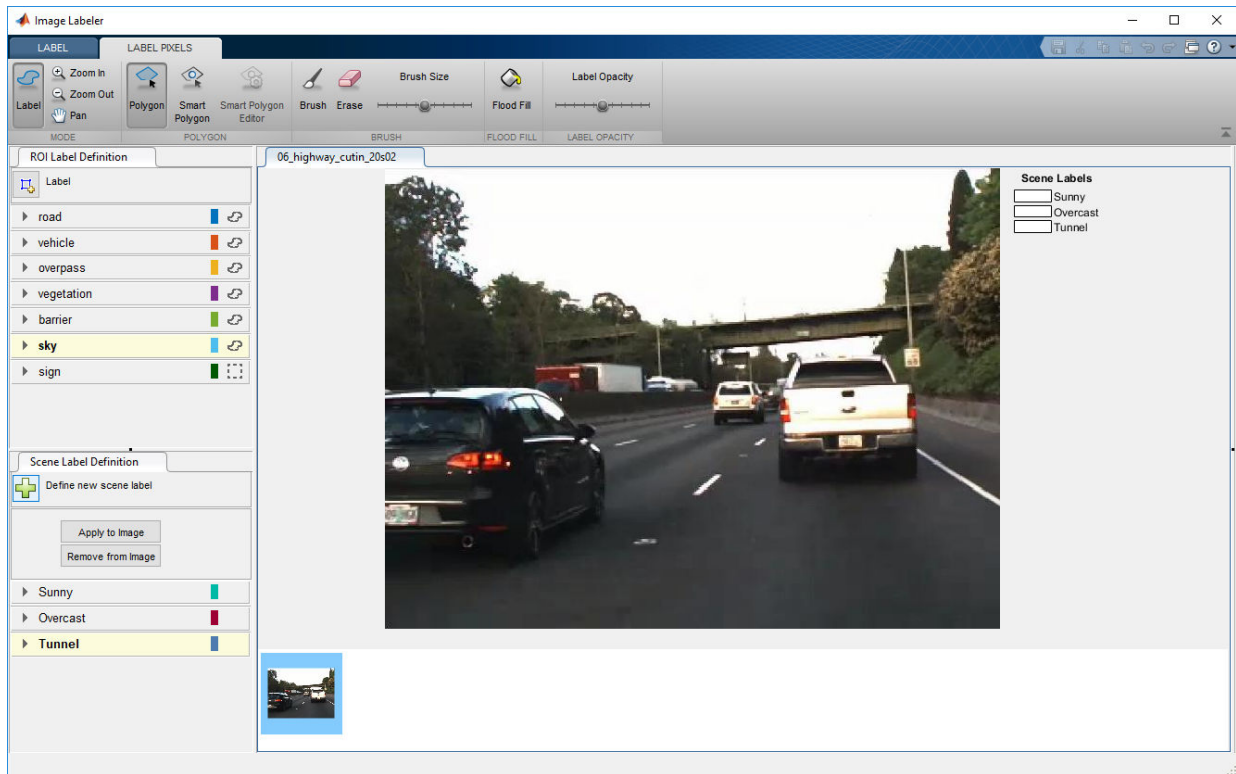
Label Pixels Using Flood Fill Tool

The **Flood Fill** tool labels a group of connected pixels that have a similar color. In this image, the sky is a good candidate for flood fill because the boundary of the bright sky is clear against the dark vegetation and overpass. In contrast, flood fill cannot isolate the vegetation because the color of the vegetation is too similar to the adjacent barriers, roads, and vehicles.

To label pixels using **Flood Fill**:

1   Select the tool and a label. The pointer changes to a paint can 🪣.

2   Click a starting pixel in the image.
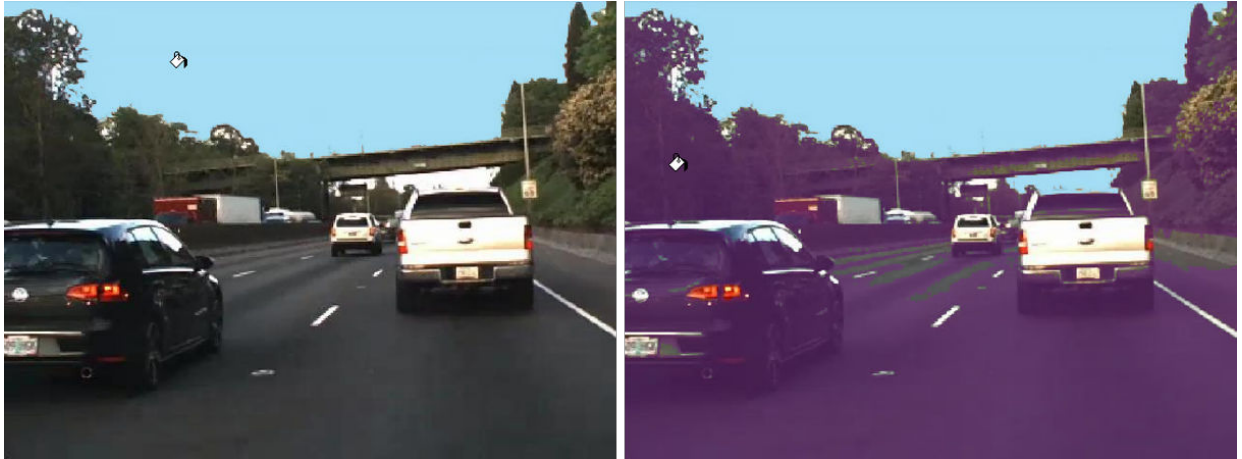
You can undo the flood fill, or any other labeling operation, by pressing **Ctrl+Z**.

Label Pixels Using Smart Polygon Tool

The **Smart Polygon** tool estimates the shape of an object of interest within a polygon that you draw. The tool is useful when the shape of the object is not a simple polygon. This example uses **Smart Polygon** to label the vegetation, which has a complicated boundary with the sky.

To label pixels using **Smart Polygon**:

1   Select the tool and a label. The pointer changes to a crosshair $+$.

2   Click to add polygon vertices. Completely surround the object of interest, with some space between the object and the polygon.

3   Close the polygon by clicking the first vertex after placing the other vertices. Alternatively, you can double-click to add the last vertex and close the polygon in one step.

   After you close the polygon, the tool draws an initial label.

4   Adjust the shape and position of the polygon. When the object of interest extends to the edge of the image, drag vertices to the edge of the image to ensure that the smart polygon completely encloses the object. For instance, this example shows the two leftmost vertices placed at the left edge of the image.
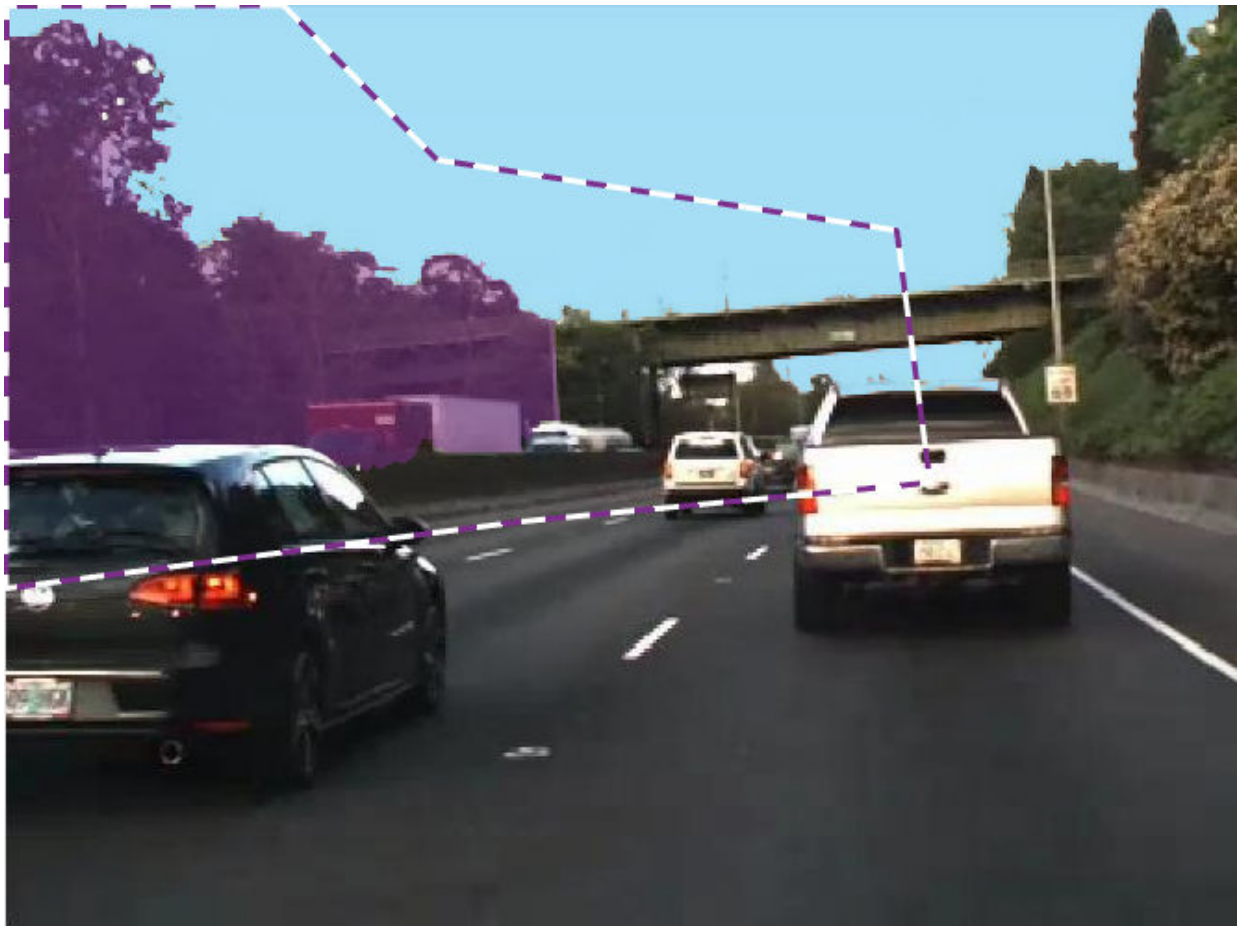
Smart Polygon Actions

Goal	Control
Move vertex	Click and drag the vertex.
Add vertex	• Right-click the polygon boundary at the position of the new vertex, and select **Add Point**. • Double-click the point on the boundary.
Delete vertex	Right-click the vertex and select **Delete Vertex**.
Move polygon	Click and drag any point on the polygon boundary (excluding vertices).
Delete polygon	Right-click the polygon boundary and select **Delete Polygon**.

5 Use the **Smart Polygon Editor** tools to refine the label.

- Select **Mark Foreground** to mark areas inside the region that you want to label. Foreground marks appear in green.

- Select **Mark Background** to mark areas inside the region that you do not want to label. Background marks appear in red.

- Select **Erase Marks** to remove foreground or background marks that are no longer needed.

- See Tips on page 7-52 for additional suggestions on using the **Smart Polygon** tool.

6 To finalize the label, press **Enter** or select a new **ROI Label Definition**. You can no longer edit the polygon vertices or mark foreground and background regions.

Label Pixels Using Polygon Tool

The **Polygon** tool labels all pixels within a polygon that you draw. The controls for defining and adjusting the vertices of a polygon are similar to the controls of the **Smart Polygon** tool.
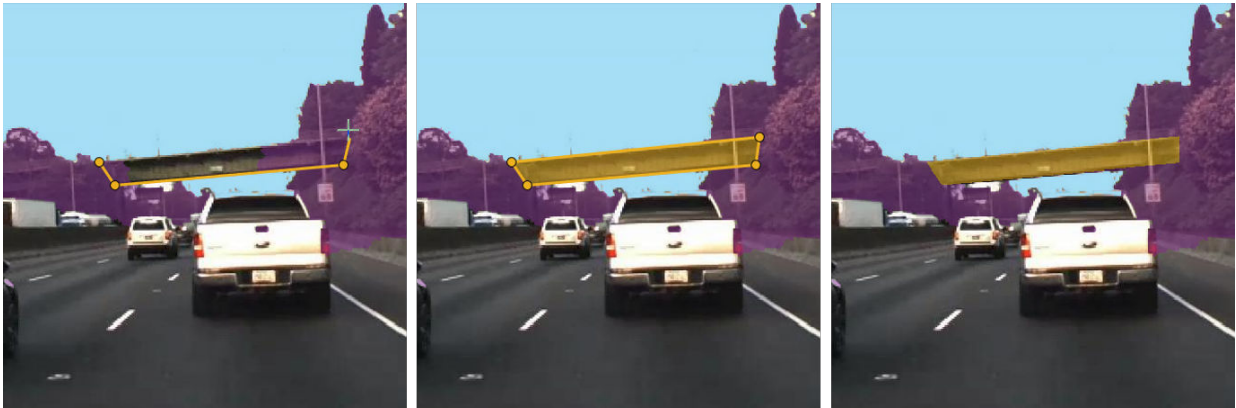
Add additional polygons over structures such as barriers and the road. Many vehicle pixels are incorrectly labeled. The next step shows how to replace the erroneous labels with the correct label.

Replace Pixel Labels

Each pixel can have at most one pixel label. When you apply a label to a pixel, the new label replaces the previous label.

This example uses the **Smart Polygon** tool to label pixels belonging to the truck. Foreground marks assign the *vehicle* label to subregions. Background marks revert subregions to their prior label. For instance, in the first pair of images, background marks revert subregions to the *sky* and *vegetation* labels. Similarly, in the second pair of images, background marks revert subregions to the *road* label.
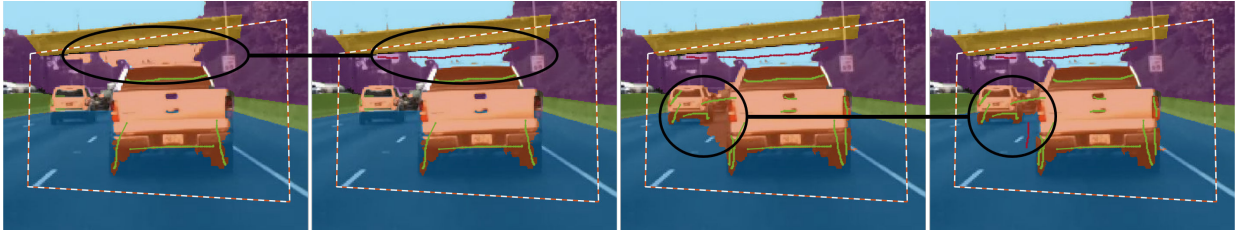
The border of the truck is jagged because **Smart Polygon** labels entire subregions, not individual pixels. The next step shows how to refine the labels along the border of the truck.

Refine Labels Using Brush Tool

The **Brush** tool labels pixels when you draw over the image with the mouse. This example uses **Brush** to remove spurs from the road and to make the edges of the truck smoother.

To label pixels using **Brush**:

1   Select the tool and a label. The pointer changes to a pen ✐ , and a square appears to indicate the size of the brush.

2   Adjust the size of the brush by using the **Brush Size** slider.

3   Click and drag the mouse to label pixels.

The **Erase** tool removes pixel labels when you draw over the image with the mouse.

Visualize Pixel Labels

You can modify the view of the image to facilitate pixel labeling. The **Zoom In**, **Zoom Out**, and **Pan** options enable you to zoom and pan the image with the mouse. To resume pixel labeling, click the **Label** icon.

The **Label Opacity** slider adjusts the opacity of all pixel labels.

- Decrease the opacity to see the image more clearly. For instance, decrease the opacity to make it easier to find the border between the bottom of the car and the road.

- Increase the opacity to see the segmentation more clearly. For instance, increase the opacity to see that edge along the front bumper of the car should be smoothed. Also, observe that the barrier and some distant vehicles have unlabeled pixels.
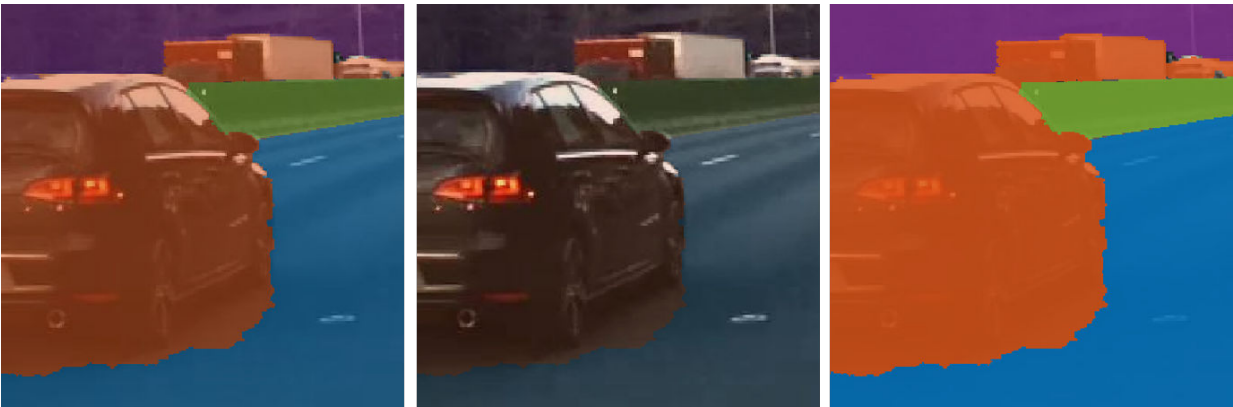
This is the final pixel-labeled image.
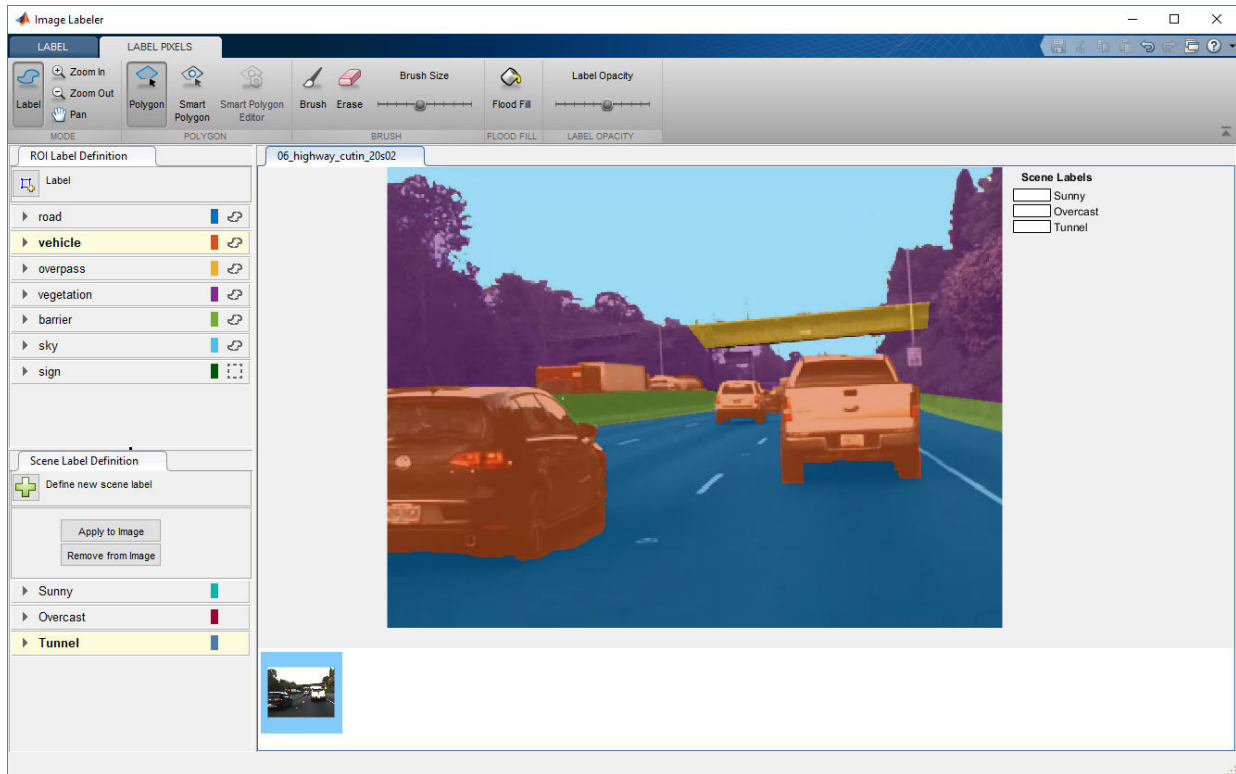
Tips

- The **Smart Polygon** tool identifies an object of interest by using regional graph-based segmentation ("GrabCut") [1]. The **Smart Polygon** tool divides the image into subregions. The tool treats all subregions that are fully or partially outside the polygon as belonging to the background. Therefore, to get an optimal segmentation, make sure the object to be labeled is fully contained within the polygon, surrounded by a few background pixels.

  All pixels within a subregion have the same label. Marking pixels outside the polygon has no effect on the label.

- To delete the most recently labeled ROI, press **Ctrl+Z**.

- Each pixel can have at most one pixel label. When you apply a label to a pixel, the new label replaces the previous label.

- Pixel labeling is disabled when you pan and zoom the image. You must click the **Label** button to resume pixel labeling.
- To ensure that all pixels in an image are labeled, begin by labeling the entire image with a single label. Pick a label that represents a predominant ROI in the image, such as *sky*, *road*, or *background*. Then, use the labeling tools to relabel objects with their correct label.

References

[1] Rother, C., V. Kolmogorov, and A. Blake. "GrabCut - Interactive Foreground Extraction using Iterated Graph Cuts". *ACM Transactions on Graphics (SIGGRAPH)*. Vol. 23, Number 3, 2004, pp. 309–314.

See Also

Image Labeler

More About

- "Define Ground Truth for Image Collections" on page 7-54

Define Ground Truth for Image Collections

The **Image Labeler** app labels rectangular regions of interest (ROIs) for object detection, pixels for semantic segmentation, and scenes for image classification.
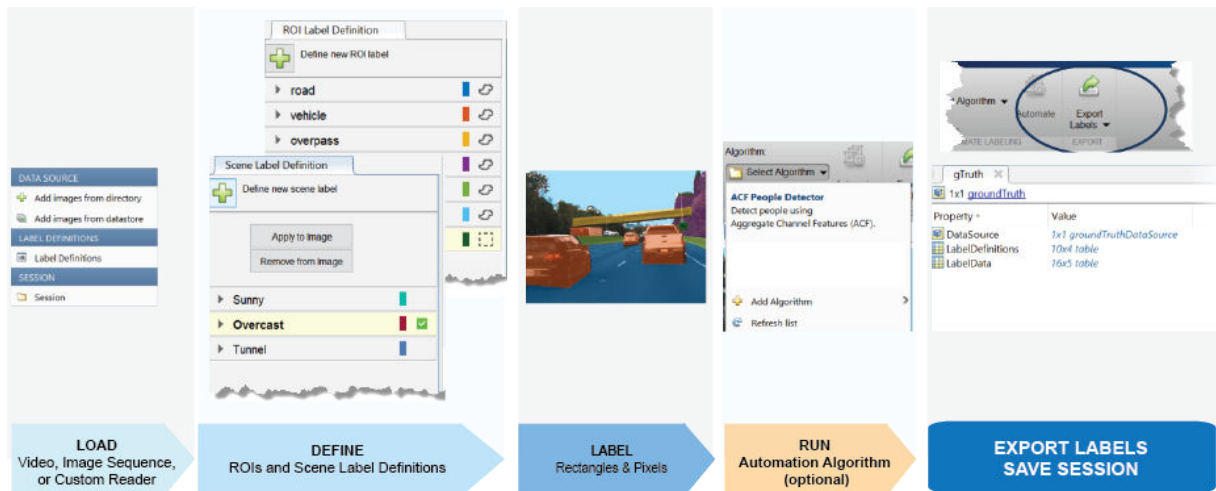
ROI and Scene Label Definitions

- An ROI label corresponds to either a rectangular or pixel region of interest. These labels contain two components: the label name, such as "cars," and the region you create.

- A Scene label describes the nature of a scene, such as "sunny." You can associate this label with a frame.

Using the **Image Labeler** app, you can:

- Interactively specify ROI labels and scene labels.

  - Use rectangular ROI labels for objects such as vehicles, pedestrians, and road signs.

  - Use pixel labels for areas such as backgrounds, roads, and buildings.

  - Use scene labels for conditions such as lighting and weather conditions, or for events such as lane changes.

- Use built-in detection or tracking to automatically label the regions and scene labels.

- Write, import, and use your own custom automation algorithm to automatically label a region and scene labels.
- Export the ground truth labels for object detector training, semantic segmentation, or image classification.

Open the Image Labeler

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click the **Image Labeler**.
- MATLAB command prompt: Enter `imageLabeler`.

Load a Video or Image Sequence and Import Labels

To load data into the **Image Labeler**, from the app toolstrip, click **Load**. You can load the following data:

- **Data Source**: Add images from a folder or by using the `datastore` function.
- **Label Definitions**: Load a previously saved set of label definitions from a file. Label definitions specify the names and types of items to label.
- **Session**: Load a previously saved session.

To import ROI and scene labels into the app, click **Import Labels**. You can import labels from the MATLAB workspace or from previously exported MAT-files. The imported labels must be `groundTruth` objects.

Define ROIs and Scene Label Definitions

Before you can label your images, you must define the name and type of each label category. To define an ROI label, click the **Define new ROI label**, specify a name to represent the label, and then choose either **Rectangle** or **Pixel label** for the label type. To define a scene label, specify a descriptive name and optionally enter a description.
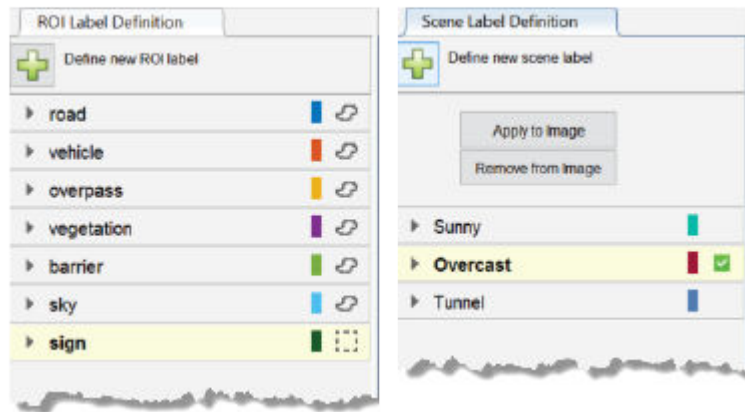
In addition, you can enter descriptions for ROI and scene labels that can be used as instructions for labeling.
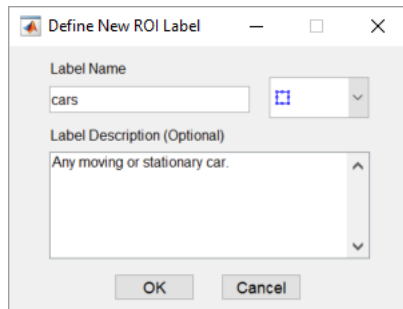
Label Ground Truth

After you set up the ROI label definitions, you can start labeling. You can create labels manually or use an automation algorithm.

Create Labels Manually

- To draw ROI labels manually, select an ROI label definition from the left pane and use the mouse to draw the regions on the image frames.
- To label individual pixels, see "Label Pixels for Semantic Segmentation" on page 7-43.
- To mark scene labels manually, select a scene label definition from the left pane and then click **Add Label**.

Create Labels Using an Automation Algorithm

Use the **Select Algorithm** section to select an algorithm for automated labeling. You can use a built-in algorithm, create a custom algorithm, or import an algorithm.

- **Built-In Algorithm**: Track people using the aggregated channel features (ACF) people detector algorithm.

- **Add a Custom Algorithm:** To define and use a custom automation algorithm with the **Image Labeler** app, see "Create Automation Algorithm for Image Labeling" on page 7-39.

- **Import an Algorithm**: To import your own algorithm, select **Algorithm > Add Algorithm > Import Algorithm**.

Run an Automation Algorithm

1 Click **Automate**. Only ROI and scene label definitions that are valid for the selected algorithm are used. Valid label definitions are enabled in the left pane and algorithm instructions appear in the right pane.
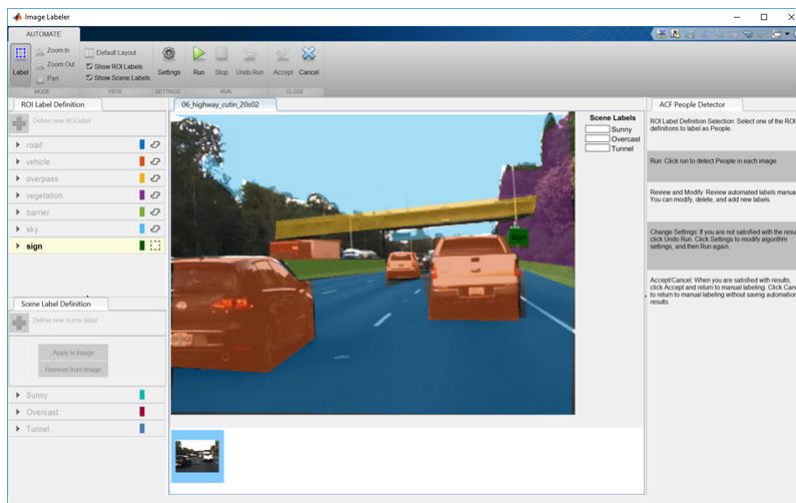
2 Click **Run**.

3 Examine the results of running the algorithm. If they are not satisfactory, click **Undo Run** and change algorithm settings by clicking **Settings**.

4   When you are satisfied with the algorithm results, click **Accept**. To delete the labels generated during the automation session, click **Cancel**. The **Cancel** button cancels only the algorithm session, not the app session.

Export Labels and Save Session

To export the ground truth labels to the MATLAB workspace or to a MAT-file, click **Export Labels**. The labels are exported as a `groundTruth` object. Click **Save** to save the session. The session and the exported labels are saved as MAT-files. You can use the exported `groundTruth` object to train an object detector. See "Train an Object Detector from Ground Truth Data" (Automated Driving System Toolbox).

Note Pixel label data and ground truth data are saved in separate files. The app saves both files in the same folder. Keep these tips in mind:

- Once label matrices are created, do not move them to a different folder. The `groundTruth` object contains the locations of pixel data for the label matrices. These locations cannot be modified.

- If you used an image collection to create your ground truth, do not delete images from the location you loaded them from. The path to those images is saved in the `groundTruth` object and cannot be modified.

- You can move the `groundTruth` MAT-file to a different folder.

See Also

Apps
Image Labeler

Objects
`groundTruth` | `groundTruthDataSource`

More About

- "Create Automation Algorithm for Image Labeling" on page 7-39
- "Train an Object Detector from Ground Truth Data" on page 7-37

Choose Among R-CNN, Fast R-CNN, or Faster R-CNN Object Detectors
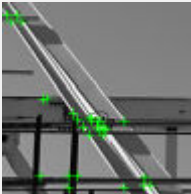
This family of object detectors uses region proposal to detect objects within images. The number of proposed regions dictates the time it takes to detect objects in an image. The `trainFasterRCNNObjectDetector` and the `trainFastRCNNObjectDetector` are designed to improve detection performance with a large number of regions.
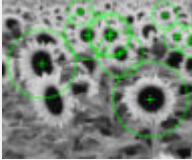
R-CNN Detector	Description
`trainFasterRCNNObjectDetector`	• Optimal runtime performance • Does not support a custom region proposal
`trainFastRCNNObjectDetector`	• Allows custom region proposal
`trainRCNNObjectDetector`	• Less time to train an object detector, but detection time is slow. • Allows custom region proposal

Point Feature Types

Image feature detection is a building block of many computer vision tasks, such as image registration, tracking, and object detection. The Computer Vision System Toolbox includes a variety of functions for image feature detection. These functions return points objects that store information specific to particular types of features, including (x,y) coordinates (in the `Location` property). You can pass a points object from a detection function to a variety of other functions that require feature points as inputs. The algorithm that a detection function uses determines the type of points object it returns.

Functions That Return Points Objects

Points Object	Returned By	Type of Feature
cornerPoints	detectFASTFeatures Features from accelerated segment test (FAST) algorithm Uses an approximate metric to determine corners. [1]	 Corners Single-scale detection Point tracking, image registration with little or no scale change, corner detection in scenes of human origin, such as streets and indoor scenes.
	detectMinEigenFeatures Minimum eigenvalue algorithm Uses minimum eigenvalue metric to determine corner locations. [4]	
	detectHarrisFeatures Harris-Stephens algorithm More efficient than the minimum eigenvalue algorithm. [3]	

Points Object	Returned By	Type of Feature
BRISKPoints	detectBRISKFeatures Binary Robust Invariant Scalable Keypoints (BRISK) algorithm [6]	 Corners Multiscale detection Point tracking, image registration, handles changes in scale and rotation, corner detection in scenes of human origin, such as streets and indoor scenes
SURFPoints	detectSURFFeatures Speeded-up robust features (SURF) algorithm [11]	 Blobs Multiscale detection Object detection and image registration with scale and rotation changes
KAZEPoints	detectKAZEFeatures KAZE is not an acronym, but a name derived from the Japanese word *kaze*, which means wind. The reference is to the flow of air ruled by nonlinear processes on a large scale. [12]	 Multi-scale blob features Reduced blurring of object boundaries

Points Object	Returned By	Type of Feature
MSERRegions	detectMSERFeatures Maximally stable extremal regions (MSER) algorithm [7] [8] [9] [10]	 Regions of uniform intensity Multi-scale detection Registration, wide baseline stereo calibration, text detection, object detection. Handles changes to scale and rotation. More robust to affine transforms in contrast to other detectors.

Functions That Accept Points Objects

Function	Description		
relativeCameraPose	Compute relative rotation and translation between camera poses		
estimateFundamentalMatrix	Estimate fundamental matrix from corresponding points in stereo images		
estimateGeometricTransform	Estimate geometric transform from matching point pairs		
estimateUncalibratedRectification	Uncalibrated stereo rectification		
extractFeatures	Extract interest point descriptors		
		Method	**Feature Vector**
		BRISK	The function sets the Orientation property of the validPoints output object to the orientation of the extracted features, in radians.

Function	Description	
	FREAK	The function sets the `Orientation` property of the `validPoints` output object to the orientation of the extracted features, in radians.
	SURF	The function sets the `Orientation` property of the `validPoints` output object to the orientation of the extracted features, in radians. When you use an `MSERRegions` object with the `SURF` method, the `Centroid` property of the object extracts SURF descriptors. The `Axes` property of the object selects the scale of the SURF descriptors such that the circle representing the feature has an area proportional to the MSER ellipse area. The scale is calculated as `1/4*sqrt((majorAxes/2).* (minorAxes/2))` and saturated to `1.6`, as required by the `SURFPoints` object.

Function	Description	
	KAZE	Non-linear pyramid-based features. The function sets the `Orientation` property of the `validPoints` output object to the orientation of the extracted features, in radians. When you use an `MSERRegions` object with the `KAZE` method, the `Location` property of the object is used to extract KAZE descriptors. The `Axes` property of the object selects the scale of the KAZE descriptors such that the circle representing the feature has an area proportional to the MSER ellipse area.
	Block	Simple square neighbhorhood. The `Block` method extracts only the neighborhoods fully contained within the image boundary. Therefore, the output, `validPoints`, can contain fewer points than the input `POINTS`.
	Auto	The function selects the `Method` based on the class of the input points and implements: The `FREAK` method for a `cornerPoints` input object. The `SURF` method for a `SURFPoints` or `MSERRegions` input object. The `FREAK` method for a `BRISKPoints` input object. For an *M*-by-2 input matrix of [*x y*] coordinates, the function implements the `Block` method.

Function	Description
extractHOGFeatures	Extract histogram of oriented gradients (HOG) features
insertMarker	Insert markers in image or video
showMatchedFeatures	Display corresponding feature points
triangulate	3-D locations of undistorted matching points in stereo images
undistortPoints	Correct point coordinates for lens distortion

References

[1] Rosten, E., and T. Drummond, "Machine Learning for High-Speed Corner Detection." *9th European Conference on Computer Vision*. Vol. 1, 2006, pp. 430–443.

[2] Mikolajczyk, K., and C. Schmid. "A performance evaluation of local descriptors." *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Vol. 27, Issue 10, 2005, pp. 1615–1630.

[3] Harris, C., and M. J. Stephens. "A Combined Corner and Edge Detector." *Proceedings of the 4th Alvey Vision Conference*. August 1988, pp. 147–152.

[4] Shi, J., and C. Tomasi. "Good Features to Track." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. June 1994, pp. 593–600.

[5] Tuytelaars, T., and K. Mikolajczyk. "Local Invariant Feature Detectors: A Survey." *Foundations and Trends in Computer Graphics and Vision*. Vol. 3, Issue 3, 2007, pp. 177–280.

[6] Leutenegger, S., M. Chli, and R. Siegwart. "BRISK: Binary Robust Invariant Scalable Keypoints." *Proceedings of the IEEE International Conference*. ICCV, 2011.

[7] Nister, D., and H. Stewenius. "Linear Time Maximally Stable Extremal Regions." *Lecture Notes in Computer Science. 10th European Conference on Computer Vision*. Marseille, France: 2008, no. 5303, pp. 183–196.

[8] Matas, J., O. Chum, M. Urba, and T. Pajdla. "Robust wide-baseline stereo from maximally stable extremal regions." *Proceedings of British Machine Vision Conference*. 2002, pp. 384–396.

[9] Obdrzalek D., S. Basovnik, L. Mach, and A. Mikulik. "Detecting Scene Elements Using Maximally Stable Colour Regions." *Communications in Computer and Information Science*. La Ferte-Bernard, France: 2009, Vol. 82 CCIS (2010 12 01), pp 107–115.

[10] Mikolajczyk, K., T. Tuytelaars, C. Schmid, A. Zisserman, T. Kadir, and L. Van Gool. "A Comparison of Affine Region Detectors." *International Journal of Computer Vision*. Vol. 65, No. 1–2, November, 2005, pp. 43–72 .

[11] Bay, H., A. Ess, T. Tuytelaars, and L. Van Gool. "SURF:Speeded Up Robust Features." *Computer Vision and Image Understanding (CVIU)*.Vol. 110, No. 3, 2008, pp. 346–359.

[12] Alcantarilla, P.F., A. Bartoli, and A.J. Davison. "KAZE Features", *ECCV 2012, Part VI, LNCS 7577* pp. 214, 2012

See Also

Related Examples

- "Detect BRISK Points in an Image and Mark Their Locations"
- "Find Corner Points in an Image Using the FAST Algorithm"
- "Find Corner Points Using the Harris-Stephens Algorithm"
- "Find Corner Points Using the Eigenvalue Algorithm"
- "Find MSER Regions in an Image"
- "Detect SURF Interest Points in a Grayscale Image"
- "Automatically Detect and Recognize Text in Natural Images"
- "Object Detection in a Cluttered Scene Using Point Feature Matching"

Local Feature Detection and Extraction

Local features and their descriptors, which are a compact vector representations of a local neighborhood, are the building blocks of many computer vision algorithms. Their applications include image registration, object detection and classification, tracking, and motion estimation. Using local features enables these algorithms to better handle scale changes, rotation, and occlusion. The Computer Vision System Toolbox provides the FAST, Harris, and Shi & Tomasi methods for detecting corner features, and the SURF, KAZE, and MSER methods for detecting blob features. The toolbox includes the SURF, KAZE, FREAK, BRISK, and HOG descriptors. You can mix and match the detectors and the descriptors depending on the requirements of your application.

What Are Local Features?

Local features refer to a pattern or distinct structure found in an image, such as a point, edge, or small image patch. They are usually associated with an image patch that differs from its immediate surroundings by texture, color, or intensity. What the feature actually represents does not matter, just that it is distinct from its surroundings. Examples of local features are blobs, corners, and edge pixels.

Example 7.1. Example of Corner Detection

```
I = imread('circuit.tif');
corners = detectFASTFeatures(I,'MinContrast',0.1);
J = insertMarker(I,corners,'circle');
imshow(J)
```
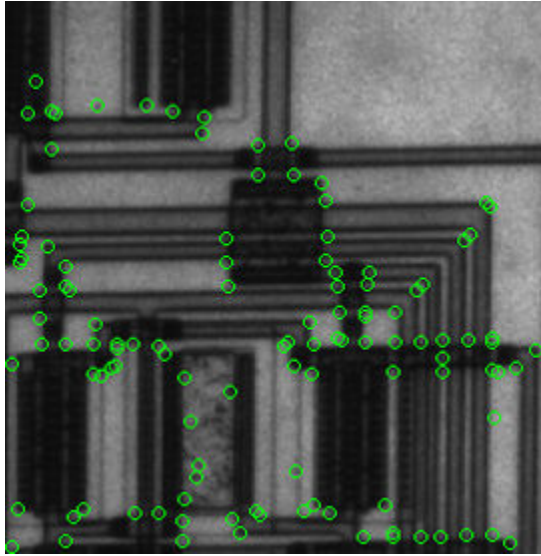
Benefits and Applications of Local Features

Local features let you find image correspondences regardless of occlusion, changes in viewing conditions, or the presence of clutter. In addition, the properties of local features make them suitable for image classification, such as in "Image Classification with Bag of Visual Words" on page 7-120.

Local features are used in two fundamental ways:

- To localize anchor points for use in image stitching or 3-D reconstruction.
- To represent image contents compactly for detection or classification, without requiring image segmentation.

Application	MATLAB Examples
Image registration and stitching	"Feature Based Panoramic Image Stitching"
Object detection	"Object Detection in a Cluttered Scene Using Point Feature Matching"
Object recognition	"Digit Classification Using HOG Features"

Application	MATLAB Examples
Object tracking	"Face Detection and Tracking Using the KLT Algorithm"
Image category recognition	"Image Category Classification Using Bag of Features"
Finding geometry of a stereo system	"Uncalibrated Stereo Image Rectification"
3-D reconstruction	"Structure From Motion From Two Views" "Structure From Motion From Multiple Views"
Image retrieval	"Image Retrieval Using Customized Bag of Features"

What Makes a Good Local Feature?

Detectors that rely on gradient-based and intensity variation approaches detect good local features. These features include edges, blobs, and regions. Good local features exhibit the following properties:

- **Repeatable detections**:
  When given two images of the same scene, most features that the detector finds in both images are the same. The features are robust to changes in viewing conditions and noise.
- **Distinctive**:
  The neighborhood around the feature center varies enough to allow for a reliable comparison between the features.
- **Localizable**:
  The feature has a unique location assigned to it. Changes in viewing conditions do not affect its location.

Feature Detection and Feature Extraction

Feature detection selects regions of an image that have unique content, such as corners or blobs. Use feature detection to find points of interest that you can use for further processing. These points do not necessarily correspond to physical structures, such as the corners of a table. The key to feature detection is to find features that remain locally invariant so that you can detect them even in the presence of rotation or scale change.

Feature extraction involves computing a descriptor, which is typically done on regions centered around detected features. Descriptors rely on image processing to transform a

local pixel neighborhood into a compact vector representation. This new representation permits comparison between neighborhoods regardless of changes in scale or orientation. Descriptors, such as SIFT or SURF, rely on local gradient computations. Binary descriptors, such as BRISK or FREAK, rely on pairs of local intensity differences, which are then encoded into a binary vector.

Choose a Feature Detector and Descriptor

Select the best feature detector and descriptor by considering the criteria of your application and the nature of your data. The first table helps you understand the general criteria to drive your selection. The next two tables provide details on the detectors and descriptors available in Computer Vision System Toolbox.

Considerations for Selecting a Detector and Descriptor

Criteria	Suggestion
Type of features in your image	Use a detector appropriate for your data. For example, if your image contains an image of bacteria cells, use the blob detector rather than the corner detector. If your image is an aerial view of a city, you can use the corner detector to find man-made structures.
Context in which you are using the features: • Matching key points • Classification	The HOG, SURF, and KAZE descriptors are suitable for classification tasks. In contrast, binary descriptors, such as BRISK and FREAK, are typically used for finding point correspondences between images, which are used for registration.
Type of distortion present in your image	Choose a detector and descriptor that addresses the distortion in your data. For example, if there is no scale change present, consider a corner detector that does not handle scale. If your data contains a higher level of distortion, such as scale and rotation, then use the more computationally intensive SURF or KAZE feature detector and descriptor.
Performance requirements: • Real-time performance required • Accuracy versus speed	Binary descriptors are generally faster but less accurate than gradient-based descriptors. For greater accuracy, use several detectors and descriptors at the same time.

Choose a Detection Function Based on Feature Type

Detector	Feature Type	Function	Scale Independent
FAST [1]	Corner	`detectFASTFeatures`	No
Minimum eigenvalue algorithm[4]	Corner	`detectMinEigenFeatures`	No
Corner detector [3]	Corner	`detectHarrisFeatures`	No
SURF [11]	Blob	`detectSURFFeatures`	Yes
KAZE[12]	Blob	`detectKAZEFeatures`	Yes
BRISK [6]	Corner	`detectBRISKFeatures`	Yes
MSER [8]	Region with uniform intensity	`detectMSERFeatures`	Yes

Note Detection functions return objects that contain information about the features. The `extractHOGFeatures` and `extractFeatures` functions use these objects to create descriptors.

Choose a Descriptor Method

Descriptor	Binary	Function and Method	Invariance		Typical Use	
			Scale	Rotation	Finding Point Correspondences	Classification
HOG	No	`extractHOGFeatures(I, …)`	No	No	No	Yes
LBP	No	`extractLBPFeatures(I, …)`	No	Yes	No	Yes
SURF	No	`extractFeatures(I, 'Method','SURF')`	Yes	Yes	Yes	Yes
KAZE	No	`extractFeatures(I,'Method','KAZE')`	Yes	Yes	Yes	Yes
FREAK	Yes	`extractFeatures(I, 'Method','FREAK')`	Yes	Yes	Yes	No
BRISK	Yes	`extractFeatures(I, 'Method','BRISK')`	Yes	Yes	Yes	No
• Block • Simple pixel neighborhood around a keypoint	No	`extractFeatures(I, 'Method','Block')`	No	No	Yes	Yes

Note

- The `extractFeatures` function provides different extraction methods to best match the requirements of your application. When you do not specify the `'Method'` input for the `extractFeatures` function, the function automatically selects the method based on the type of input point class.

- Binary descriptors are fast but less precise in terms of localization. They are not suitable for classification tasks. The `extractFeatures` function returns a `binaryFeatures` object. This object enables the Hamming-distance-based matching metric used in the `matchFeatures` function.

Use Local Features

Registering two images is a simple way to understand local features. This example finds a geometric transformation between two images. It uses local features to find well-localized anchor points.

Display two images.

The first image is the original image.

```
original = imread('cameraman.tif');
figure;
imshow(original);
```


The second image, is the original image rotated and scaled.

```
scale = 1.3;
J = imresize(original,scale);
theta = 31;
```

```
distorted = imrotate(J,theta);
figure
imshow(distorted)
```

Detect matching features between the original and distorted image.

Detecting the matching SURF features is the first step in determining the transform needed to correct the distorted image.

```
ptsOriginal  = detectSURFFeatures(original);
ptsDistorted = detectSURFFeatures(distorted);
```

Extract features and compare the detected blobs between the two images.

The detection step found several roughly corresponding blob structures in both images. Compare the detected blob features. This process is facilitated by feature extraction, which determines a local patch descriptor.

```
[featuresOriginal,validPtsOriginal] = ...
            extractFeatures(original,ptsOriginal);
[featuresDistorted,validPtsDistorted] = ...
            extractFeatures(distorted,ptsDistorted);
```

It is possible that not all of the original points were used to extract descriptors. Points might have been rejected if they were too close to the image border. Therefore, the valid points are returned in addition to the feature descriptors.

The patch size used to compute the descriptors is determined during the feature extraction step. The patch size corresponds to the scale at which the feature is detected. Regardless of the patch size, the two feature vectors, `featuresOriginal` and `featuresDistorted`, are computed in such a way that they are of equal length. The descriptors enable you to compare detected features, regardless of their size and rotation.

Find candidate matches.

Obtain candidate matches between the features by inputting the descriptors to the `matchFeatures` function. Candidate matches imply that the results can contain some invalid matches. Two patches that match can indicate like features but might not be a correct match. A table corner can look like a chair corner, but the two features are obviously not a match.

```
indexPairs = matchFeatures(featuresOriginal,featuresDistorted);
```

Find point locations from both images.

Each row of the returned `indexPairs` contains two indices of candidate feature matches between the images. Use the indices to collect the actual point locations from both images.

```
matchedOriginal  = validPtsOriginal(indexPairs(:,1));
matchedDistorted = validPtsDistorted(indexPairs(:,2));
```

Display the candidate matches.

```
figure
showMatchedFeatures(original,distorted,matchedOriginal,matchedDistorted)
title('Candidate matched points (including outliers)')
```

Candidate matched points (including outliers)

Analyze the feature locations.

If there are a sufficient number of valid matches, remove the false matches. An effective technique for this scenario is the RANSAC algorithm. The `estimateGeometricTransform` function implements M-estimator sample consensus

(MSAC), which is a variant of the RANSAC algorithm. MSAC finds a geometric transform and separates the inliers (correct matches) from the outliers (spurious matches).

```
[tform, inlierDistorted,inlierOriginal] = ...
          estimateGeometricTransform(matchedDistorted,...
              matchedOriginal,'similarity');
```

Display the matching points.

```
figure
showMatchedFeatures(original,distorted,inlierOriginal,inlierDistorted)
title('Matching points (inliers only)')
legend('ptsOriginal','ptsDistorted')
```

Matching points (inliers only)

Verify the computed geometric transform.

Apply the computed geometric transform to the distorted image.

```
outputView = imref2d(size(original));
recovered  = imwarp(distorted,tform,'OutputView',outputView);
```

Display the recovered image and the original image.

```
figure
imshowpair(original,recovered,'montage')
```


Image Registration Using Multiple Features

This example builds on the results of the "Use Local Features" example. Using more than one detector and descriptor pair enables you to combine and reinforce your results. Multiple pairs are also useful for when you cannot obtain enough good matches (inliers) using a single feature detector.

Load the original image.

```
original = imread('cameraman.tif');
figure;
imshow(original);
text(size(original,2),size(original,1)+15, ...
    'Image courtesy of Massachusetts Institute of Technology', ...
    'FontSize',7,'HorizontalAlignment','right');
```

Image courtesy of Massachusetts Institute of Technology

Scale and rotate the original image to create the distorted image.

```
scale = 1.3;
J = imresize(original, scale);

theta = 31;
distorted = imrotate(J,theta);
figure
imshow(distorted)
```

Detect the features in both images. Use the BRISK detectors first, followed by the SURF detectors.

```
ptsOriginalBRISK  = detectBRISKFeatures(original,'MinContrast',0.01);
ptsDistortedBRISK = detectBRISKFeatures(distorted,'MinContrast',0.01);
```

```
ptsOriginalSURF  = detectSURFFeatures(original);
ptsDistortedSURF = detectSURFFeatures(distorted);
```

Extract descriptors from the original and distorted images. The BRISK features use the FREAK descriptor by default.

```
[featuresOriginalFREAK,validPtsOriginalBRISK]  = ...
        extractFeatures(original,ptsOriginalBRISK);
[featuresDistortedFREAK,validPtsDistortedBRISK] = ...
        extractFeatures(distorted,ptsDistortedBRISK);

[featuresOriginalSURF,validPtsOriginalSURF]  = ...
        extractFeatures(original,ptsOriginalSURF);
[featuresDistortedSURF,validPtsDistortedSURF] = ...
        extractFeatures(distorted,ptsDistortedSURF);
```

Determine candidate matches by matching FREAK descriptors first, and then SURF descriptors. To obtain as many feature matches as possible, start with detector and matching thresholds that are lower than the default values. Once you get a working solution, you can gradually increase the thresholds to reduce the computational load required to extract and match features.

```
indexPairsBRISK = matchFeatures(featuresOriginalFREAK,...
            featuresDistortedFREAK,'MatchThreshold',40,'MaxRatio',0.8);

indexPairsSURF = matchFeatures(featuresOriginalSURF,featuresDistortedSURF);
```

Obtain candidate matched points for BRISK and SURF.

```
matchedOriginalBRISK  = validPtsOriginalBRISK(indexPairsBRISK(:,1));
matchedDistortedBRISK = validPtsDistortedBRISK(indexPairsBRISK(:,2));

matchedOriginalSURF  = validPtsOriginalSURF(indexPairsSURF(:,1));
matchedDistortedSURF = validPtsDistortedSURF(indexPairsSURF(:,2));
```

Visualize the BRISK putative matches.

```
figure
showMatchedFeatures(original,distorted,matchedOriginalBRISK,...
            matchedDistortedBRISK)
title('Putative matches using BRISK & FREAK')
legend('ptsOriginalBRISK','ptsDistortedBRISK')
```

Putative matches using BRISK & FREAK
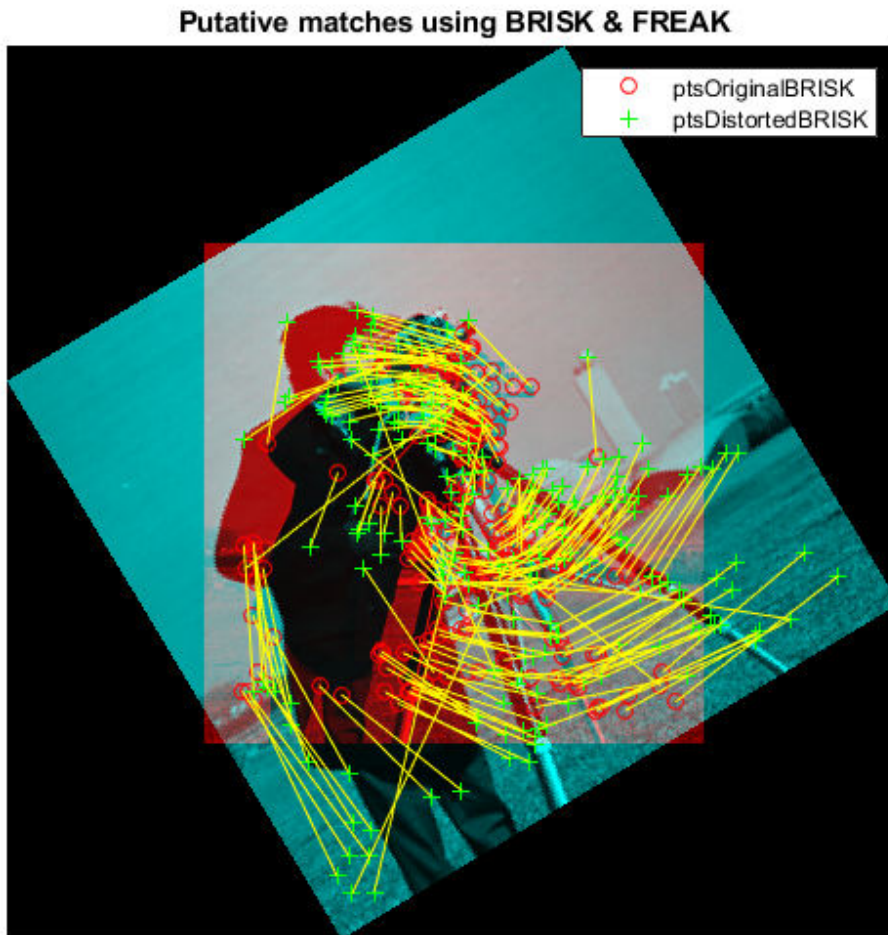
Combine the candidate matched BRISK and SURF local features. Use the `Location` property to combine the point locations from BRISK and SURF features.

```
matchedOriginalXY  = ...
    [matchedOriginalSURF.Location; matchedOriginalBRISK.Location];
```

```
matchedDistortedXY = ...
    [matchedDistortedSURF.Location; matchedDistortedBRISK.Location];
```

Determine the inlier points and the geometric transform of the BRISK and SURF features.

```
[tformTotal,inlierDistortedXY,inlierOriginalXY] = ...
    estimateGeometricTransform(matchedDistortedXY,...
        matchedOriginalXY,'similarity');
```

Display the results. The result provides several more matches than the example that used a single feature detector.

```
figure
showMatchedFeatures(original,distorted,inlierOriginalXY,inlierDistortedXY)
title('Matching points using SURF and BRISK (inliers only)')
legend('ptsOriginal','ptsDistorted')
```

Matching points using SURF and BRISK (inliers only)
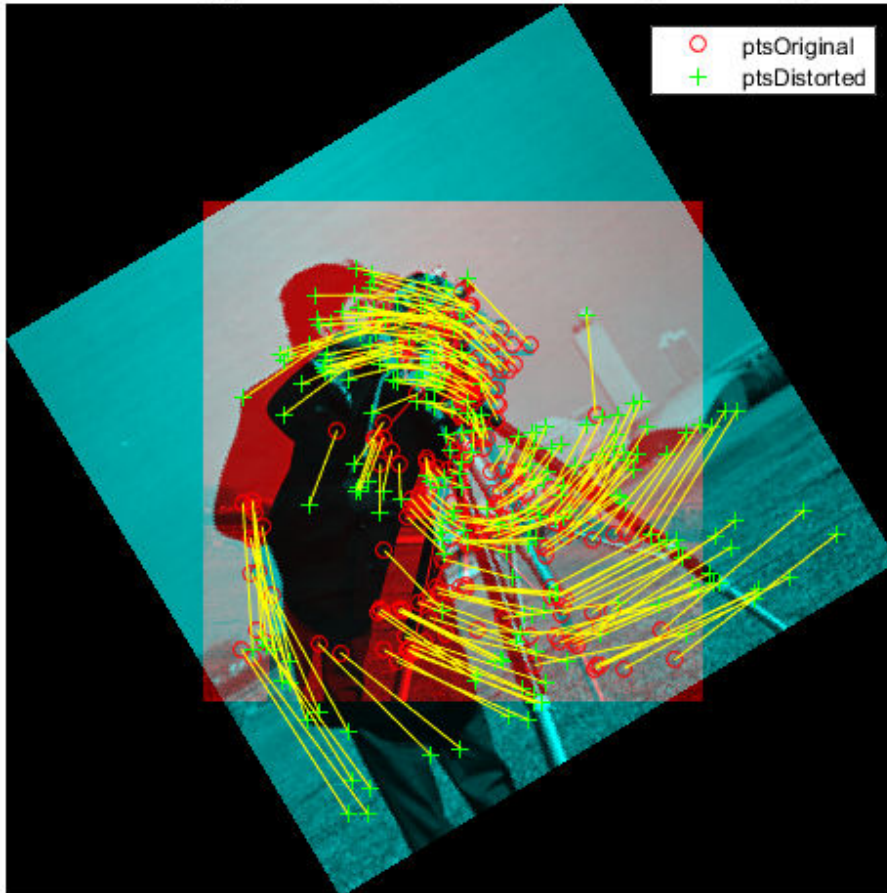
Compare the original and recovered image.

```
outputView = imref2d(size(original));
recovered  = imwarp(distorted,tformTotal,'OutputView',outputView);
```

```
figure;
imshowpair(original,recovered,'montage')
```


References

[1] Rosten, E., and T. Drummond. "Machine Learning for High-Speed Corner Detection." *9th European Conference on Computer Vision*. Vol. 1, 2006, pp. 430–443.

[2] Mikolajczyk, K., and C. Schmid. "A performance evaluation of local descriptors." *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Vol. 27, Issue 10, 2005, pp. 1615–1630.

[3] Harris, C., and M. J. Stephens. "A Combined Corner and Edge Detector." *Proceedings of the 4th Alvey Vision Conference*. August 1988, pp. 147–152.

[4] Shi, J., and C. Tomasi. "Good Features to Track." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. June 1994, pp. 593–600.

[5] Tuytelaars, T., and K. Mikolajczyk. "Local Invariant Feature Detectors: A Survey." *Foundations and Trends in Computer Graphics and Vision*. Vol. 3, Issue 3, 2007, pp. 177–280.

[6] Leutenegger, S., M. Chli, and R. Siegwart. "BRISK: Binary Robust Invariant Scalable Keypoints." *Proceedings of the IEEE International Conference*. ICCV, 2011.

[7] Nister, D., and H. Stewenius. "Linear Time Maximally Stable Extremal Regions." *10th European Conference on Computer Vision*. Marseille, France: 2008, No. 5303, pp. 183–196.

[8] Matas, J., O. Chum, M. Urba, and T. Pajdla. "Robust wide-baseline stereo from maximally stable extremal regions."*Proceedings of British Machine Vision Conference*. 2002, pp. 384–396.

[9] Obdrzalek D., S. Basovnik, L. Mach, and A. Mikulik. "Detecting Scene Elements Using Maximally Stable Colour Regions."*Communications in Computer and Information Science*. La Ferte-Bernard, France: 2009, Vol. 82 CCIS (2010 12 01), pp. 107–115.

[10] Mikolajczyk, K., T. Tuytelaars, C. Schmid, A. Zisserman, T. Kadir, and L. Van Gool. "A Comparison of Affine Region Detectors. "*International Journal of Computer Vision*. Vol. 65, No. 1–2, November 2005, pp. 43–72 .

[11] Bay, H., A. Ess, T. Tuytelaars, and L. Van Gool. "SURF: Speeded Up Robust Features." *Computer Vision and Image Understanding (CVIU)*. Vol. 110, No. 3, 2008, pp. 346–359.

[12] Alcantarilla, P.F., A. Bartoli, and A.J. Davison. "KAZE Features", *ECCV 2012, Part VI, LNCS 7577* pp. 214, 2012

See Also

Related Examples

- "Detect BRISK Points in an Image and Mark Their Locations"
- "Find Corner Points in an Image Using the FAST Algorithm"
- "Find Corner Points Using the Harris-Stephens Algorithm"
- "Find Corner Points Using the Eigenvalue Algorithm"

- "Find MSER Regions in an Image"
- "Detect SURF Interest Points in a Grayscale Image"
- "Automatically Detect and Recognize Text in Natural Images"
- "Object Detection in a Cluttered Scene Using Point Feature Matching"

Train a Cascade Object Detector

In this section...

Why Train a Detector?

The `vision.CascadeObjectDetector` System object comes with several pretrained classifiers for detecting frontal faces, profile faces, noses, eyes, and the upper body. However, these classifiers are not always sufficient for a particular application. Computer Vision System Toolbox provides the `trainCascadeObjectDetector` function to train a custom classifier.
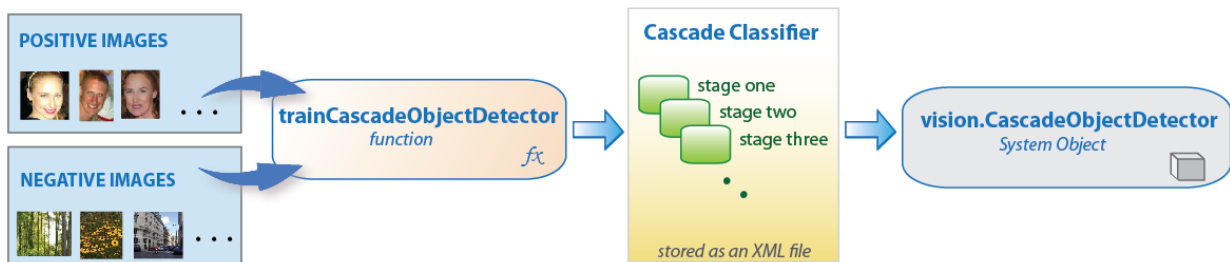
What Kinds of Objects Can You Detect?

The Computer Vision System Toolbox cascade object detector can detect object categories whose aspect ratio does not vary significantly. Objects whose aspect ratio remains fixed include faces, stop signs, and cars viewed from one side.

The `vision.CascadeObjectDetector` System object detects objects in images by sliding a window over the image. The detector then uses a cascade classifier to decide

whether the window contains the object of interest. The size of the window varies to detect objects at different scales, but its aspect ratio remains fixed. The detector is very sensitive to out-of-plane rotation, because the aspect ratio changes for most 3-D objects. Thus, you need to train a detector for each orientation of the object. Training a single detector to handle all orientations will not work.

How Does the Cascade Classifier Work?

The cascade classifier consists of stages, where each stage is an ensemble of weak learners. The weak learners are simple classifiers called decision stumps. Each stage is trained using a technique called boosting. Boosting provides the ability to train a highly accurate classifier by taking a weighted average of the decisions made by the weak learners.

Each stage of the classifier labels the region defined by the current location of the sliding window as either positive or negative. Positive indicates that an object was found and negative indicates no objects were found. If the label is negative, the classification of this region is complete, and the detector slides the window to the next location. If the label is positive, the classifier passes the region to the next stage. The detector reports an object found at the current window location when the final stage classifies the region as positive.

The stages are designed to reject negative samples as fast as possible. The assumption is that the vast majority of windows do not contain the object of interest. Conversely, true positives are rare and worth taking the time to verify.

- A true positive occurs when a positive sample is correctly classified.
- A false positive occurs when a negative sample is mistakenly classified as positive.
- A false negative occurs when a positive sample is mistakenly classified as negative.

To work well, each stage in the cascade must have a low false negative rate. If a stage incorrectly labels an object as negative, the classification stops, and you cannot correct the mistake. However, each stage can have a high false positive rate. Even if the detector incorrectly labels a nonobject as positive, you can correct the mistake in subsequent stages.

The overall false positive rate of the cascade classifier is f^s, where f is the false positive rate per stage in the range (0 1), and s is the number of stages. Similarly, the overall true positive rate is t^s, where t is the true positive rate per stage in the range (0 1]. Thus,

adding more stages reduces the overall false positive rate, but it also reduces the overall true positive rate.

Create a Cascade Classifier Using the trainCascadeObjectDetector

Cascade classifier training requires a set of positive samples and a set of negative images. You must provide a set of positive images with regions of interest specified to be used as positive samples. You can use the **Image Labeler** to label objects of interest with bounding boxes. The Image Labeler outputs a table to use for positive samples. You also must provide a set of negative images from which the function generates negative samples automatically. To achieve acceptable detector accuracy, set the number of stages, feature type, and other function parameters.
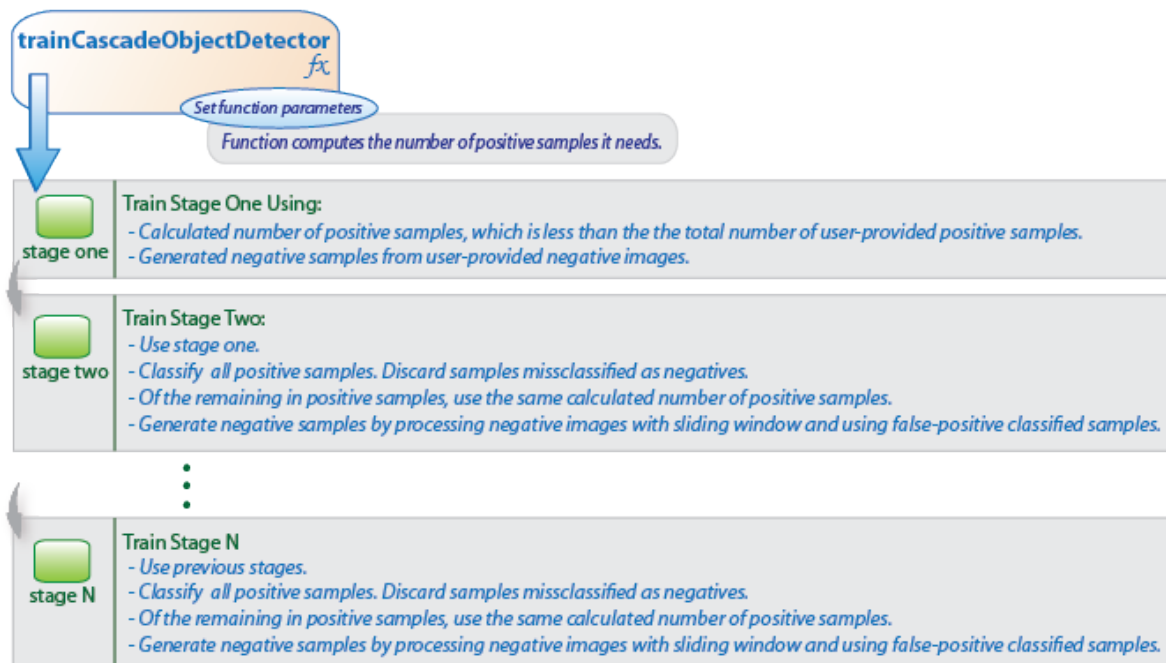
Considerations when Setting Parameters

Select the function parameters to optimize the number of stages, the false positive rate, the true positive rate, and the type of features to use for training. When you set the parameters, consider these tradeoffs.

Condition	Consideration
A large training set (in the thousands).	Increase the number of stages and set a higher false positive rate for each stage.
A small training set.	Decrease the number of stages and set a lower false positive rate for each stage.
To reduce the probability of missing an object.	Increase the true positive rate. However, a high true positive rate can prevent you from achieving the desired false positive rate per stage, making the detector more likely to produce false detections.
To reduce the number of false detections.	Increase the number of stages or decrease the false alarm rate per stage.

Feature Types Available for Training

Choose the feature that suits the type of object detection you need. The `trainCascadeObjectDetector` supports three types of features: Haar, local binary patterns (LBP), and histograms of oriented gradients (HOG). Haar and LBP features are often used to detect faces because they work well for representing fine-scale textures. The HOG features are often used to detect objects such as people and cars. They are useful for capturing the overall shape of an object. For example, in the following visualization of the HOG features, you can see the outline of the bicycle.
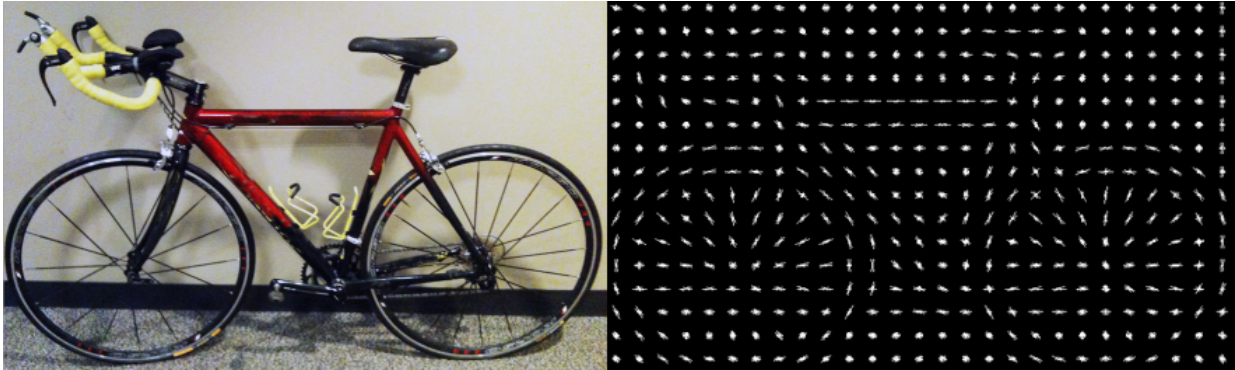
You might need to run the `trainCascadeObjectDetector` function multiple times to tune the parameters. To save time, you can use LBP or HOG features on a small subset of your data. Training a detector using Haar features takes much longer. After that, you can run the Haar features to see if the accuracy improves.

Supply Positive Samples

To create positive samples easily, you can use the **Image Labeler** app. The Image Labeler provides an easy way to label positive samples by interactively specifying rectangular regions of interest (ROIs).
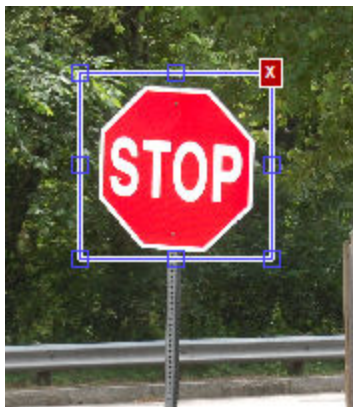
You can also specify positive samples manually in one of two ways. One way is to specify rectangular regions in a larger image. The regions contain the objects of interest. The other approach is to crop out the object of interest from the image and save it as a

separate image. Then, you can specify the region to be the entire image. You can also generate more positive samples from existing ones by adding rotation or noise, or by varying brightness or contrast.

Supply Negative Images

Negative samples are not specified explicitly. Instead, the `trainCascadeObjectDetector` function automatically generates negative samples from user-supplied negative images that do not contain objects of interest. Before training each new stage, the function runs the detector consisting of the stages already trained on the negative images. Any objects detected from these image are false positives, which are used as negative samples. In this way, each new stage of the cascade is trained to correct mistakes made by previous stages.
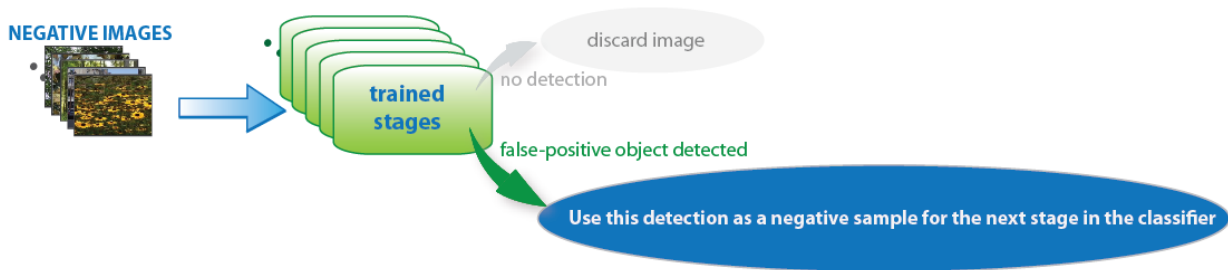
As more stages are added, the detector's overall false positive rate decreases, causing generation of negative samples to be more difficult. For this reason, it is helpful to supply as many negative images as possible. To improve training accuracy, supply negative images that contain backgrounds typically associated with the objects of interest. Also, include negative images that contain nonobjects similar in appearance to the objects of interest. For example, if you are training a stop-sign detector, include negative images that contain road signs and shapes similar to a stop sign.

Choose the Number of Stages

There is a trade-off between fewer stages with a lower false positive rate per stage or more stages with a higher false positive rate per stage. Stages with a lower false positive rate are more complex because they contain a greater number of weak learners. Stages with a higher false positive rate contain fewer weak learners. Generally, it is better to have a greater number of simple stages because at each stage the overall false positive rate decreases exponentially. For example, if the false positive rate at each stage is 50%, then the overall false positive rate of a cascade classifier with two stages is 25%. With

three stages, it becomes 12.5%, and so on. However, the greater the number of stages, the greater the amount of training data the classifier requires. Also, increasing the number of stages increases the false negative rate. This increase results in a greater chance of rejecting a positive sample by mistake. Set the false positive rate (`FalseAlarmRate`) and the number of stages, (`NumCascadeStages`) to yield an acceptable overall false positive rate. Then you can tune these two parameters experimentally.

Training can sometimes terminate early. For example, suppose that training stops after seven stages, even though you set the number of stages parameter to 20. It is possible that the function cannot generate enough negative samples. If you run the function again and set the number of stages to seven, you do not get the same result. The results between stages differ because the number of positive and negative samples to use for each stage is recalculated for the new number of stages.

Training Time of Detector

Training a good detector requires thousands of training samples. Large amounts of training data can take hours or even days to process. During training, the function displays the time it took to train each stage in the MATLAB Command Window. Training time depends on the type of feature you specify. Using Haar features takes much longer than using LBP or HOG features.

Troubleshooting

What if you run out of positive samples?

The `trainCascadeObjectDetector` function automatically determines the number of positive samples to use to train each stage. The number is based on the total number of positive samples supplied by the user and the values of the `TruePositiveRate` and `NumCascadeStages` parameters.

The number of available positive samples used to train each stage depends on the true positive rate. The rate specifies what percentage of positive samples the function can classify as negative. If a sample is classified as a negative by any stage, it never reaches subsequent stages. For example, suppose you set the `TruePositiveRate` to `0.9`, and all of the available samples are used to train the first stage. In this case, 10% of the positive samples are rejected as negatives, and only 90% of the total positive samples are available for training the second stage. If training continues, then each stage is trained with fewer and fewer samples. Each subsequent stage must solve an increasingly more

difficult classification problem with fewer positive samples. With each stage getting fewer samples, the later stages are likely to overfit the data.

Ideally, use the same number of samples to train each stage. To do so, the number of positive samples used to train each stage must be less than the total number of available positive samples. The only exception is that when the value of `TruePositiveRate` times the total number of positive samples is less than 1, no positive samples are rejected as negatives.

The function calculates the number of positive samples to use at each stage using the following formula:

number of positive samples = `floor(totalPositiveSamples / (1 + (NumCascadeStages - 1) * (1 - TruePositiveRate)))`

This calculation does not guarantee that the same number of positive samples are available for each stage. The reason is that it is impossible to predict with certainty how many positive samples will be rejected as negatives. The training continues as long as the number of positive samples available to train a stage is greater than 10% of the number of samples the function determined automatically using the preceding formula. If there are not enough positive samples the training stops and the function issues a warning. The function also outputs a classifier consisting of the stages that it had trained up to that point. If the training stops, you can add more positive samples. Alternatively, you can increase `TruePositiveRate`. Reducing the number of stages can also work, but such reduction can also result in a higher overall false alarm rate.

What to do if you run out of negative samples?

The function calculates the number of negative samples used at each stage. This calculation is done by multiplying the number of positive samples used at each stage by the value of `NegativeSamplesFactor`.

Just as with positive samples, there is no guarantee that the calculated number of negative samples are always available for a particular stage. The `trainCascadeObjectDetector` function generates negative samples from the negative images. However, with each new stage, the overall false alarm rate of the cascade classifier decreases, making it less likely to find the negative samples.

The training continues as long as the number of negative samples available to train a stage is greater than 10% of the calculated number of negative samples. If there are not enough negative samples, the training stops and the function issues a warning. It outputs

a classifier consisting of the stages that it had trained up to that point. When the training stops, the best approach is to add more negative images. Alternatively, you can reduce the number of stages or increase the false positive rate.

Examples

Train a Five-Stage Stop-Sign Detector

This example shows you how to set up and train a five-stage, stop-sign detector, using 86 positive samples. The default value for `TruePositiveRate` is `0.995`.

Step 1: Load the positive samples data from a MAT-file. In this example, file names and bounding boxes are contained in the array of structures labeled `'data'`.

```
load('stopSigns.mat');
```

Step 2: Add the image directory to the MATLAB path.

```
imDir = fullfile(matlabroot,'toolbox','vision','visiondata','stopSignImages');
addpath(imDir);
```

Step 3: Specify the folder with negative images.

```
negativeFolder = fullfile(matlabroot,'toolbox','vision','visiondata','nonStopSigns');
```

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector.xml',data,negativeFolder,'FalseAlarmRate',0.2,'NumCascadeStages',5);
```

Computer Vision System Toolbox software returns the following message:

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 86 of 86 positive samples per stage
Using at most 172 negative samples per stage

Training stage 1 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 2 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 3 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 4 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 5 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training complete
```

All 86 positive samples were used to train each stage. This high rate occurs because the true positive rate is very high relative to the number of positive samples.

Train a Five-Stage Stop-Sign Detector with a Decreased True Positive Rate

This example shows you how to train a stop-sign detector on the same data set as the first example, (steps 1–3), but with the TruePositiveRate decreased to 0.98.

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector_tpr0_98.xml',data,negativeFolder,...
'FalseAlarmRate',0.2,'NumCascadeStages', 5,...
'TruePositiveRate', 0.98);
```

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 79 of 86 positive samples per stage
Using at most 158 negative samples per stage

Training stage 1 of 5
[.....................................................................]
Used 79 positive and 158 negative samples

Training stage 2 of 5
[.....................................................................]
Used 79 positive and 158 negative samples

Training stage 3 of 5
[.....................................................................]
Used 79 positive and 158 negative samples

Training stage 4 of 5
[.....................................................................]
Used 79 positive and 158 negative samples

Training stage 5 of 5
[.....................................................................]
Used 79 positive and 85 negative samples

Training complete
```

Only 79 of the total 86 positive samples were used to train each stage. This lowered rate occurs because the true positive rate was low enough for the function to start rejecting some of the positive samples as false negatives.

Train a Ten-Stage Stop-Sign Detector

This example shows you how to train a stop-sign detector on the same data set as the first example, (steps 1–3), but with the number of stages increased to 10.

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector_10stages.xml',data,negativeFolder,...
'FalseAlarmRate',0.2,'NumCascadeStages',10);
```

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 86 of 86 positive samples per stage
Using at most 172 negative samples per stage

Training stage 1 of 10
[........................................................................]
Used 86 positive and 172 negative samples

Training stage 2 of 10
[........................................................................]
Used 86 positive and 172 negative samples

Training stage 3 of 10
[........................................................................]
Used 86 positive and 172 negative samples

Training stage 4 of 10
[........................................................................]
Used 86 positive and 172 negative samples

Training stage 5 of 10
[........................................................................]
Used 86 positive and 172 negative samples

Training stage 6 of 10
[........................................................................]
Used 86 positive and 33 negative samples

Training stage 7 of 10
[...........................................Warning:
Unable to generate a sufficient number of negative samples for this stage.
Consider reducing the number of stages, reducing the false alarm rate
or adding more negative images.

Cannot find enough samples for training.
Training will halt and return cascade detector with 6 stages
Training complete
```

In this case, `NegativeSamplesFactor` was set to 2, therefore the number of negative samples used to train each stage was 172. Notice that the function generated only 33 negative samples for stage 6 and was not able to train stage 7 at all. This condition occurs because the number of negatives in stage 7 was less than 17, (roughly half of the previous number of negative samples). The function produced a stop-sign detector with 6 stages, instead of the 10 previously specified. The resulting overall false alarm rate is 0.2^7=1.28e-05, while the expected false alarm rate is 1.024e-07.

At this point, you can add more negative images, reduce the number of stages, or increase the false positive rate. For example, you can increase the false positive rate, `FalseAlarmRate`, to 0.5. The expected overall false-positive rate in this case is 0.0039.

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector_10stages_far0_5.xml',data,negativeFolder,...
'FalseAlarmRate',0.5,'NumCascadeStages',10);
```

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 86 of 86 positive samples per stage
Using at most 172 negative samples per stage

Training stage 1 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 2 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 3 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 4 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 5 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 6 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 7 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 8 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 9 of 10
[.........................................................
Very low false alarm rate 0.000587108 reached in stage.
 Training will halt and return cascade detector with 8 stages
Training complete
```

This time the function trains eight stages before the threshold reaches the overall false alarm rate of 0.000587108 and training stops.

Train Stop Sign Detector

Load the positive samples data from a MAT file. The file contains a table specifying bounding boxes for several object categories. The table was exported from the `Training Image Labeler` app.

Load positive samples.

```
load('stopSignsAndCars.mat');
```

Select the bounding boxes for stop signs from the table.

```
positiveInstances = stopSignsAndCars(:,1:2);
```

Add the image folder to the MATLAB path.

```
imDir = fullfile(matlabroot,'toolbox','vision','visiondata',...
    'stopSignImages');
addpath(imDir);
```

Specify the folder for negative images.

```
negativeFolder = fullfile(matlabroot,'toolbox','vision','visiondata',...
    'nonStopSigns');
```

Create an `imageDatastore` object containing negative images.

```
negativeImages = imageDatastore(negativeFolder);
```

Train a cascade object detector called 'stopSignDetector.xml' using HOG features. NOTE: The command can take several minutes to run.

```
trainCascadeObjectDetector('stopSignDetector.xml',positiveInstances, ...
    negativeFolder,'FalseAlarmRate',0.1,'NumCascadeStages',5);
```

```
Automatically setting ObjectTrainingSize to [35, 32]
Using at most 42 of 42 positive samples per stage
Using at most 84 negative samples per stage

--cascadeParams--
Training stage 1 of 5
```

```
[..................................................................]
Used 42 positive and 84 negative samples
Time to train stage 1: 1 seconds

Training stage 2 of 5
[..................................................................]
Used 42 positive and 84 negative samples
Time to train stage 2: 0 seconds

Training stage 3 of 5
[..................................................................]
Used 42 positive and 84 negative samples
Time to train stage 3: 3 seconds

Training stage 4 of 5
[..................................................................]
Used 42 positive and 84 negative samples
Time to train stage 4: 9 seconds

Training stage 5 of 5
[..................................................................]
Used 42 positive and 17 negative samples
Time to train stage 5: 14 seconds

Training complete
```

Use the newly trained classifier to detect a stop sign in an image.

```
detector = vision.CascadeObjectDetector('stopSignDetector.xml');
```

Read the test image.

```
img = imread('stopSignTest.jpg');
```

Detect a stop sign.

```
bbox = step(detector,img);
```

Insert bounding box rectangles and return the marked image.

```
 detectedImg = insertObjectAnnotation(img,'rectangle',bbox,'stop sign');
```

Display the detected stop sign.

```
figure; imshow(detectedImg);
```

7-105

Remove the image directory from the path.

```
rmpath(imDir);
```

See Also

More About
- "Define Ground Truth for Image Collections" on page 7-54

External Websites
- Cascade Trainer

Train Optical Character Recognition for Custom Fonts

In this section...
"Open the OCR Trainer App" on page 7-107
"Train OCR" on page 7-107
"App Controls" on page 7-109

The optical character recognition (OCR) app trains the `ocr` function to recognize a custom language or font. You can use this app to label character data interactively for OCR training and to generate an OCR language data file for use with the `ocr` function.

Open the OCR Trainer App

- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click , the OCR app icon.
- MATLAB command prompt: Enter `ocrTrainer`.

Train OCR

1   In the OCR Trainer, click **New Session** to open the OCR Training Session Settings dialog box.

2   Under **Output Settings**, enter a name for the OCR language data file and choose the output folder location for the file. The location you specify must be writable.

3   Under **Labeling Method**, either label the data manually or pre-label it using optical character recognition. If you use OCR, you can select either the pre-installed English or Japanese language, or you can download additional language support files.

> **Note** To download a language support file, type `visionSupportPackages` in a MATLAB Command Window. Alternatively, on the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Add-Ons**. Then use the search box to find "Computer Vision System Toolbox OCR Language Data."

To limit the OCR to a specific character set, select the **Character set** check box and add the characters.

4   Add images at any time during the training session. The trainer automatically segments the images for OCR training. Inspect the results to verify expected text segmentation. To improve the segmentation, pre-process your images using the **Image Segmenter** app. Once the images are added, you can inspect segmentation results from the training image view.
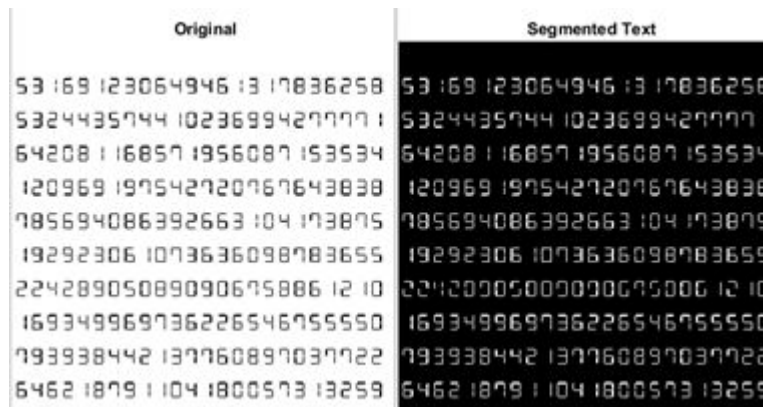
5   Remove any noisy images. To improve segmentation results, you can draw a region of interest to select a portion of an image. The display shows the original image on the left and the edited one on the right. When you are done, click **Accept All**.

6   Modify the extracted samples from the character view window.

   •   To correct samples, select a group of samples from the **Data Browser** pane and change the labels using the **Character Label** field.

   •   To exclude a sample from training, right-click the sample and select the option to move that sample to the **Unknown** category. Unknown samples are listed at the top of the **Data Browser** pane and are not used for training.

   •   If the bounding box clipped a character, double-click the character and modify it in the image it was extracted from.
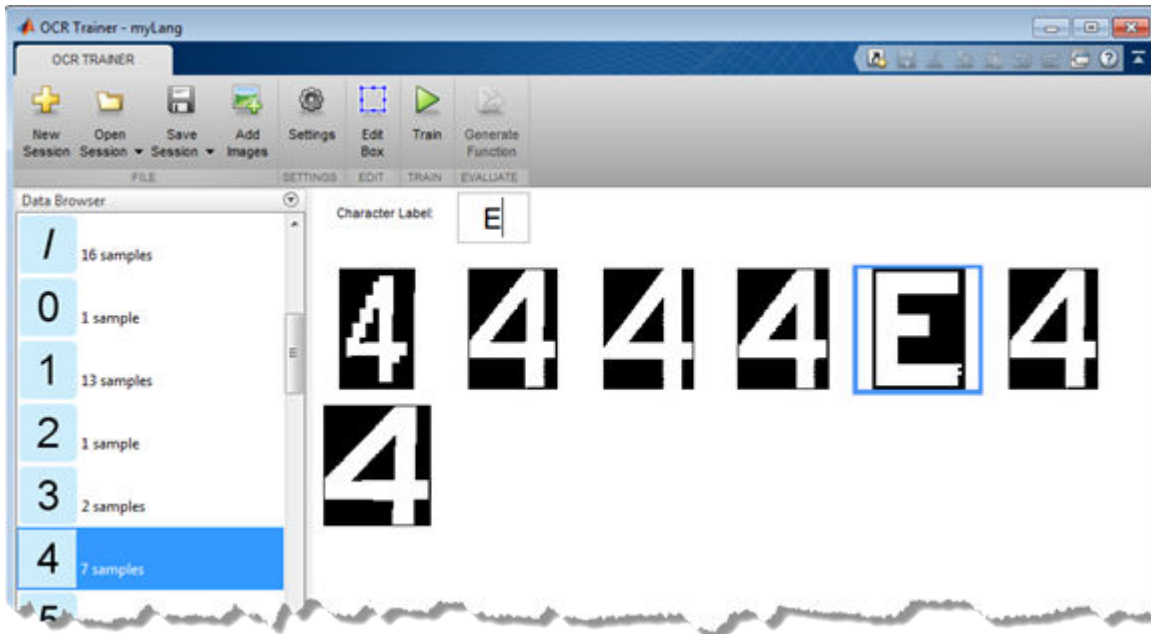
7   After correcting the samples, click **Train**. When the trainer completes training, the app creates an OCR language data file and saves it to the folder you specified.

App Controls

Sessions

Starts a new session, opens a saved session, or adds a session to the current one. You can also save and name the session. The sessions are saved as MAT files.

Add Images

Adds images. You can add images when you start a new session or after you accept the current collection of images.

Settings

Set or change the font display.

7-109

Edit Box

Selects the image that contains the selected character, along with the bounding boxes. You can create additional regions, merge, modify, or delete existing images. To delete an ROI, use the **delete** key.

Train

Creates an OCR data file from the session. To use the `.traineddata` file with the `ocr` function, set the `'Language'` property for the `ocr` function, and follow the directions for a custom language.

Generate Function

Creates an autogenerated evaluation function for verification of training results.

See Also
OCR Trainer | `ocr`

Troubleshoot ocr Function Results

Performance Options with the ocr Function

If your `ocr` results are not what you expect, try one or more of the following options:

- Increase image size 2-to-4 times larger.
- If the characters in the image are too close together or their edges are touching, use morphology to thin out the characters. Using morphology to thin out the characters separates the characters.
- Use binarization to check for non-uniform lighting issues. Use the `graythresh` and `imbinarize` functions to binarize the image. If the characters are not visible in the results of the binarization, it indicates a potential non-uniform lighting issue. Try top hat, using the `imtophat` function, or other techniques that deal with removing non-uniform illumination.
- Use the region of interest `roi` option to isolate the text. Specify the `roi` manually or use text detection.
- If your image looks like a natural scene containing words, like a street scene, rather than a scanned document, try setting the `TextLayout` property to either `'Block'` or `'Word'`.

See Also
`graythresh` | `imbinarize` | `imtophat` | `ocr` | `ocrText` | `visionSupportPackages`

More About
- "Install Computer Vision System Toolbox Add-on Support Files" on page 3-2

7-111

Create a Custom Feature Extractor

You can use the bag-of-features (BoF) framework with many different types of image features. To use a custom feature extractor instead of the default speeded-up robust features (SURF) feature extractor, use the `CustomExtractor` property of a `bagOfFeatures` object.

Example of a Custom Feature Extractor

This example shows how to write a custom feature extractor function for `bagOfFeatures`. You can open this example function file and use it as a template by typing the following command at the MATLAB command prompt:

```
edit('exampleBagOfFeaturesExtractor.m')
```

- Step 1. Define the image sets.
- Step 2. Create a new extractor function file.
- Step 3. Preprocess the image.
- Step 4. Select a point location for feature extraction.
- Step 5. Extract features.
- Step 6. Compute the feature metric.

Define the set of images and labels

Read the category images and create image sets.

```
setDir  = fullfile(toolboxdir('vision'),'visiondata','imageSets');
imds = imageDatastore(setDir,'IncludeSubfolders',true,'LabelSource',...
    'foldernames');
```

Create a new extractor function file

The extractor function must be specified as a function handle:

```
extractorFcn = @exampleBagOfFeaturesExtractor;
bag = bagOfFeatures(imgSets,'CustomExtractor',extractorFcn)
```

`exampleBagOfFeaturesExtractor` is a MATLAB function. For example:

```
function [features,featureMetrics] = exampleBagOfFeaturesExtractor(img)
...
```

You can also specify the optional `location` output:

```
function [features,featureMetrics,location] = exampleBagOfFeaturesExtractor(img)
...
```

The function must be on the path or in the current working folder.

Argument	Input/Output	Description
`img`	Input	• Binary, grayscale, or truecolor image. • The input image is from the image set that was originally passed into `bagOfFeatures`.
`features`	Output	• An *M*-by-*N* numeric matrix of image features, where *M* is the number of features and *N* is the length of each feature vector. • The feature length, *N*, must be greater than zero and be the same for all images processed during the `bagOfFeatures` creation process. • If you cannot extract features from an image, supply an empty feature matrix and an empty feature metrics vector. Use the empty matrix and vector if, for example, you did not find any keypoints for feature extraction. • Numeric, real, and nonsparse.
`featureMetrics`	Output	• An *M*-by-1 vector of feature metrics indicating the strength of each feature vector. • Used to apply the `'SelectStrongest'` criteria in `bagOfFeatures` framework. • Numeric, real, and nonsparse.
`location`	Output	• An *M*-by-2 matrix of 1-based [*x y*] values. • The [*x y*] values can be fractional. • Numeric, real, and nonsparse.

Preprocess the image

Input images can require preprocessing before feature extraction. To extract SURF features and to use the `detectSURFFeatures` or `detectMSERFeatures` functions, the images must be grayscale. If the images are not grayscale, you can convert them using the `rgb2gray` function.

```
[height,width,numChannels] = size(I);
if numChannels > 1
    grayImage = rgb2gray(I);
else
    grayImage = I;
end
```

Select a point location for feature extraction

Use a regular spaced grid of point locations. Using the grid over the image allows for dense SURF feature extraction. The grid step is in pixels.

```
gridStep = 8;
gridX = 1:gridStep:width;
gridY = 1:gridStep:height;

[x,y] = meshgrid(gridX,gridY);

gridLocations = [x(:) y(:)];
```

You can manually concatenate multiple SURFPoints objects at different scales to achieve multiscale feature extraction.

```
multiscaleGridPoints = [SURFPoints(gridLocations,'Scale',1.6);
    SURFPoints(gridLocations,'Scale',3.2);
    SURFPoints(gridLocations,'Scale',4.8);
    SURFPoints(gridLocations,'Scale',6.4)];
```

Alternatively, you can use a feature detector, such as detectSURFFeatures or detectMSERFeatures, to select point locations.

```
multiscaleSURFPoints = detectSURFFeatures(I);
```

Extract features

Extract features from the selected point locations. By default, bagOfFeatures extracts upright SURF features.

```
features = extractFeatures(grayImage,multiscaleGridPoints,'Upright',true);
```

Compute the feature metric

The feature metrics indicate the strength of each feature. Larger metric values are assigned to stronger features. Use feature metrics to identify and remove weak features

before using `bagOfFeatures` to learn the visual vocabulary of an image set. Use the metric that is suitable for your feature vectors.

For example, you can use the variance of the SURF features as the feature metric.

```
featureMetrics = var(features,[],2);
```

If you used a feature detector for the point selection, then use the detection metric instead.

```
featureMetrics = multiscaleSURFPoints.Metric;
```

You can optionally return the feature location information. The feature location can be used for spatial or geometric verification image search applications. See the "Geometric Verification Using estimateGeometricTransform Function" example. The `retrieveImages` and `indexImages` functions are used for content-based image retrieval systems.

```
if nargout > 2
    varargout{1} = multiscaleGridPoints.Location;
end
```

Image Retrieval with Bag of Visual Words

You can use the Computer Vision System Toolbox functions to search by image, also known as a content-based image retrieval (CBIR) system. CBIR systems are used to retrieve images from a collection of images that are similar to a query image. The application of these types of systems can be found in many areas such as a web-based product search, surveillance, and visual place identification. First the system searches a collection of images to find the ones that are visually similar to a query image.

The retrieval system uses a bag of visual words, a collection of image descriptors, to represent your data set of images. Images are indexed to create a mapping of visual words. The index maps each visual word to their occurrences in the image set. A comparison between the query image and the index provides the images most similar to the query image. By using the CBIR system workflow, you can evaluate the accuracy for a known set of image search results.

Create image set

imds = `imageDatastore`(imageFolder)

Type of feature for retrieval?

SURF

non-SURF

Use custom feature extractor

extractor = @yourOwnExtractor

Create bag of visual words

[imds = `imageDatastore`(trainingImagesFolder)] optional
bag = `bagOfFeatures`(imds, 'CustomExtractor', extractor)

Index images

imageIndex = `indexImages`(imds)   imageIndex = `indexImages`(imds, bag)

Search image set

imageIDs = `retrieveImages`(queryImage, imageIndex)
[imageIDs, scores] = `retrieveImages`(queryImage, imageIndex)
[imageIDs, scores, imageWords] = `retrieveImages`(queryImage, imageIndex)

imageIndex

query image

visual words {

W_0 → I_1, I_{51}, I_{100}, I_{120}, ...

W_{n-1} → I_1, I_{73}, I_{100}, I_{233}, ...

150
.
.
23
19
3

I_{150}
.
.
I_{23}
I_{19}
I_3

imageIDs

7-117

Retrieval System Workflow

1 **Create image set that represents image features for retrieval**. Use `imageDatastore` to store the image data. Use a large number of images that represent various viewpoints of the object. A large and diverse number of images helps train the bag of visual words and increases the accuracy of the image search.

2 **Type of feature**. The `indexImages` function creates the bag of visual words using the speeded up robust features (SURF). For other types of features, you can use a custom extractor, and then use `bagOfFeatures` to create the bag of visual words. See the "Create Search Index Using Custom Bag of Features" example.

You can use the original `imgSet` or a different collection of images for the training set. To use a different collection, create the bag of visual words before creating the image index, using the `bagOfFeatures` function. The advantage of using the same set of images is that the visual vocabulary is tailored to the search set. The disadvantage of this approach is that the retrieval system must relearn the visual vocabulary to use on a drastically different set of images. With an independent set, the visual vocabulary is better able to handle the additions of new images into the search index.

3 **Index the images**. The `indexImages` function creates a search index that maps visual words to their occurrences in the image collection. When you create the bag of visual words using an independent or subset collection, include the `bag` as an input argument to `indexImages`. If you do not create an independent bag of visual words, then the function creates the bag based on the entire `imgSet` input collection. You can add and remove images directly to and from the image index using the `addImages` and `removeImages` methods.

4 **Search data set for similar images**. Use the `retrieveImages` function to search the image set for images which are similar to the query image. Use the `NumResults` property to control the number of results. For example, to return the top 10 similar images, set the `ROI` property to use a smaller region of a query image. A smaller region is useful for isolating a particular object in an image that you want to search for.

Evaluate Image Retrieval

Use the `evaluateImageRetrieval` function to evaluate image retrieval by using a query image with a known set of results. If the results are not what you expect, you can modify or augment image features by the bag of visual words. Examine the type of the features retrieved. The type of feature used for retrieval depends on the type of images

within the collection. For example, if you are searching an image collection made up of scenes, such as beaches, cities, or highways, use a global image feature. A global image feature, such as a color histogram, captures the key elements of the entire scene. To find specific objects within the image collections, use local image features extracted around object keypoints instead.

See Also

Related Examples

- "Image Retrieval Using Customized Bag of Features"

Image Classification with Bag of Visual Words

Use the Computer Vision System Toolbox functions for image category classification by creating a bag of visual words. The process generates a histogram of visual word occurrences that represent an image. These histograms are used to train an image category classifier. The steps below describe how to setup your images, create the bag of visual words, and then train and apply an image category classifier.

Step 1: Set Up Image Category Sets

Organize and partition the images into training and test subsets. Use the `imageDatastore` function to store images to use for training an image classifier. Organizing images into categories makes handling large sets of images much easier. You can use the `splitEachLabel` function to split the images into training and test data.

Read the category images and create image sets.

```
setDir  = fullfile(toolboxdir('vision'),'visiondata','imageSets');
imds = imageDatastore(setDir,'IncludeSubfolders',true,'LabelSource',...
    'foldernames');
```

Separate the sets into training and test image subsets. In this example, 30% of the images are partitioned for training and the remainder for testing.

```
[trainingSet,testSet] = splitEachLabel(imds,0.3,'randomize');
```
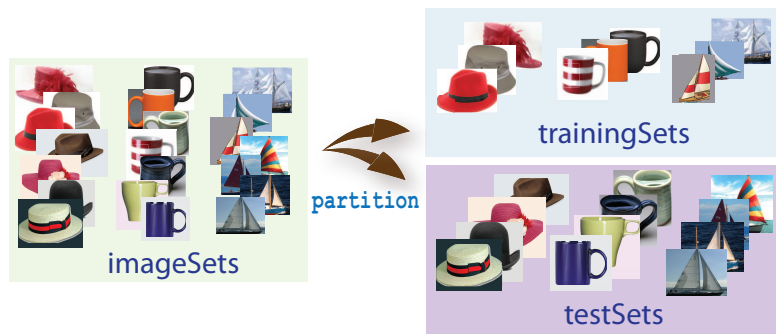
Step 2: Create Bag of Features

Create a visual vocabulary, or bag of features, by extracting feature descriptors from representative images of each category.

The `bagOfFeatures` object defines the features, or visual words, by using the k-means clustering (Statistics and Machine Learning Toolbox) algorithm on the feature descriptors extracted from `trainingSets`. The algorithm iteratively groups the descriptors into *k* mutually exclusive clusters. The resulting clusters are compact and separated by similar characteristics. Each cluster center represents a feature, or visual word.

You can extract features based on a feature detector, or you can define a grid to extract feature descriptors. The grid method may lose fine-grained scale information. Therefore, use the grid for images that do not contain distinct features, such as an image containing scenery, like the beach. Using speeded up robust features (or SURF) detector provides greater scale invariance. By default, the algorithm runs the `'grid'` method.
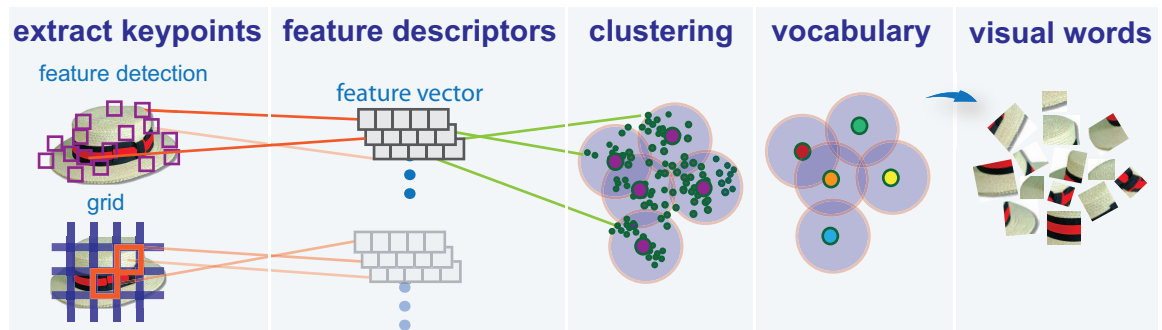
This algorithm workflow analyzes images in their entirety. Images must have appropriate labels describing the class that they represent. For example, a set of car images could be labeled cars. The workflow does not rely on spatial information nor on marking the particular objects in an image. The bag-of-visual-words technique relies on detection without localization.

Step 3: Train an Image Classifier With Bag of Visual Words

The `trainImageCategoryClassifier` function returns an image classifier. The function trains a multiclass classifier using the error-correcting output codes (ECOC) framework with binary support vector machine (SVM) classifiers. The `trainImageCategoryClassfier` function uses the bag of visual words returned by the

`bagOfFeatures` object to encode images in the image set into the histogram of visual words. The histogram of visual words are then used as the positive and negative samples to train the classifier.

1 Use the `bagOfFeatures encode` method to encode each image from the training set. This function detects and extracts features from the image and then uses the approximate nearest neighbor algorithm to construct a feature histogram for each image. The function then increments histogram bins based on the proximity of the descriptor to a particular cluster center. The histogram length corresponds to the number of visual words that the `bagOfFeatures` object constructed. The histogram becomes a feature vector for the image.
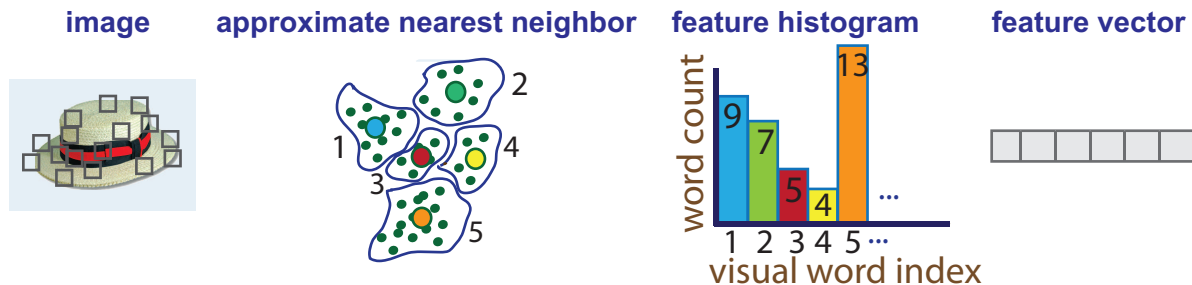
2 Repeat step 1 for each image in the training set to create the training data.
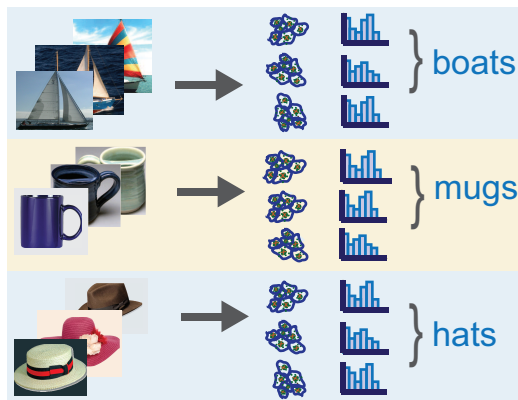
3 Evaluate the quality of the classifier. Use the `imageCategoryClassifier evaluate` method to test the classifier against the validation image set. The output confusion matrix represents the analysis of the prediction. A perfect classification

results in a normalized matrix containing 1s on the diagonal. An incorrect classification results fractional values.
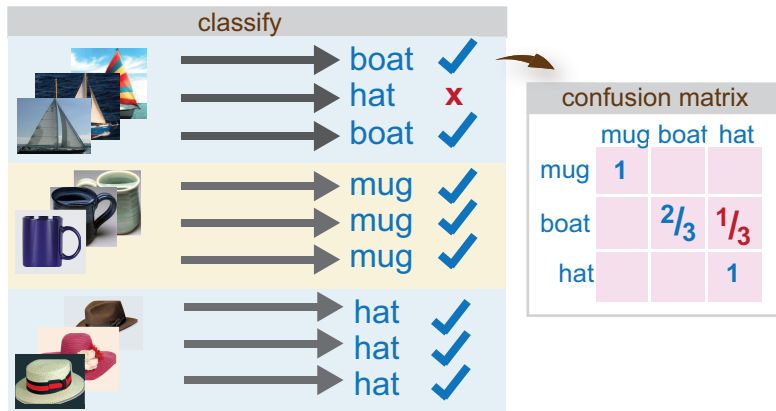
Step 4: Classify an Image or Image Set

Use the `imageCategoryClassifier` `predict` method on a new image to determine its category.

References

[1] Csurka, G., C. R. Dance, L. Fan, J. Willamowski, and C. Bray. *Visual Categorization with Bags of Keypoints*. Workshop on Statistical Learning in Computer Vision. ECCV 1 (1–22), 1–2.

See Also

Related Examples

- "Image Category Classification Using Bag of Features"
- "Image Retrieval Using Customized Bag of Features"

8

Motion Estimation and Tracking

Multiple Object Tracking

Tracking is the process of locating a moving object or multiple objects over time in a video stream. Tracking an object is not the same as object detection. Object detection is the process of locating an object of interest in a single frame. Tracking associates detections of an object across multiple frames.

Tracking multiple objects requires detection, prediction, and data association.

- **Detection**: Detect objects of interest in a video frame.
- **Prediction**: Predict the object locations in the next frame.
- **Data association**: Use the predicted locations to associate detections across frames to form *tracks*.

Detection

Selecting the right approach for detecting objects of interest depends on what you want to track and whether the camera is stationary.

Detect Objects Using a Stationary Camera

To detect objects in motion with a stationary camera, you can perform background subtraction using the `vision.ForegroundDetector` System object. The background subtraction approach works efficiently but requires the camera to be stationary.

Detect Objects Using a Moving Camera

To detect objects in motion with a moving camera, you can use a sliding-window detection approach. This approach typically works more slowly than the background subtraction approach. To detect and track a specific category of object, use the System objects or functions described in the table.

Select A Detection Algorithm

Type of Object to Track	Camera	Functionality
Anything that moves	Stationary	`vision.ForegroundDetector` System object
Faces, eyes, nose, mouth, upper body	Stationary, Moving	`vision.CascadeObjectDetector` System object

Type of Object to Track	Camera	Functionality
Pedestrians	Stationary, Moving	`vision.PeopleDetector` System object
Custom object category	Stationary, Moving	`trainCascadeObjectDetector` function or custom sliding window detector using `extractHOGFeatures` and `selectStrongestBbox`

Prediction

To track an object over time means that you must predict its location in the next frame. The simplest method of prediction is to assume that the object will be near its last known location. In other words, the previous detection serves as the next prediction. This method is especially effective for high frame rates. However, using this prediction method can fail when objects move at varying speeds, or when the frame rate is low relative to the speed of the object in motion.

A more sophisticated method of prediction is to use the previously observed motion of the object. The Kalman filter (`vision.KalmanFilter`) predicts the next location of an object, assuming that it moves according to a motion model, such as constant velocity or constant acceleration. The Kalman filter also takes into account process noise and measurement noise. Process noise is the deviation of the actual motion of the object from the motion model. Measurement noise is the detection error.

To make configuring a Kalman filter easier, use `configureKalmanFilter`. This function sets up the filter for tracking a physical object moving with constant velocity or constant acceleration within a Cartesian coordinate system. The statistics are the same along all dimensions. If you need to configure a Kalman filter with different assumptions, you need to construct the `vision.KalmanFilter` object directly.

Data Association

Data association is the process of associating detections corresponding to the same physical object across frames. The temporal history of a particular object consists of multiple detections, and is called a *track*. A track representation can include the entire history of the previous locations of the object. Alternatively, it can consist only of the object's last known location and its current velocity.

Detection to Track Cost Functions

To match a detection to a track, you must establish criteria for evaluating the matches. Typically, you establish this criteria by defining a cost function. The higher the cost of matching a detection to a track, the less likely that the detection belongs to the track. A simple cost function can be defined as the degree of overlap between the bounding boxes of the predicted and detected objects. The "Tracking Pedestrians from a Moving Car" example implements this cost function using the `bboxOverlapRatio` function. You can implement a more sophisticated cost function, one that accounts for the uncertainty of the prediction, using the `distance` method of the `vision.KalmanFilter` object. You can also implement a custom cost function than can incorporate information about the object size and appearance.

Elimination of Unlikely Matches

Gating is a method of eliminating highly unlikely matches from consideration, such as by imposing a threshold on the cost function. An observation cannot be matched to a track if the cost exceeds a certain threshold value. Using this threshold method effectively results in a circular *gating region* around each prediction, where a matching detection can be found. An alternative gating technique is to make the gating region large enough to include the *k*-nearest neighbors of the prediction.

Assign Detections to Track

Data association reduces to a minimum weight bipartite matching problem, which is a well-studied area of graph theory. A bipartite graph represents tracks and detections as vertices. It also represents the cost of matching a detection and a track as a weighted edge between the corresponding vertices.

The `assignDetectionsToTracks` function implements the Munkres' variant of the Hungarian bipartite matching algorithm. Its input is the *cost matrix*, where the rows correspond to tracks and the columns correspond to detections. Each entry contains the cost of assigning a particular detection to a particular track. You can implement gating by setting the cost of impossible matches to infinity.

Track Management

Data association must take into account the fact that new objects can appear in the field of view, or that an object being tracked can leave the field of view. In other words, in any given frame, some number of new tracks might need to be created, and some number of existing tracks might need to be discarded. The `assignDetectionsToTracks` function

returns the indices of unassigned tracks and unassigned detections in addition to the matched pairs.

One way of handling unmatched detections is to create a new track from each of them. Alternatively, you can create new tracks from unmatched detections greater than a certain size, or from detections that have certain locations or appearance. For example, if the scene has a single entry point, such as a doorway, then you can specify that only unmatched detections located near the entry point can begin new tracks, and that all other detections are considered noise.

Another way of handling unmatched tracks is to delete any track that remain unmatched for a certain number of frames. Alternatively, you can specify to delete an unmatched track when its last known location is near an exit point.

See Also

assignDetectionsToTracks | bboxOverlapRatio | configureKalmanFilter | extractHOGFeatures | selectStrongestBbox | trainCascadeObjectDetector | vision.CascadeObjectDetector | vision.ForegroundDetector | vision.KalmanFilter | vision.PeopleDetector | vision.PointTracker

Related Examples

- "Tracking Pedestrians from a Moving Car"
- "Using Kalman Filter for Object Tracking"
- "Motion-Based Multiple Object Tracking"

More About

- "Train a Cascade Object Detector" on page 7-90

External Websites

- Detect and Track Multiple Faces

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:
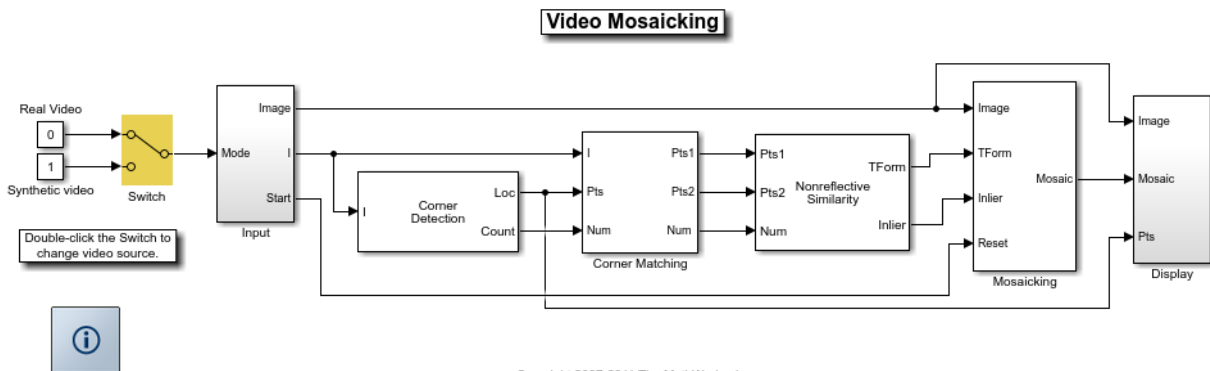
The Input subsystem loads a video sequence from either a file, or generates a synthetic video sequence. The choice is user defined. First, the Corner Detection block finds points that are matched between successive frames by the Corner Matching subsystem. Then the Estimate Geometric Transformation block 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.
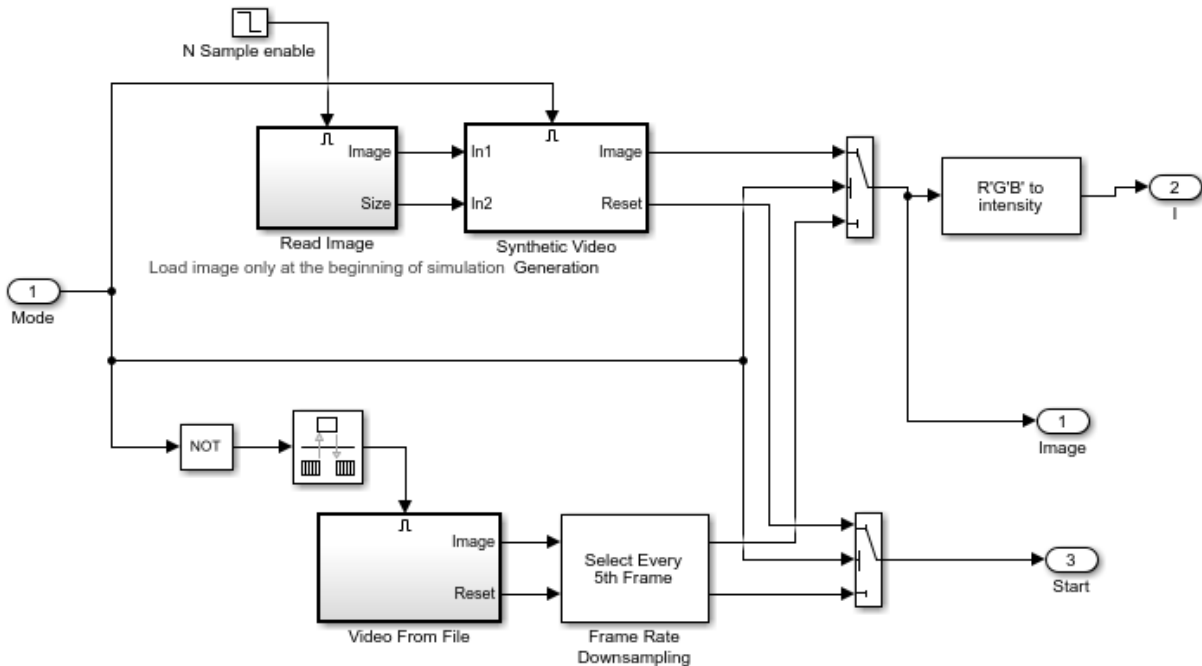
If you choose to use a video sequence from a file, you can reduce computation time by processing only some of the video frames. This is done by setting the downsampling rate in the Frame Rate Downsampling subsystem.

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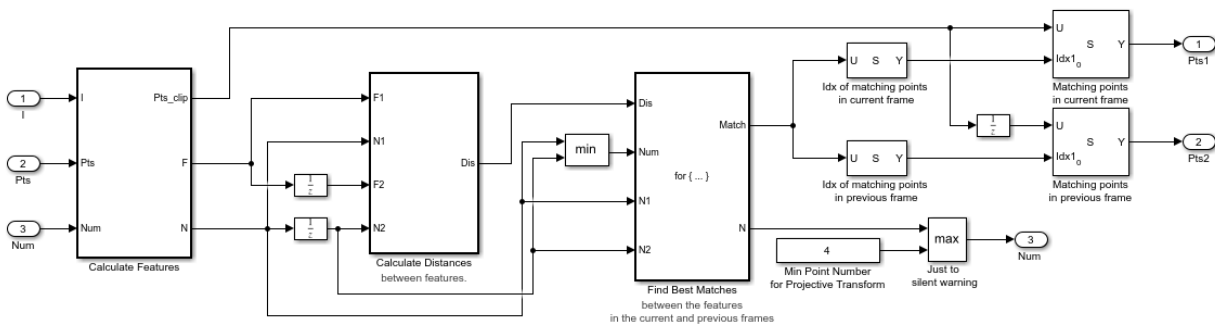
The Corner Matching Subsystem finds the number of corners, location, and their metric values. The subsystem then calculates the distances between all features in the current frame with those in the previous frame. By searching for the minimum distances, the subsystem finds the best matching features.

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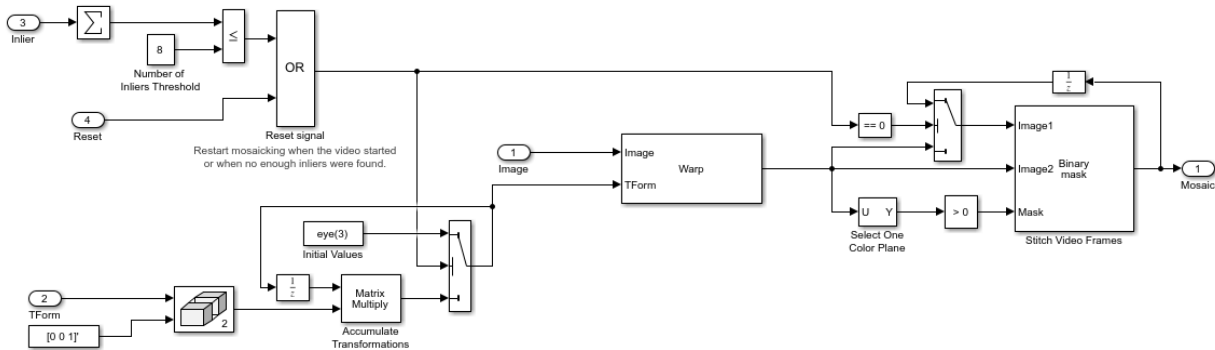
The subsystem is reset when the video sequence rewinds or when the Estimate Geometric Transformation block does not find enough inliers.

Video Mosaicking Using Synthetic Video
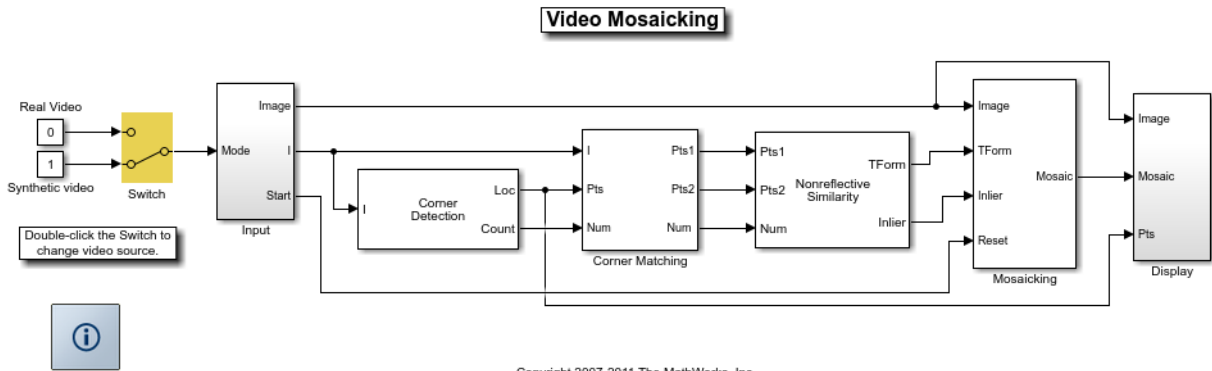
The Corners window shows the corner locations in the current video frame.

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.

Pattern Matching

This example shows how to use the 2-D normalized cross-correlation for pattern matching and target tracking. The example uses predefined or user specified target and number of similar targets to be tracked. The normalized cross correlation plot shows that when the value exceeds the set threshold, the target is identified.

Introduction

In this example you use normalized cross correlation to track a target pattern in a video. The pattern matching algorithm involves the following steps:

- The input video frame and the template are reduced in size to minimize the amount of computation required by the matching algorithm.
- Normalized cross correlation, in the frequency domain, is used to find a template in the video frame.
- The location of the pattern is determined by finding the maximum cross correlation value.

Initialize Parameters and Create a Template

Initialize required variables such as the threshold value for the cross correlation and the decomposition level for Gaussian Pyramid decomposition.

```
threshold = single(0.99);
level = 2;
```

Prepare a video file reader.

```
hVideoSrc = vision.VideoFileReader('vipboard.mp4', ...
                                   'VideoOutputDataType', 'single',...
                                   'ImageColorSpace', 'Intensity');
```

Specify the target image and number of similar targets to be tracked. By default, the example uses a predefined target and finds up to 2 similar patterns. You can set the variable useDefaultTarget to false to specify a new target and the number of similar targets to match.

```
useDefaultTarget = true;
[Img, numberOfTargets, target_image] = ...
  videopattern_gettemplate(useDefaultTarget);
```

```matlab
% Downsample the target image by a predefined factor. You do this
% to reduce the amount of computation needed by cross correlation.
target_image = single(target_image);
target_dim_nopyramid = size(target_image);
target_image_gp = multilevelPyramid(target_image, level);
target_energy = sqrt(sum(target_image_gp(:).^2));

% Rotate the target image by 180 degrees, and perform zero padding so that
% the dimensions of both the target and the input image are the same.
target_image_rot = imrotate(target_image_gp, 180);
[rt, ct] = size(target_image_rot);
Img = single(Img);
Img = multilevelPyramid(Img, level);
[ri, ci]= size(Img);
r_mod = 2^nextpow2(rt + ri);
c_mod = 2^nextpow2(ct + ci);
target_image_p = [target_image_rot zeros(rt, c_mod-ct)];
target_image_p = [target_image_p; zeros(r_mod-rt, c_mod)];

% Compute the 2-D FFT of the target image
target_fft = fft2(target_image_p);

% Initialize constant variables used in the processing loop.
target_size = repmat(target_dim_nopyramid, [numberOfTargets, 1]);
gain = 2^(level);
Im_p = zeros(r_mod, c_mod, 'single'); % Used for zero padding
C_ones = ones(rt, ct, 'single');     % Used to calculate mean using conv
```

Create a System object to calculate the local maximum value for the normalized cross correlation.

```matlab
hFindMax = vision.LocalMaximaFinder( ...
            'Threshold', single(-1), ...
            'MaximumNumLocalMaxima', numberOfTargets, ...
            'NeighborhoodSize', floor(size(target_image_gp)/2)*2 - 1);
```

Create a System object to display the tracking of the pattern.

```matlab
sz = get(0,'ScreenSize');
pos = [20 sz(4)-400 400 300];
hROIPattern = vision.VideoPlayer('Name', 'Overlay the ROI on the target', ...
    'Position', pos);
```

Initialize figure window for plotting the normalized cross correlation value

```
hPlot = videopatternplots('setup',numberOfTargets, threshold);
```

Search for a Template in Video

Create a processing loop to perform pattern matching on the input video. This loop uses the System objects you instantiated above. The loop is stopped when you reach the end of the input file, which is detected by the VideoFileReader System object.

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

    % Reduce the image size to speed up processing
    Im_gp = multilevelPyramid(Im, level);

    % Frequency domain convolution.
    Im_p(1:ri, 1:ci) = Im_gp;     % Zero-pad
    img_fft = fft2(Im_p);
    corr_freq = img_fft .* target_fft;
    corrOutput_f = ifft2(corr_freq);
    corrOutput_f = corrOutput_f(rt:ri, ct:ci);

    % Calculate image energies and block run tiles that are size of
    % target template.
    IUT_energy = (Im_gp).^2;
    IUT = conv2(IUT_energy, C_ones, 'valid');
    IUT = sqrt(IUT);

    % Calculate normalized cross correlation.
    norm_Corr_f = (corrOutput_f) ./ (IUT * target_energy);
    xyLocation = step(hFindMax, norm_Corr_f);

    % Calculate linear indices.
    linear_index = sub2ind([ri-rt, ci-ct]+1, xyLocation(:,2),...
        xyLocation(:,1));

    norm_Corr_f_linear = norm_Corr_f(:);
    norm_Corr_value = norm_Corr_f_linear(linear_index);
    detect = (norm_Corr_value > threshold);
    target_roi = zeros(length(detect), 4);
    ul_corner = (gain.*(xyLocation(detect, :)-1))+1;
    target_roi(detect, :) = [ul_corner, fliplr(target_size(detect, :))];

    % Draw bounding box.
    Imf = insertShape(Im, 'Rectangle', target_roi, 'Color', 'green');
    % Plot normalized cross correlation.
```

```matlab
        videopatternplots('update',hPlot,norm_Corr_value);
        step(hROIPattern, Imf);
end

snapnow
release(hVideoSrc);

% Function to compute pyramid image at a particular level.
function outI = multilevelPyramid(inI, level)

I = inI;
outI = I;

for i=1:level
    outI = impyramid(I, 'reduce');
    I = outI;
end

end
```
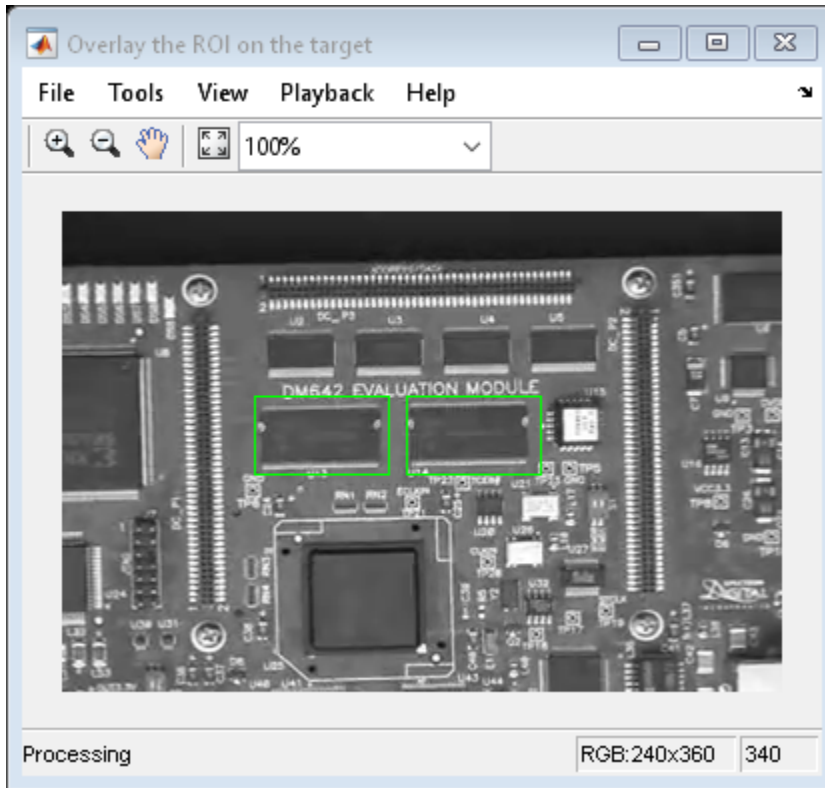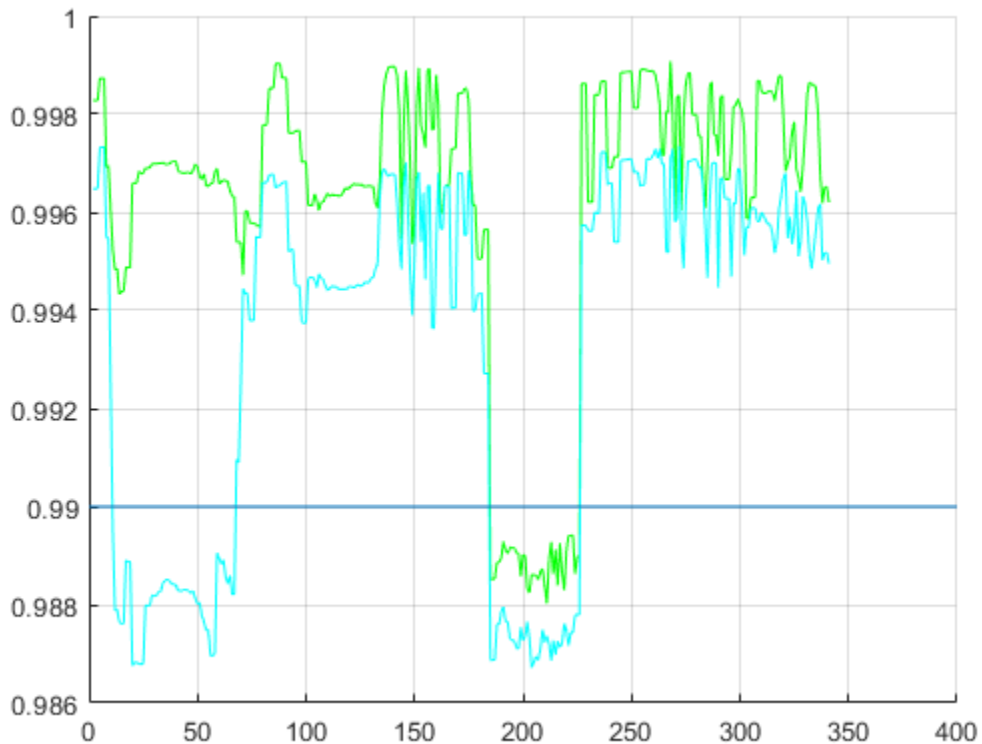
Summary

This example shows use of Computer Vision System Toolbox™ to find a user defined pattern in a video and track it. The algorithm is based on normalized frequency domain cross correlation between the target and the image under test. The video player window displays the input video with the identified target locations. Also a figure displays the normalized correlation between the target and the image which is used as a metric to match the target. As can be seen whenever the correlation value exceeds the threshold (indicated by the blue line), the target is identified in the input video and the location is marked by the green bounding box.

Appendix

The following helper functions are used in this example.

- videopattern_gettemplate.m
- videopatternplots.m

Pattern Matching

This example shows how to use the 2-D normalized cross-correlation for pattern matching and target tracking.

Double-click the Edit Parameters block to select the number of similar targets to detect. You can also change the pyramiding factor. By increasing it, you can match the target template to each video frame more quickly. Changing the pyramiding factor might require you to change the Threshold value.

Additionally, you can double-click the Correlation Method switch to specify the domain in which to perform the cross-correlation. The relative size of the target to the input video frame and the pyramiding factor determine which domain computation is faster.

Example Model

The following figure shows the Pattern Matching model:
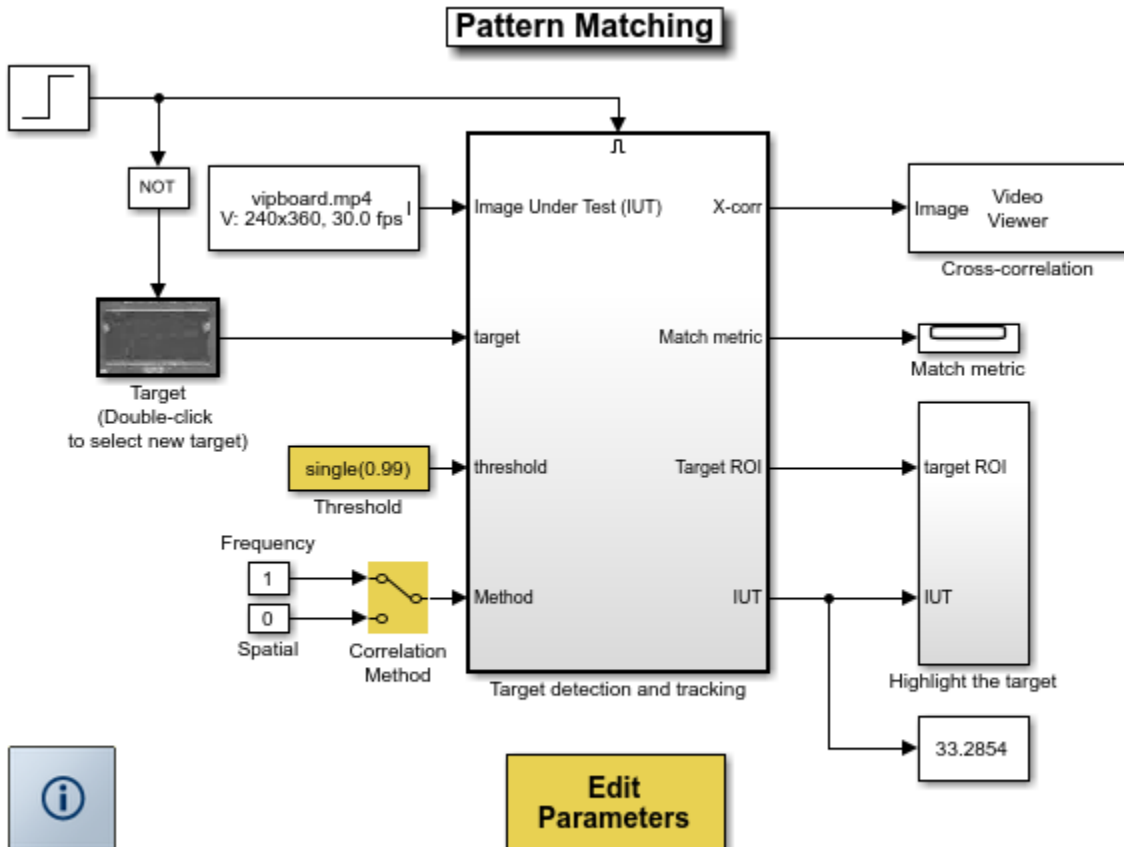
Pattern Matching

Copyright 2003-2008 The MathWorks, Inc.

Pattern Matching Results

The Match metric window shows the variation of the target match metrics. The model determines that the target template is present in a video frame when the match metric exceeds a threshold (cyan line).
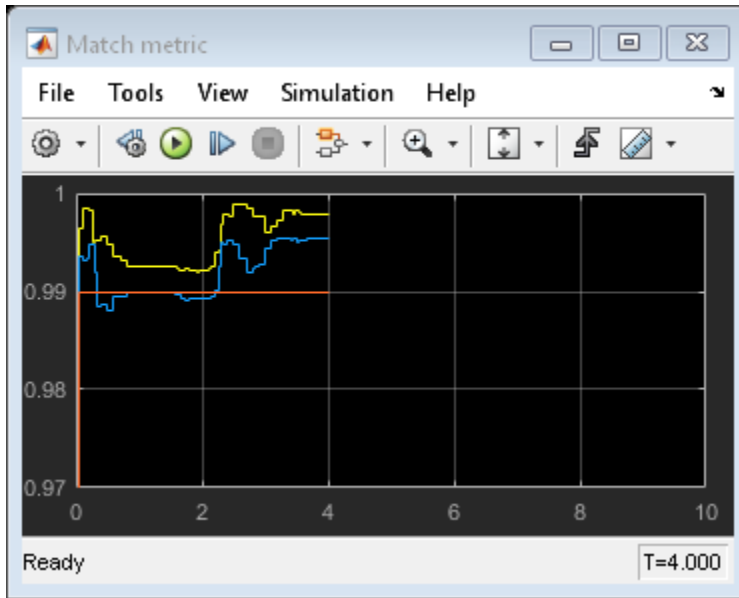
The Cross-correlation window shows the result of cross-correlating the target template with a video frame. Large values in this window correspond to the locations of the targets in the input image.
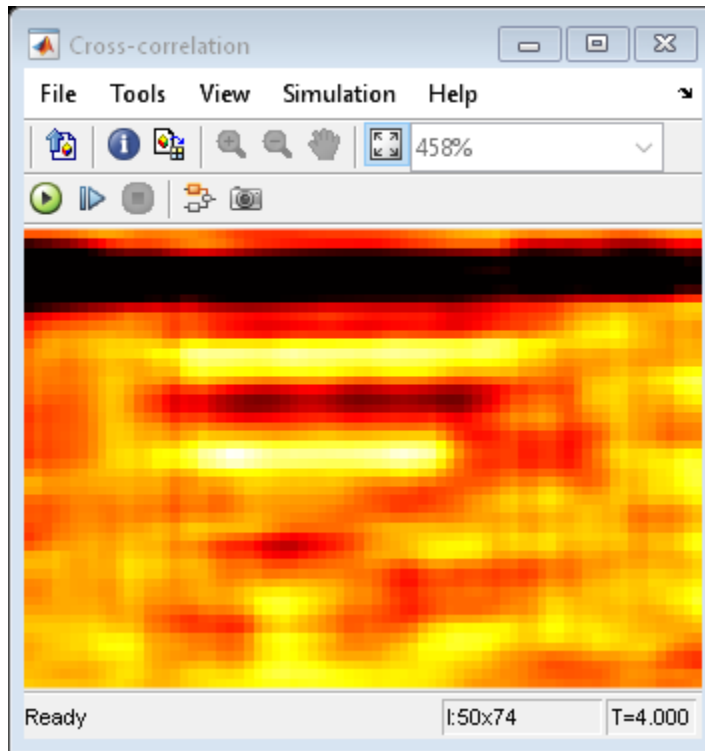
The Overlay window shows the locations of the targets by highlighting them with rectangular regions of interest (ROIs). These ROIs are present only when the targets are detected in the video frame.

9
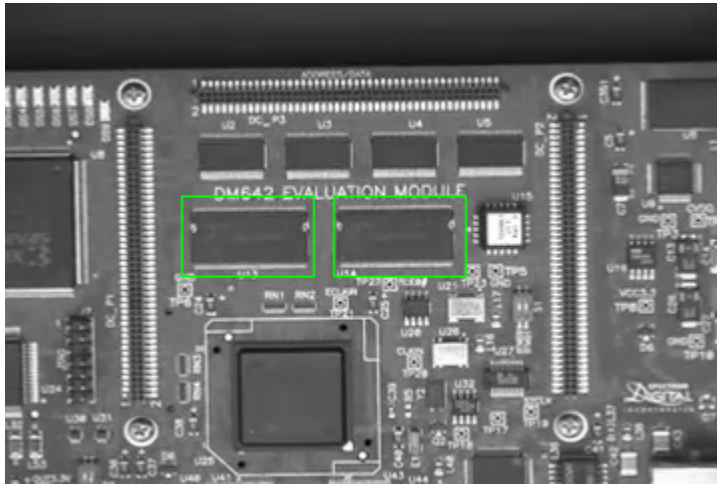
Geometric Transformations

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)
    ```
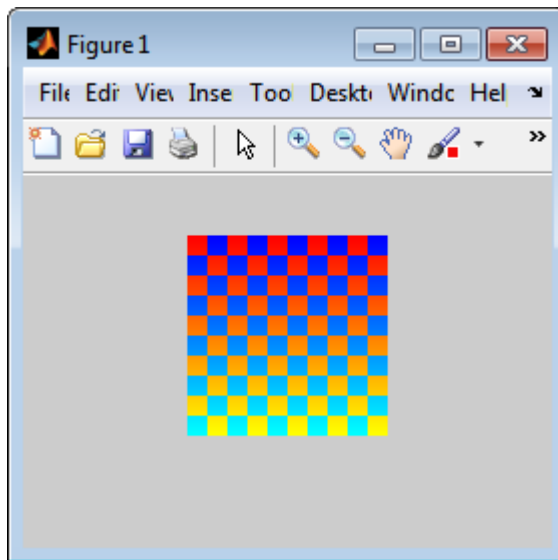

3 Create a new Simulink model, and add to it the blocks shown in the following table.

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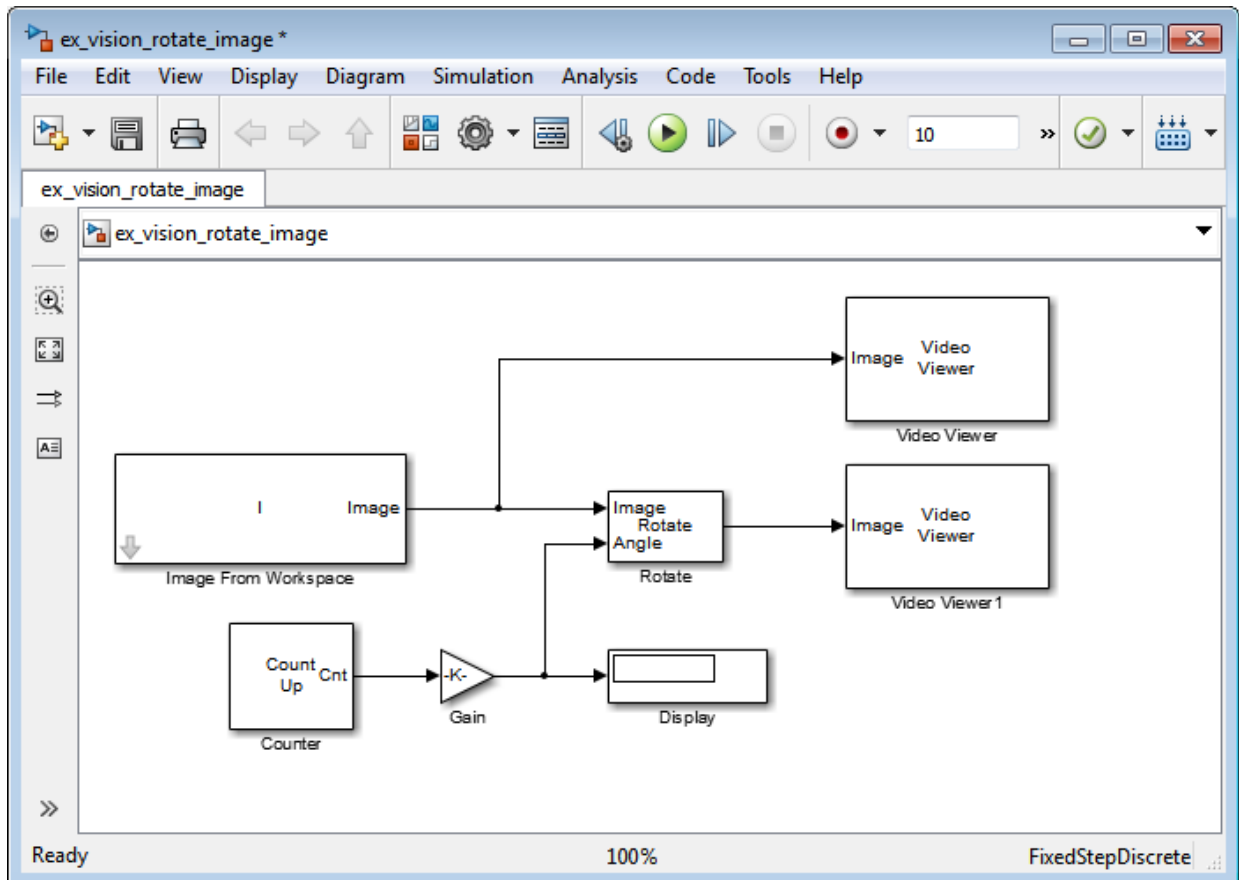
11 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **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.
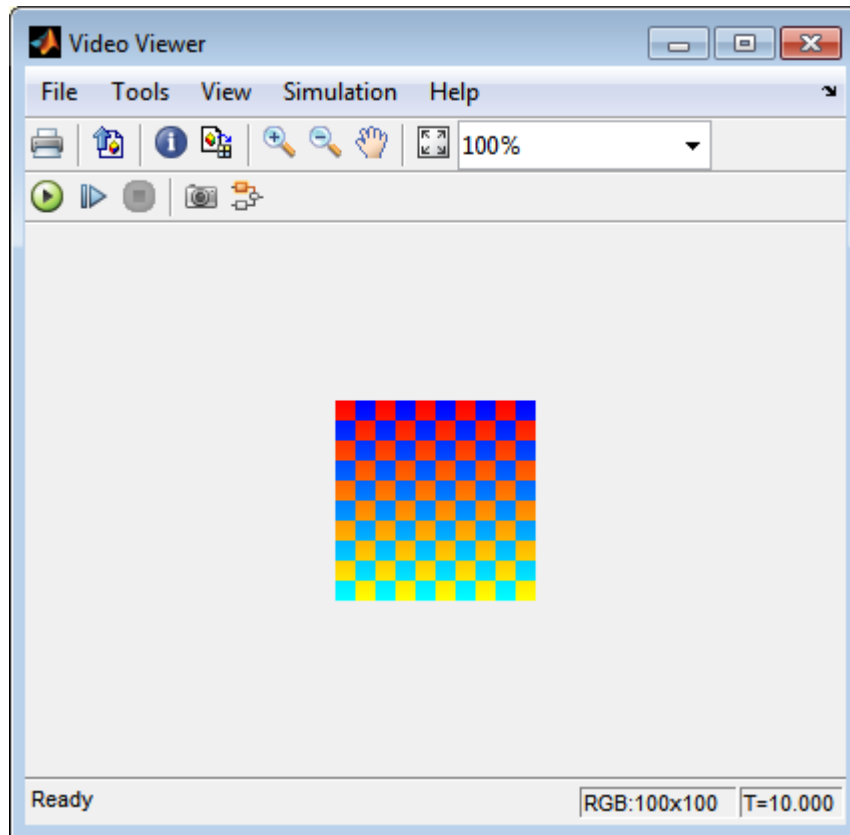
The rotating image appears in the Video Viewer1 window.
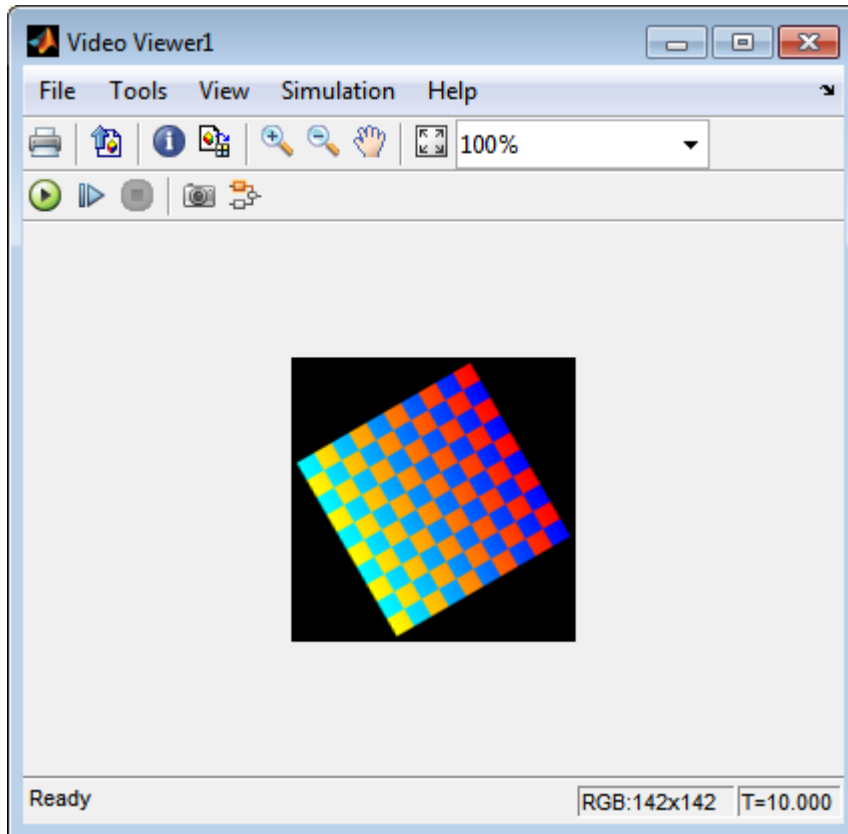
In this example, you used the Rotate block to continuously rotate your image. For more information about this block, see the Rotate block reference page in the *Computer Vision System Toolbox Reference*. For more information about other geometric transformation blocks, see the Resize and Shear block 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.
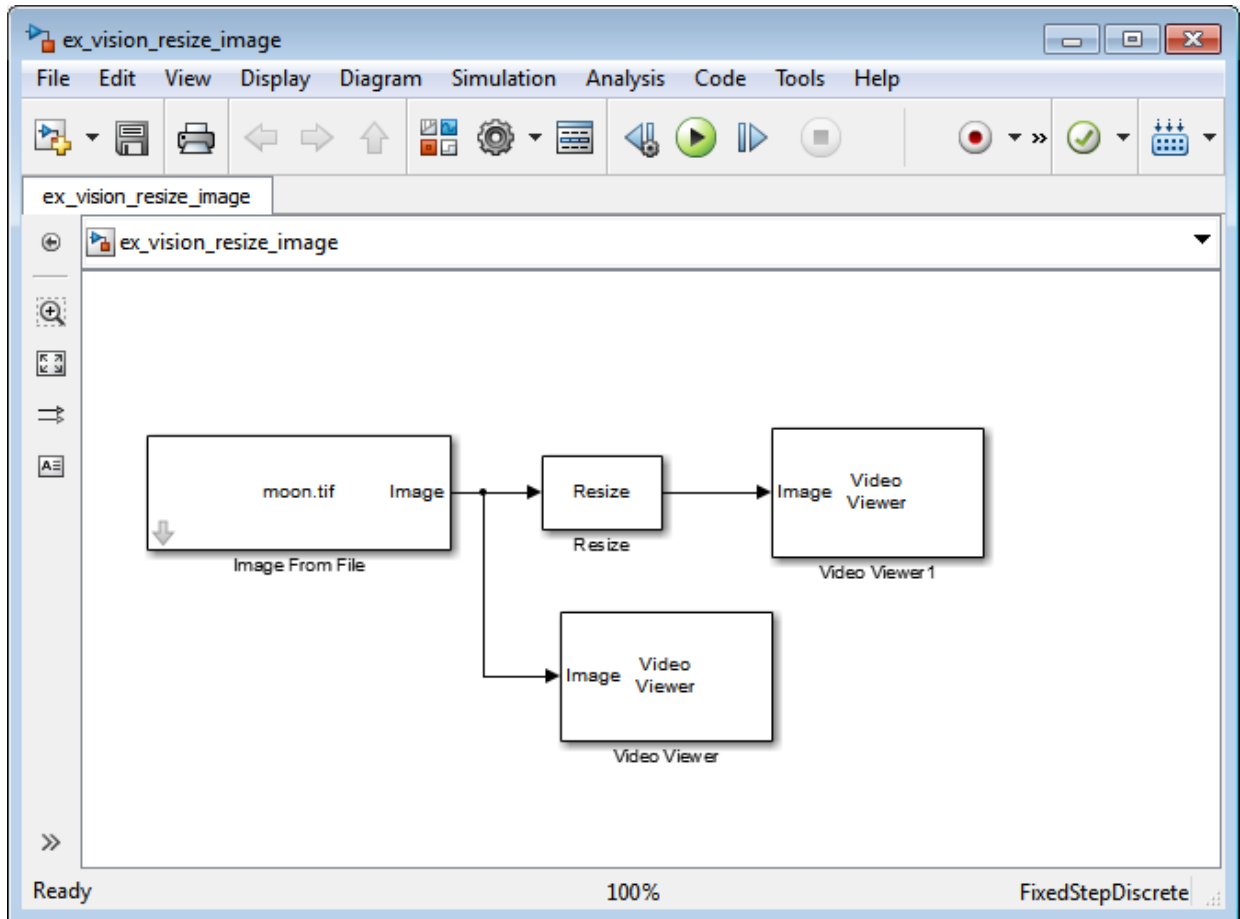
7 Set the configuration parameters. Open the Configuration dialog box by selecting
   Model Configuration Parameters from the **Simulation** menu. Set the parameters
   as follows:

   - **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.
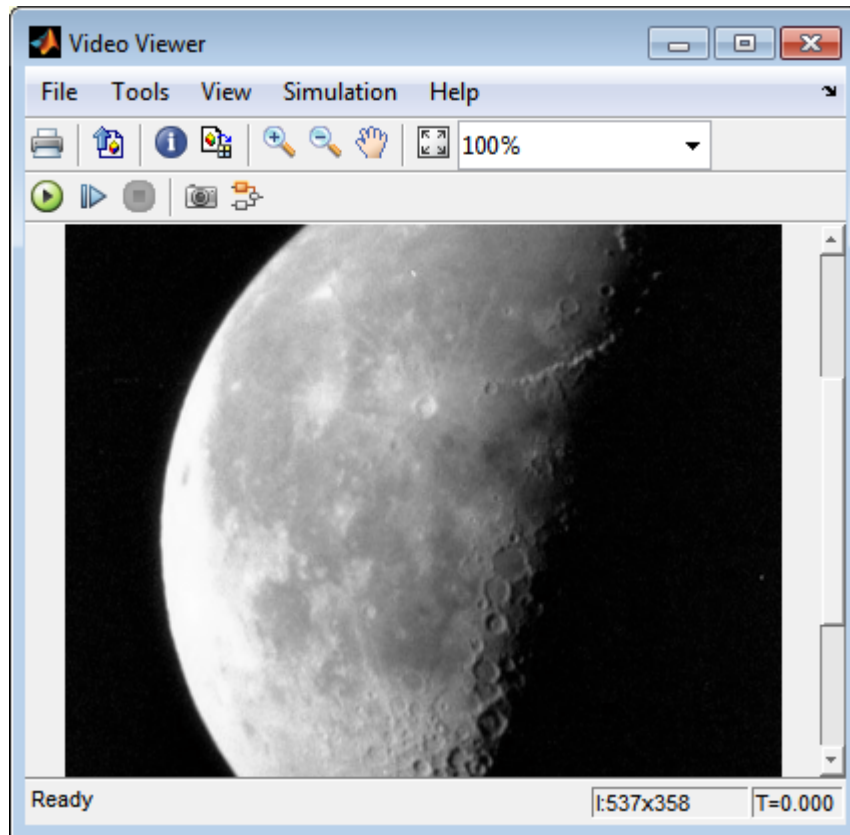
The reduced image appears in the Video Viewer1 window.
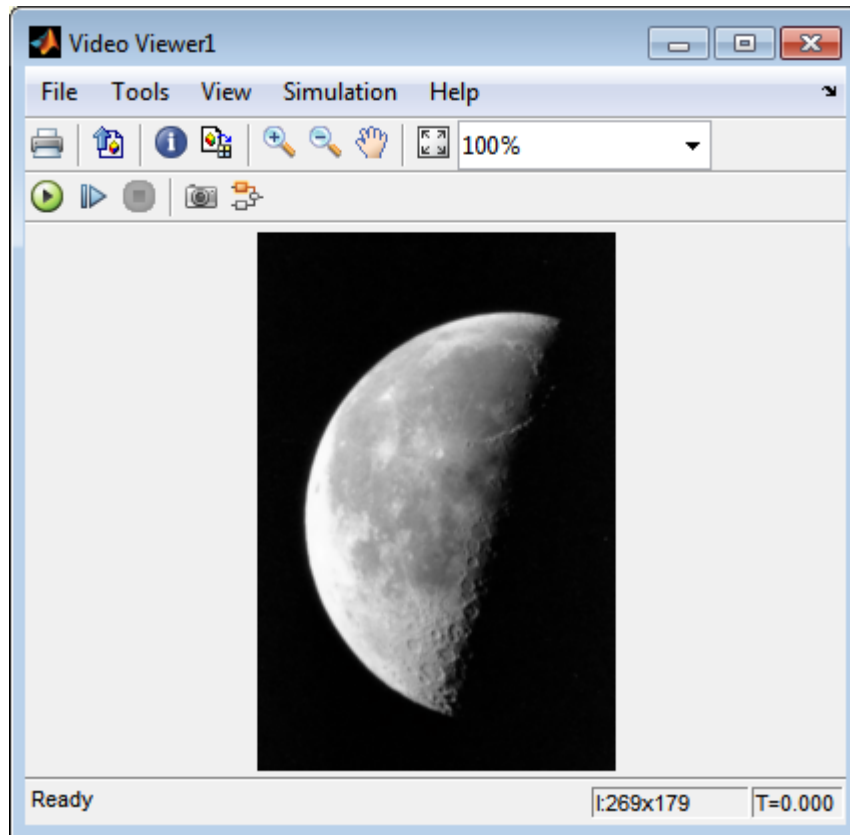
In this example, you used the Resize block to shrink an image. For more information about this block, see the Resize block reference page. For more information about other geometric transformation blocks, see the Rotate, Warp, Estimate Geometric Transformation, and Translate block reference pages.

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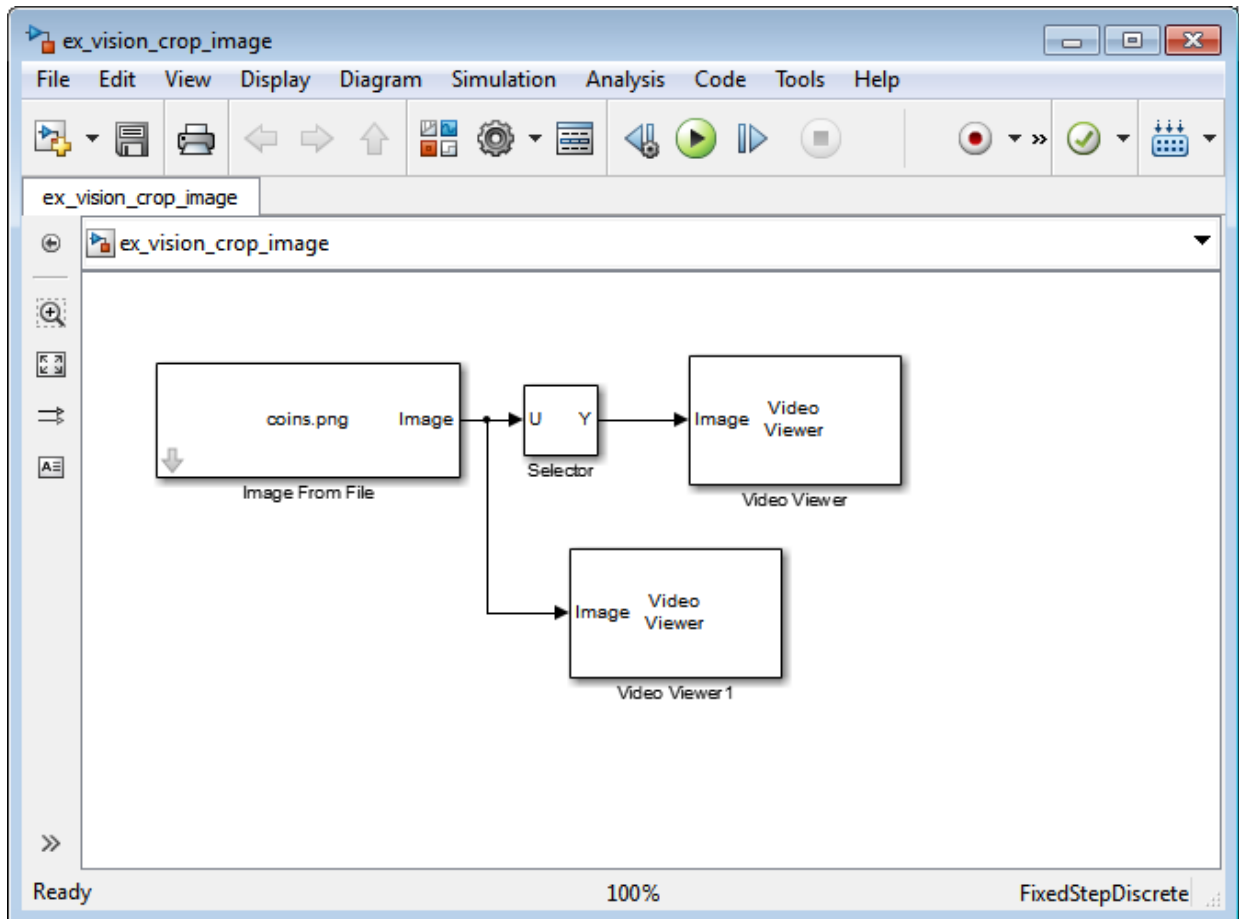
7 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **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.
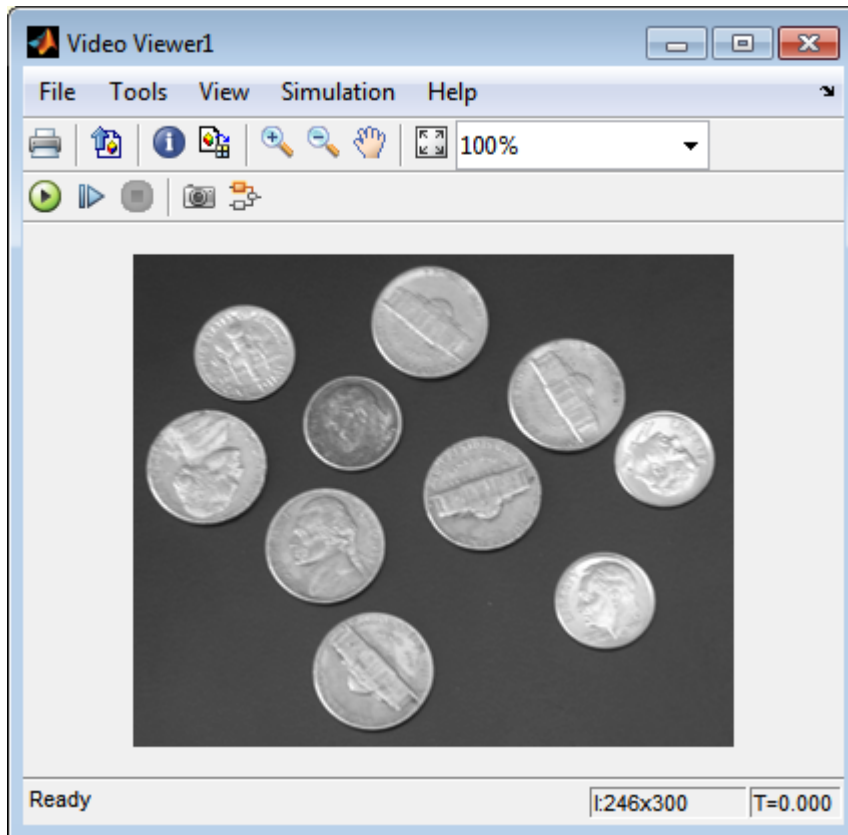
The cropped image appears in the Video Viewer1 window. The following image is shown at its true size.
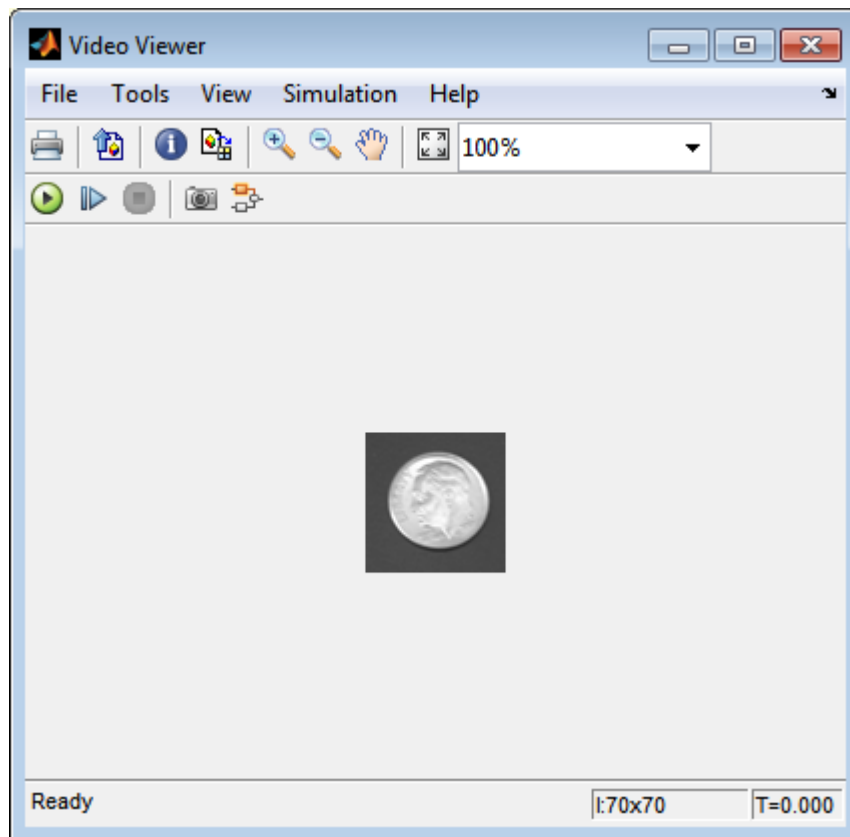
In this example, you used the Selector block to crop an image. For more information about the Selector block, see the Simulink documentation. For information about the `imcrop` function, see the Image Processing Toolbox documentation.

Nearest Neighbor, Bilinear, and Bicubic Interpolation Methods

In this section...
"Nearest Neighbor Interpolation" on page 9-16
"Bilinear Interpolation" on page 9-17
"Bicubic Interpolation" on page 9-18

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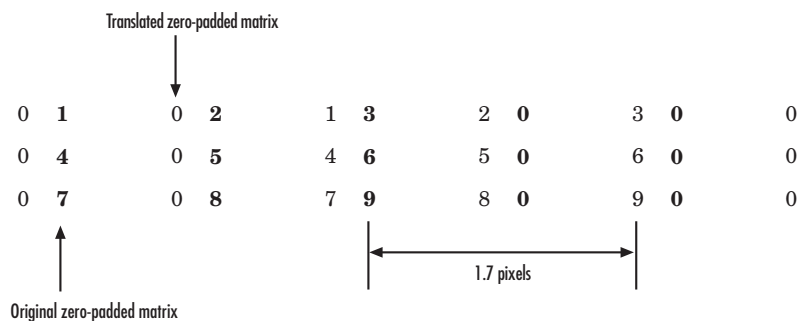
2 Create the output matrix by replacing each input pixel value with the translated value nearest to it. The result is the following matrix:

```
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.

Translated zero-padded matrix

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

Original zero-padded matrix   0.5 pixel
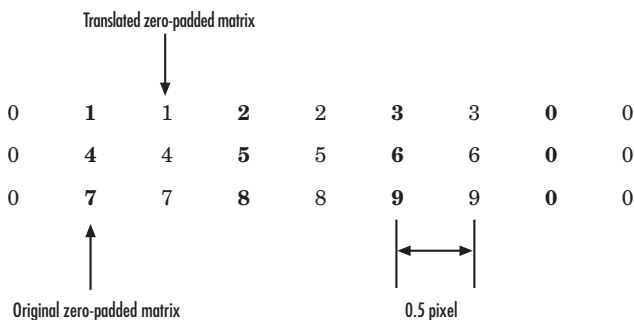
2 Create the output matrix by replacing each input pixel value with the weighted average of the translated values on either side. The result is the following matrix where the output matrix has one more column than the input matrix:

```
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.

Translated zero-padded matrix

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

Original zero-padded matrix

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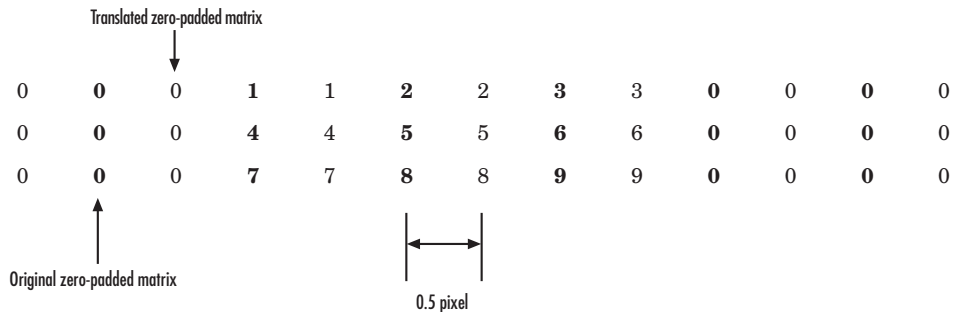
2 Create the output matrix by replacing each input pixel value with the weighted average of the two translated values on either side. The result is the following matrix where the output matrix has one more column than the input matrix:

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

Filters, Transforms, and Enhancements

- "Adjust the Contrast of Intensity Images" on page 10-2
- "Adjust the Contrast of Color Images" on page 10-6
- "Remove Salt and Pepper Noise from Images" on page 10-11
- "Sharpen an Image" on page 10-16

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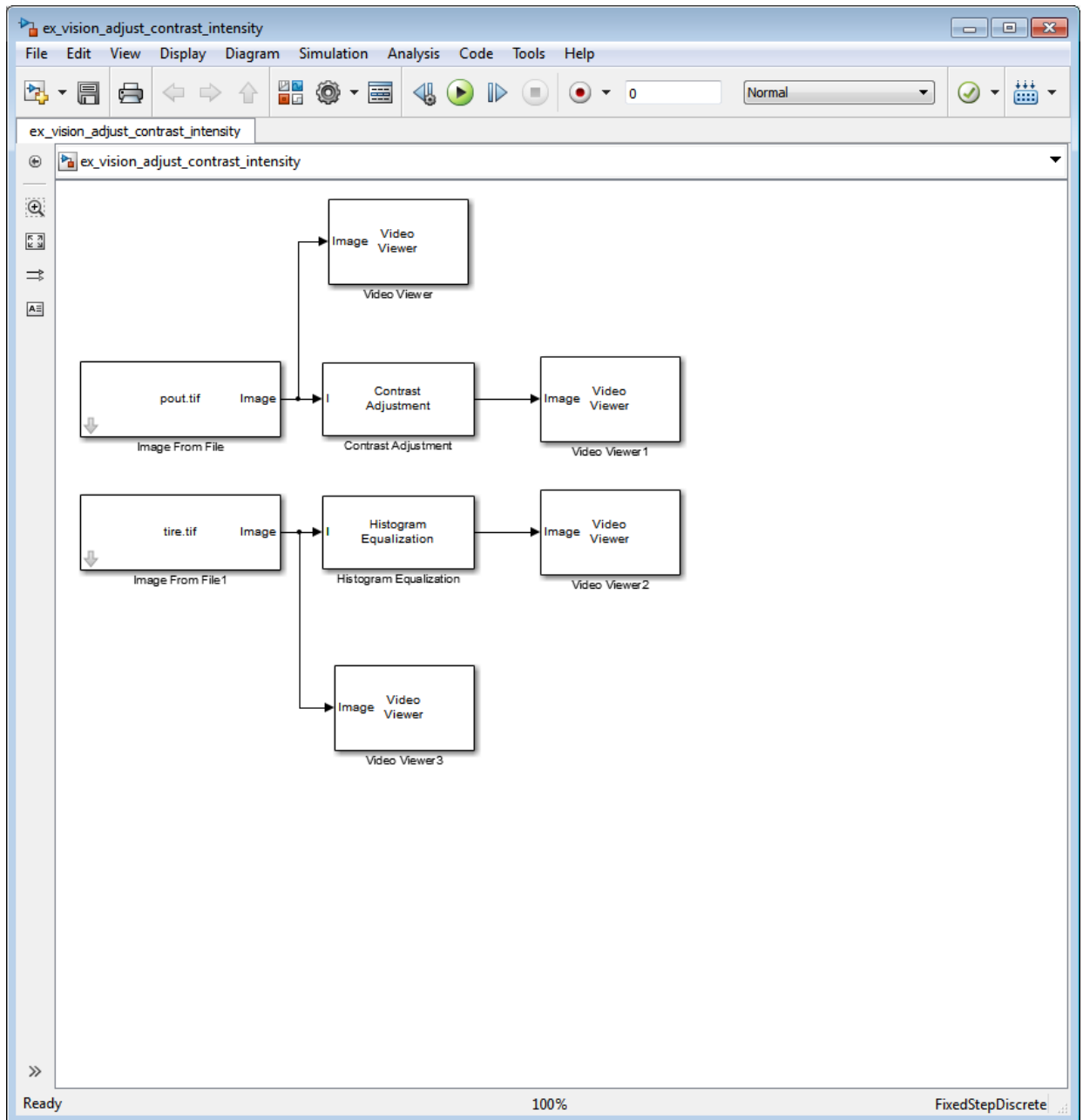
9 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **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.
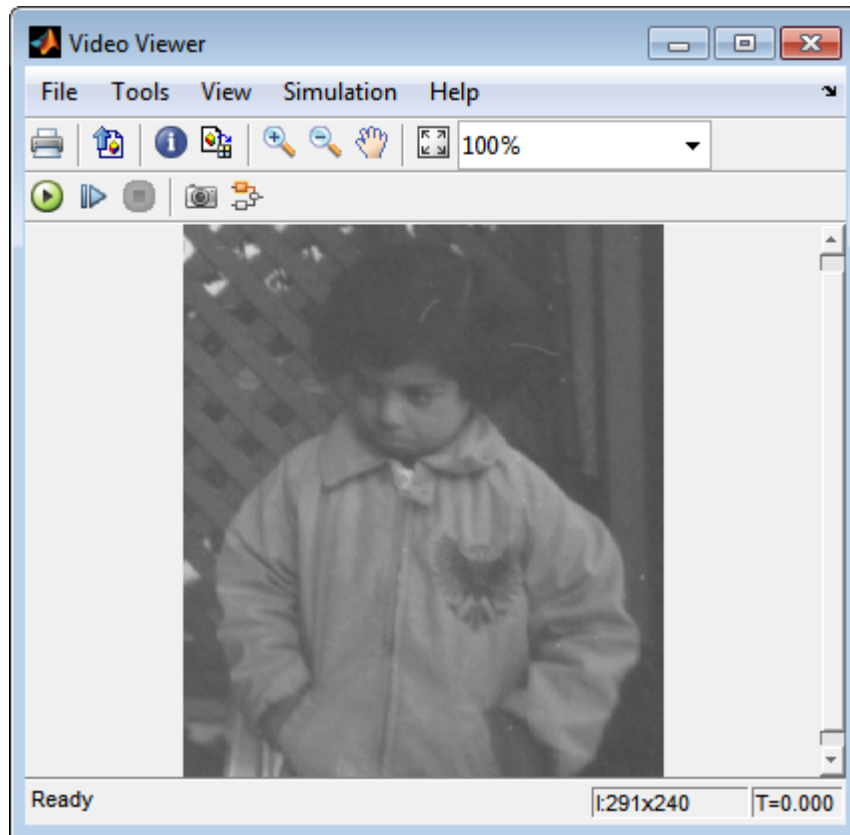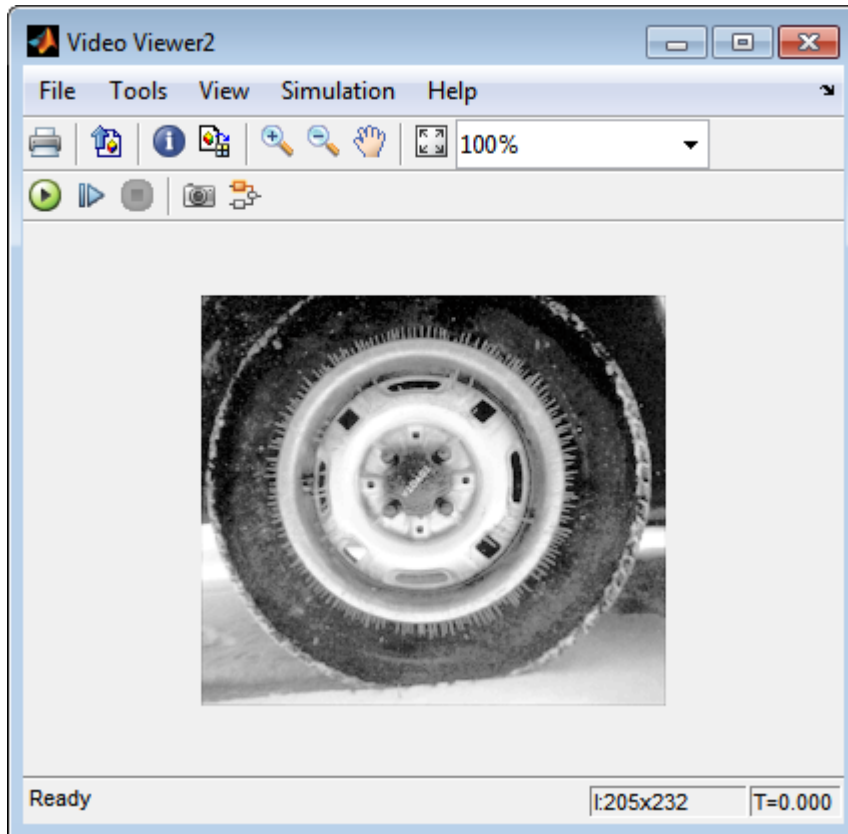
In this example, you used the Contrast Adjustment block to linearly scale the pixel values in `pout.tif` between new upper and lower limits. You used the Histogram Equalization block to transform the values in `tire.tif` so that the histogram of the output image approximately matches a uniform histogram. 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.
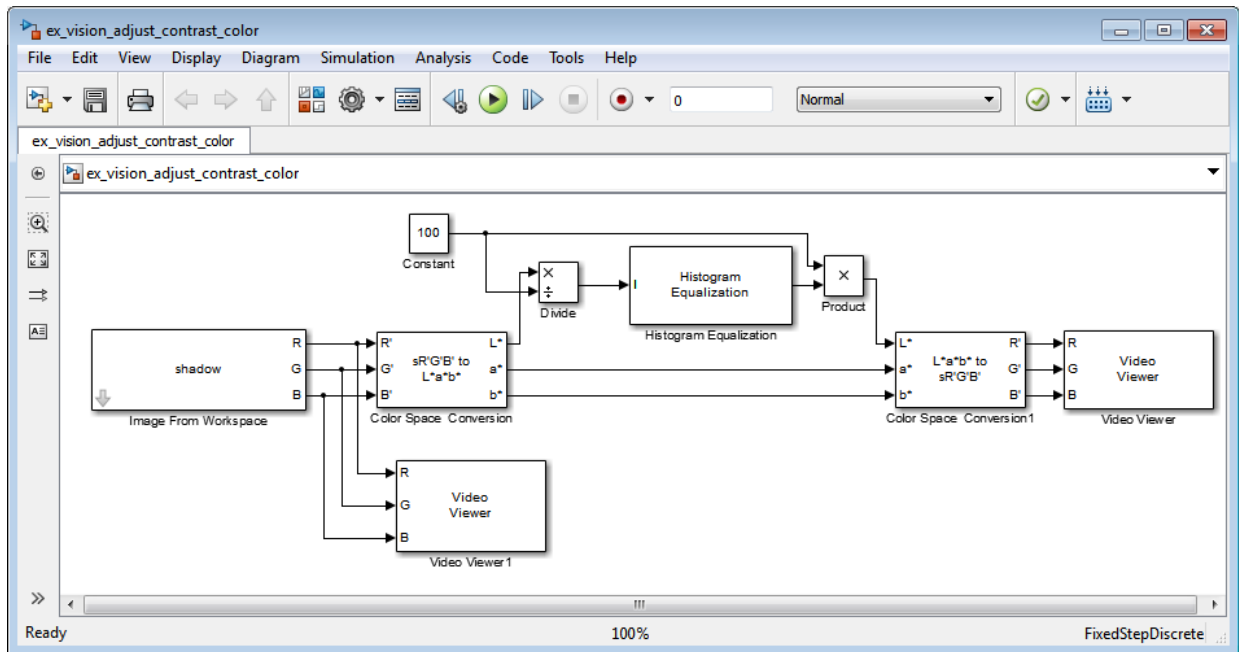
13 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

- **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.
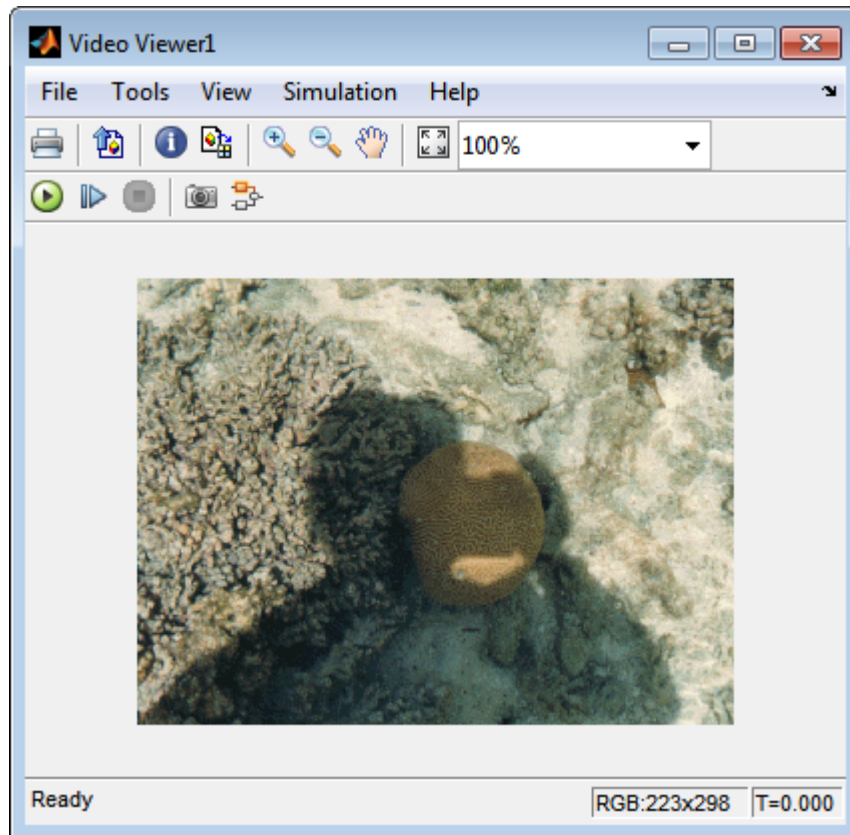
As the next figure shows, the model displays the enhanced contrast image in the Video Viewer window.
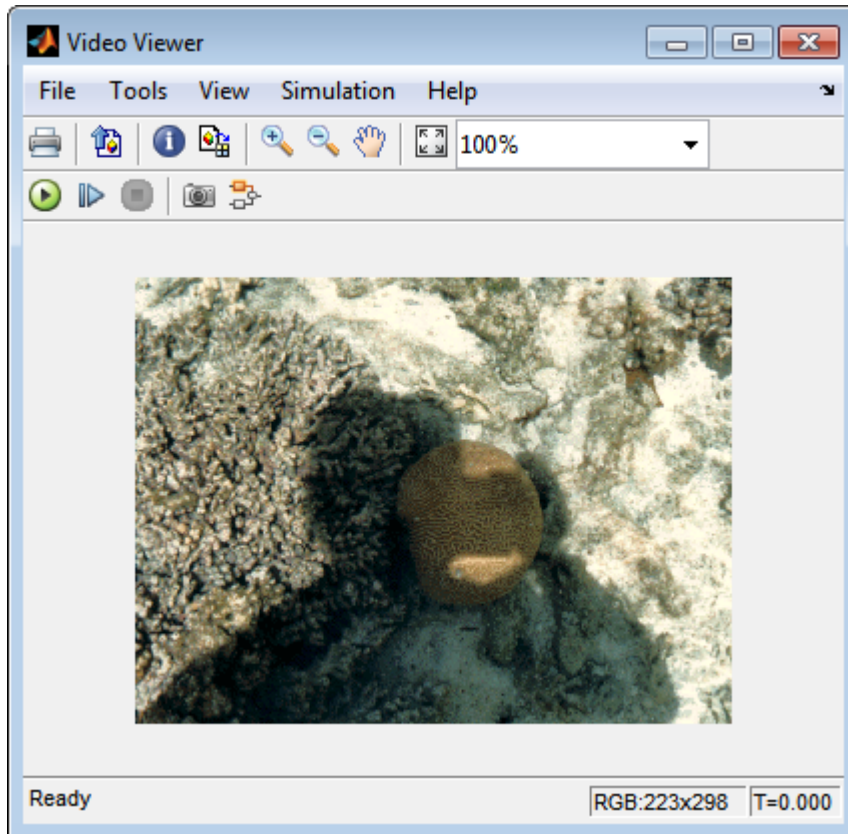
In this example, you used the Histogram Equalization block to transform the values in a color image so that the histogram of the output image approximately matches a uniform histogram. For more information, see the Histogram Equalization reference page.

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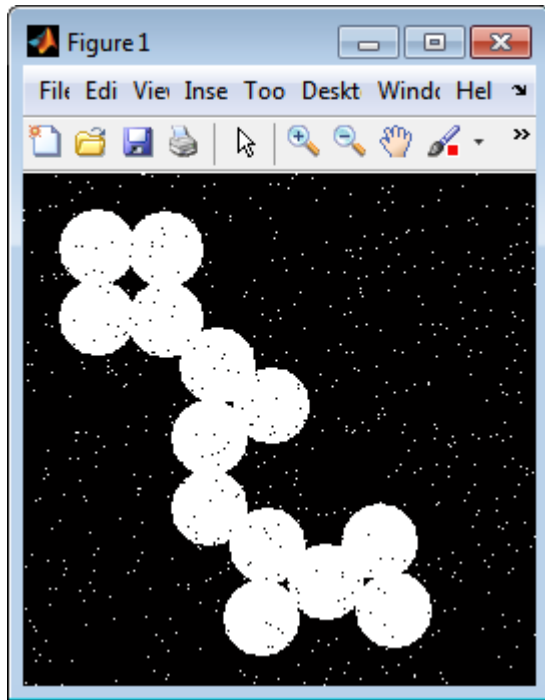
The intensity image contains noise that you want your model to eliminate.

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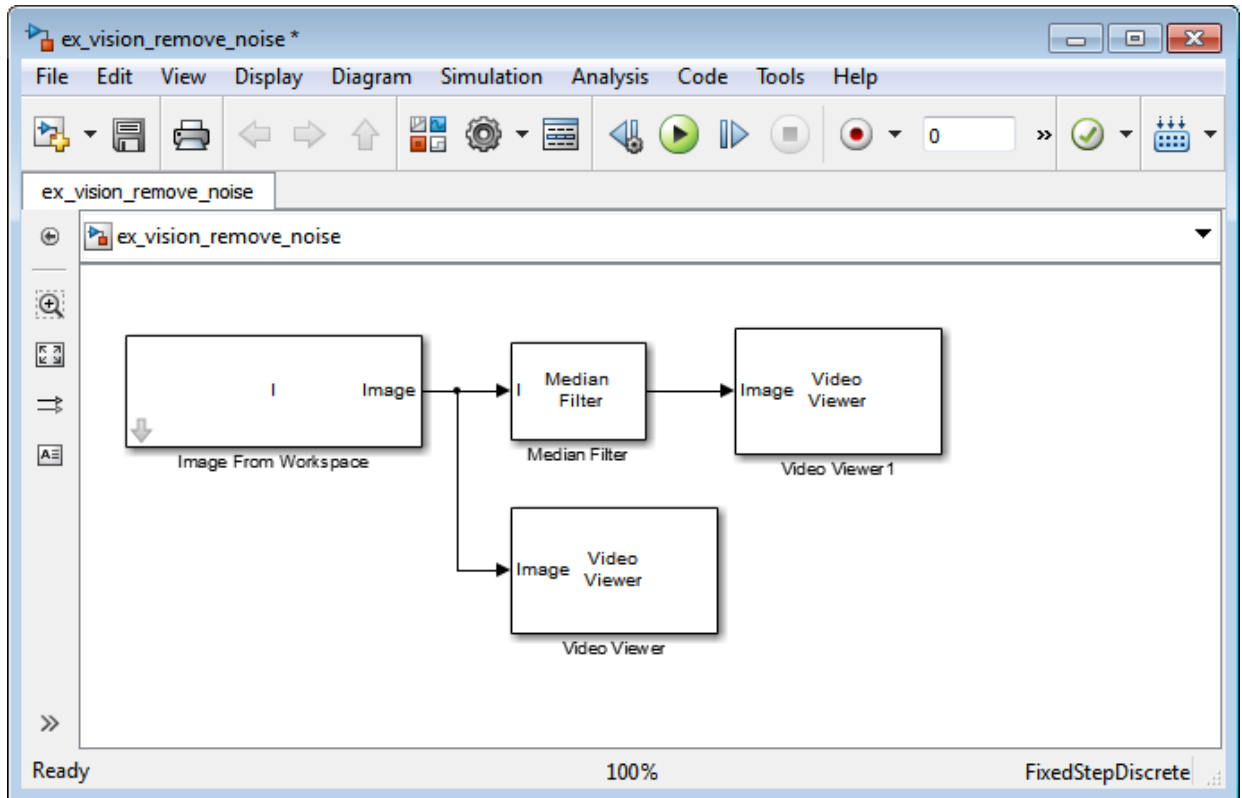
**7** Connect the blocks as shown in the following figure.

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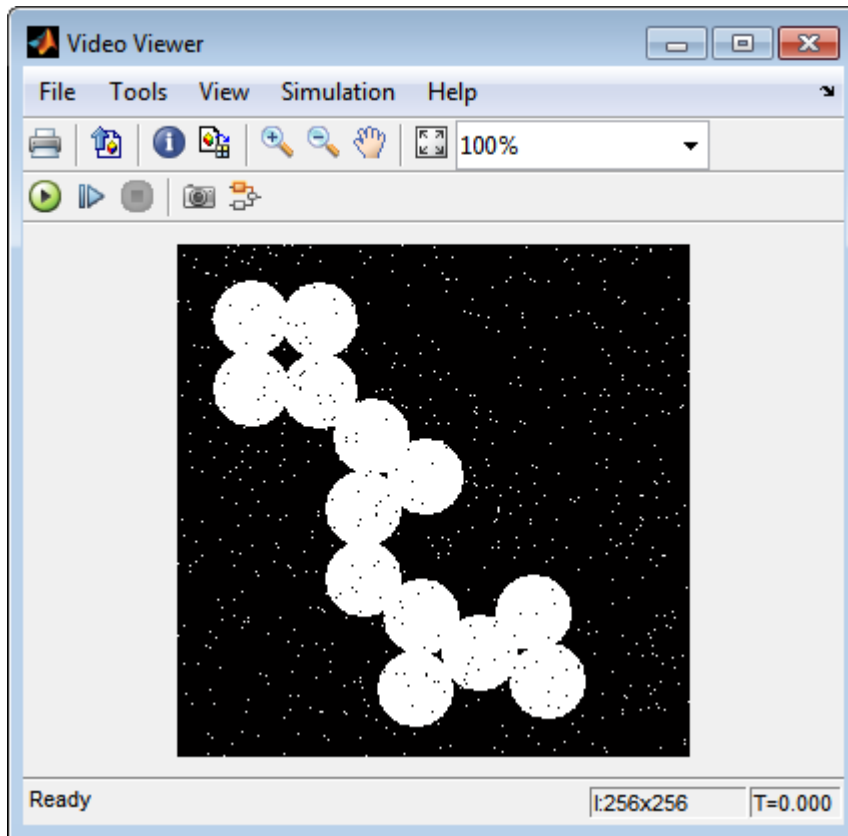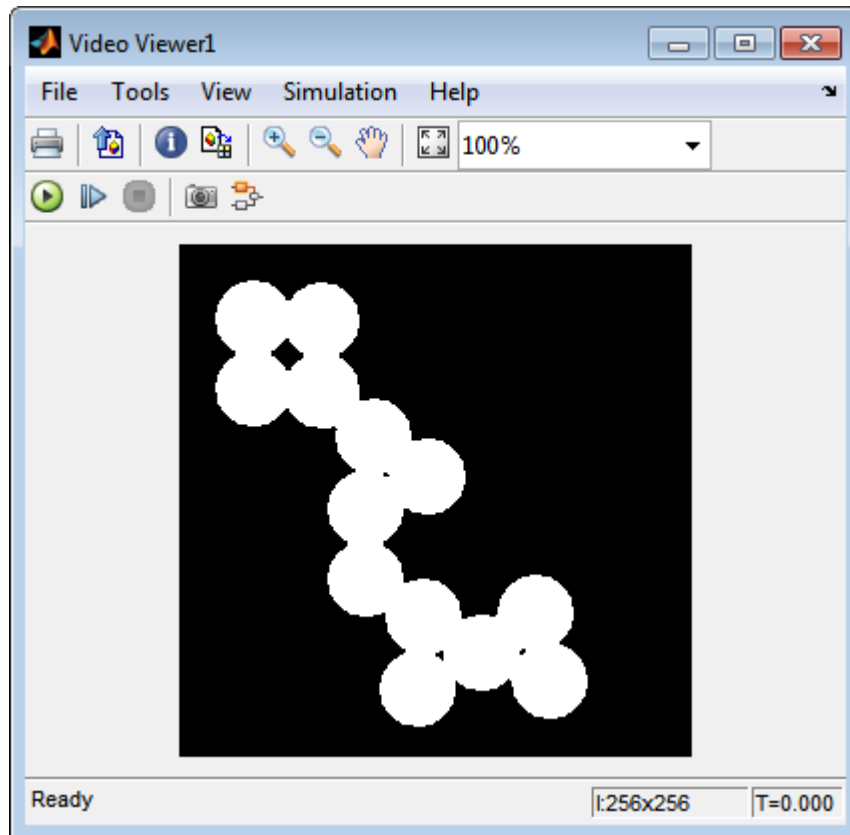
You have used the Median Filter block to remove noise from your image. For more information about this block, see the Median Filter block reference page in the *Computer Vision System Toolbox Reference*.

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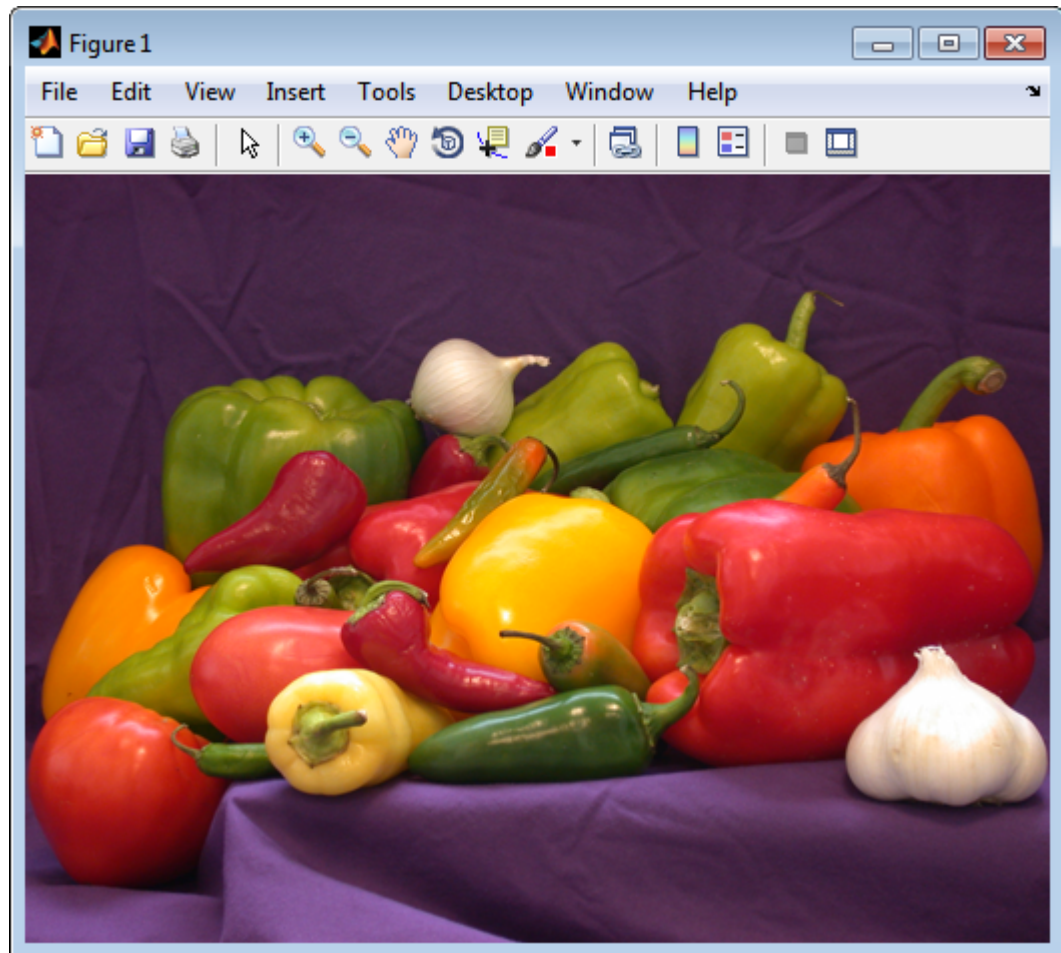
Now that you have defined your image, you can create your model.

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

10-17

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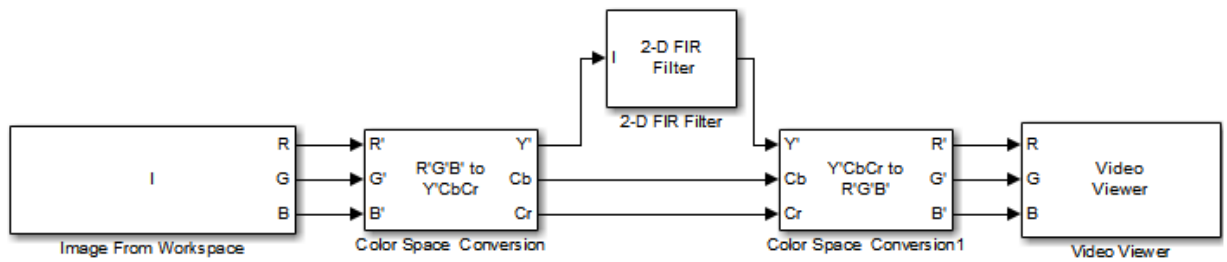
10 Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

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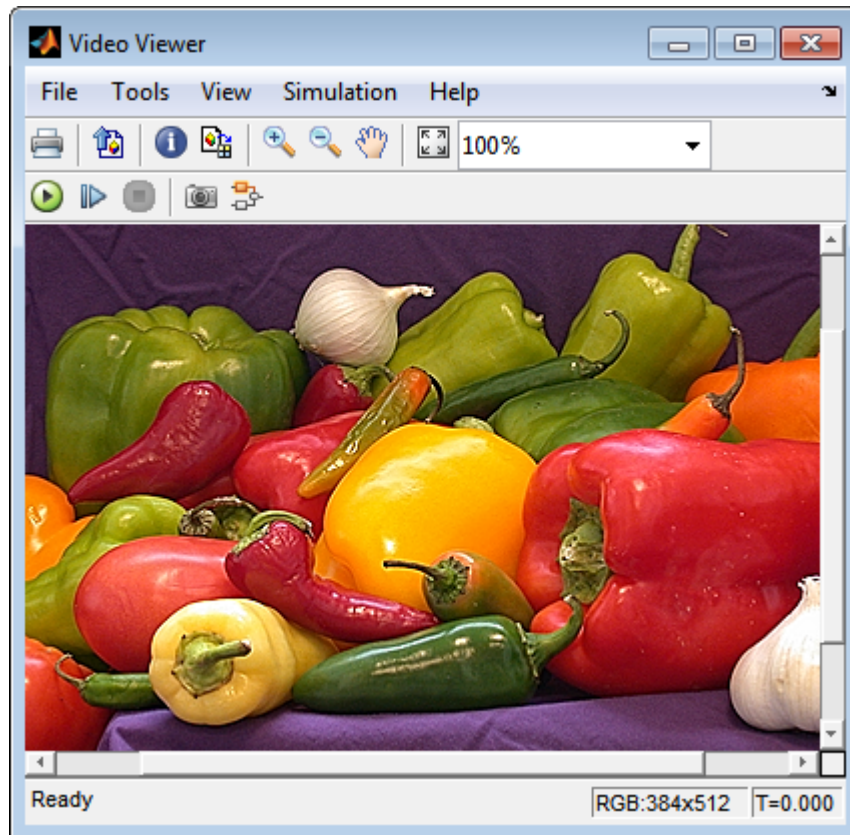
To blur the image, double-click the 2-D FIR Filter block. Set **Coefficients** parameter to `fspecial('gaussian',[15 15],7)` and then click **OK**. The `fspecial('gaussian',[15 15],7)` command creates two-dimensional Gaussian lowpass filter coefficients. This lowpass filter blurs the image by removing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

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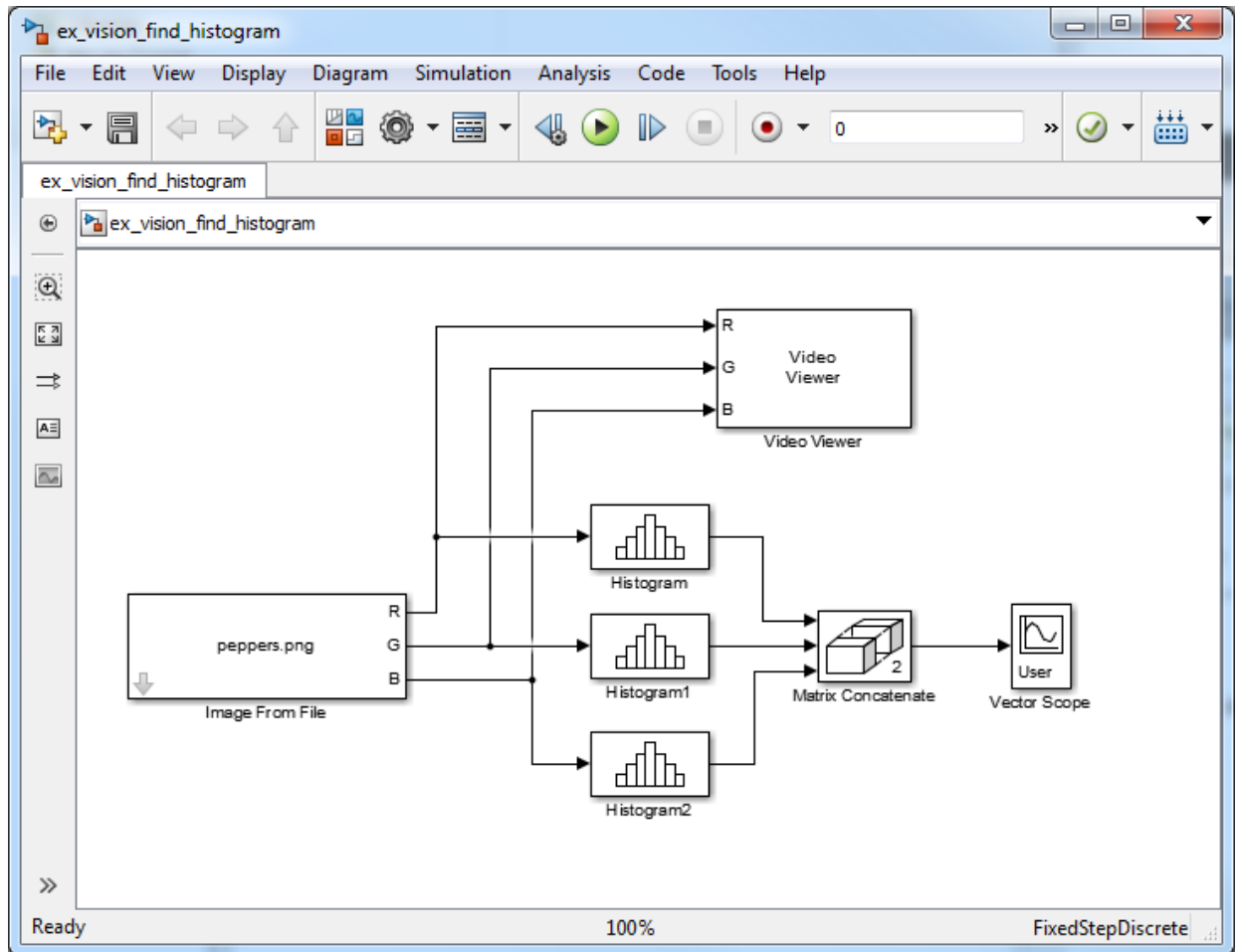
8   Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

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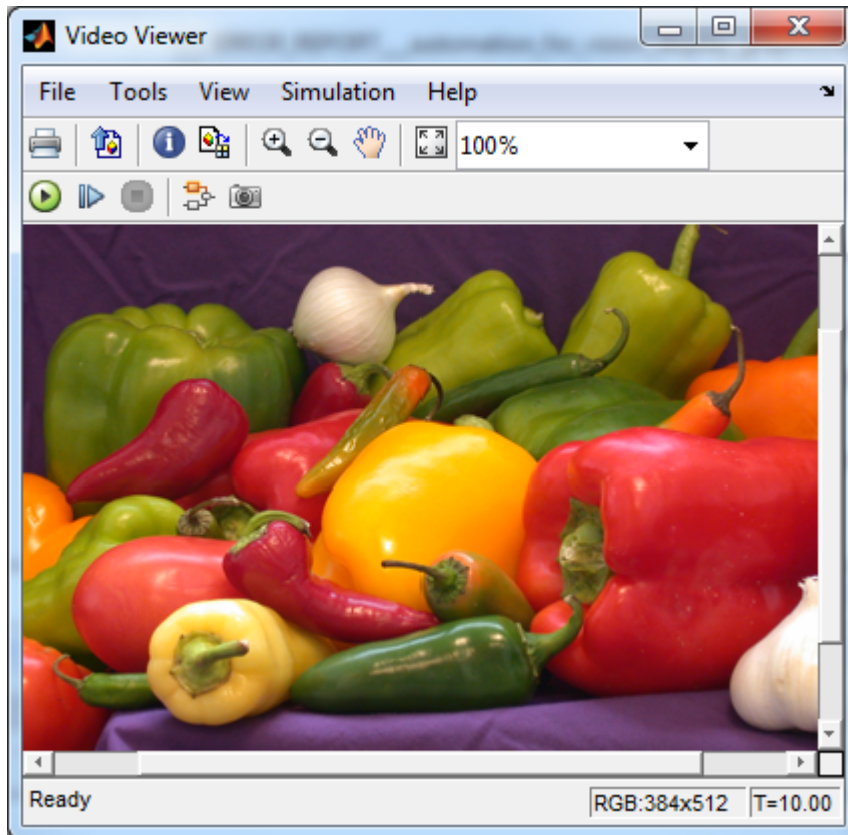
10 Right-click in the Vector Scope window and select **Autoscale**.

The scaled histogram of the image appears in the Vector Scope window.
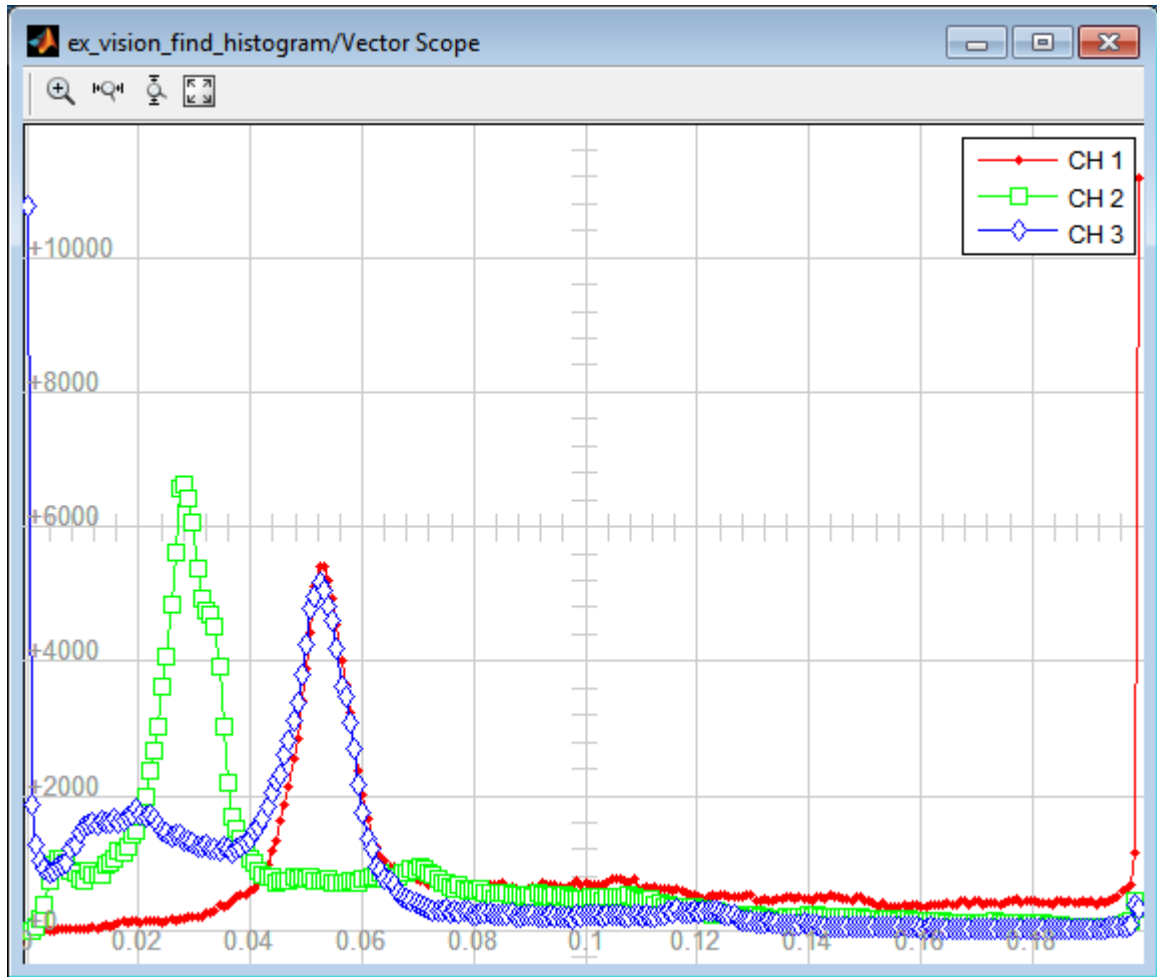
You have now used the 2-D Histogram block to calculate the histogram of the R, G, and B values in an RGB image. To open a model that illustrates how to use this block to calculate the histogram of the R, G, and B values in an RGB video stream, type `viphistogram` at the MATLAB command prompt.

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_uniform`

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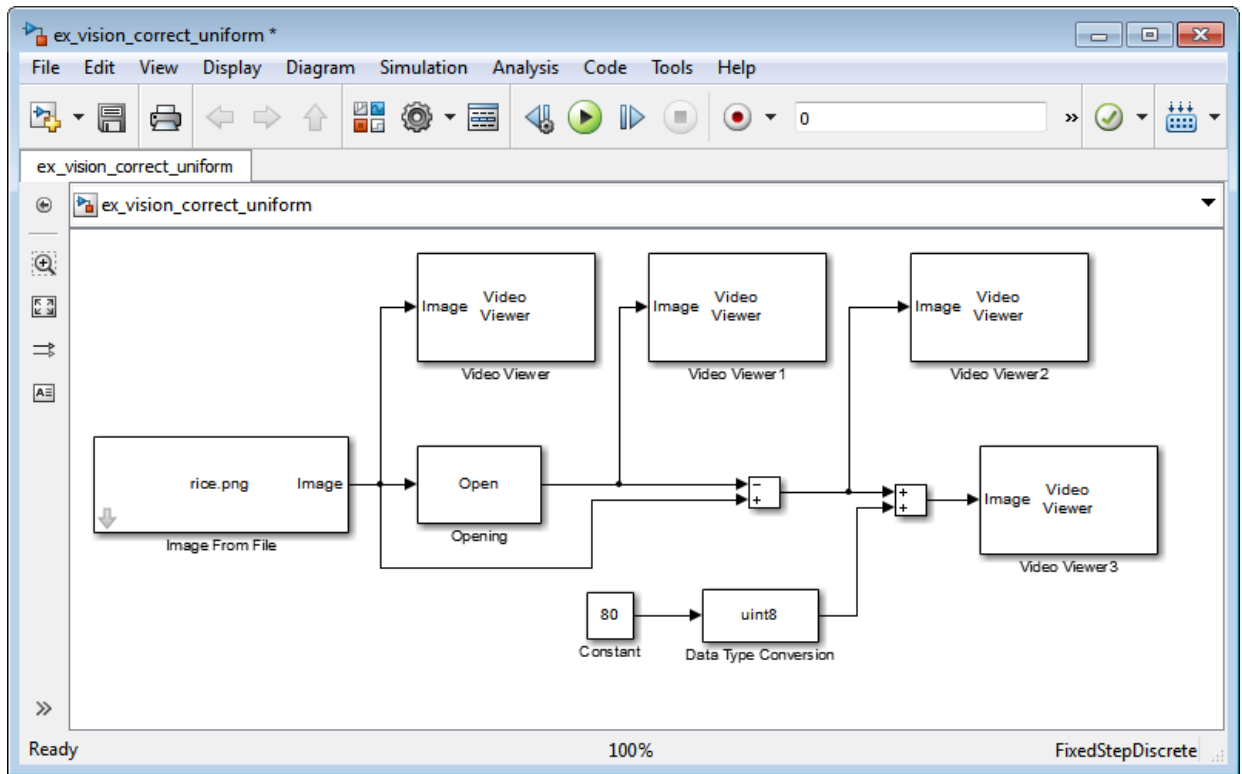
13 Set the configuration parameters. Open the Configuration dialog box by selecting
    Model Configuration Parameters from the **Simulation** menu. Set the parameters
    as follows:

    - **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.
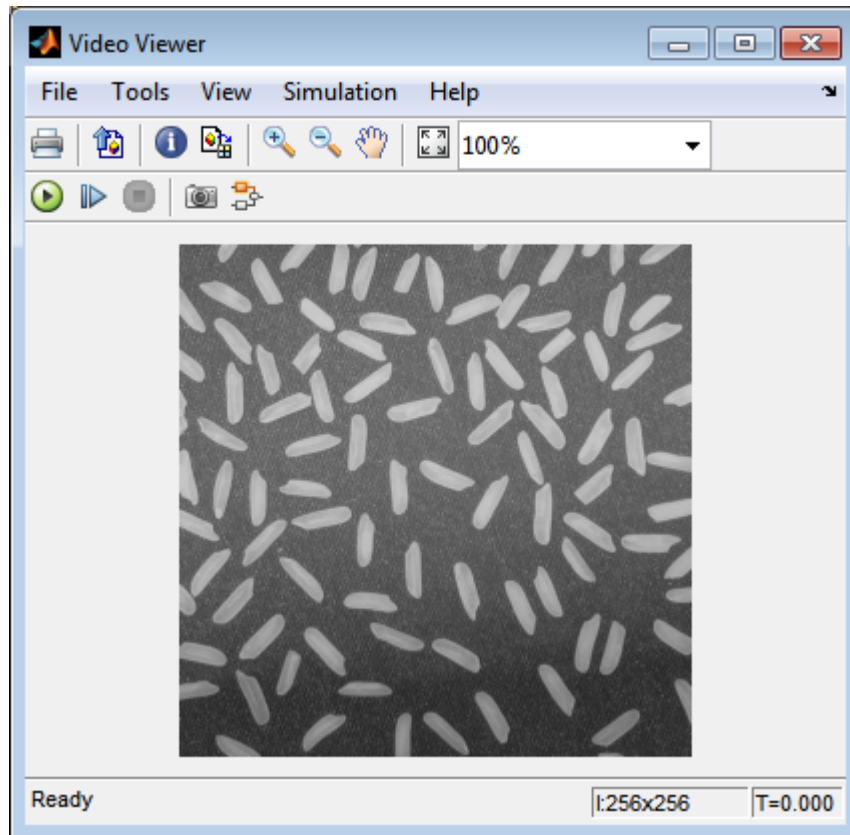
The estimated background appears in the Video Viewer1 window.
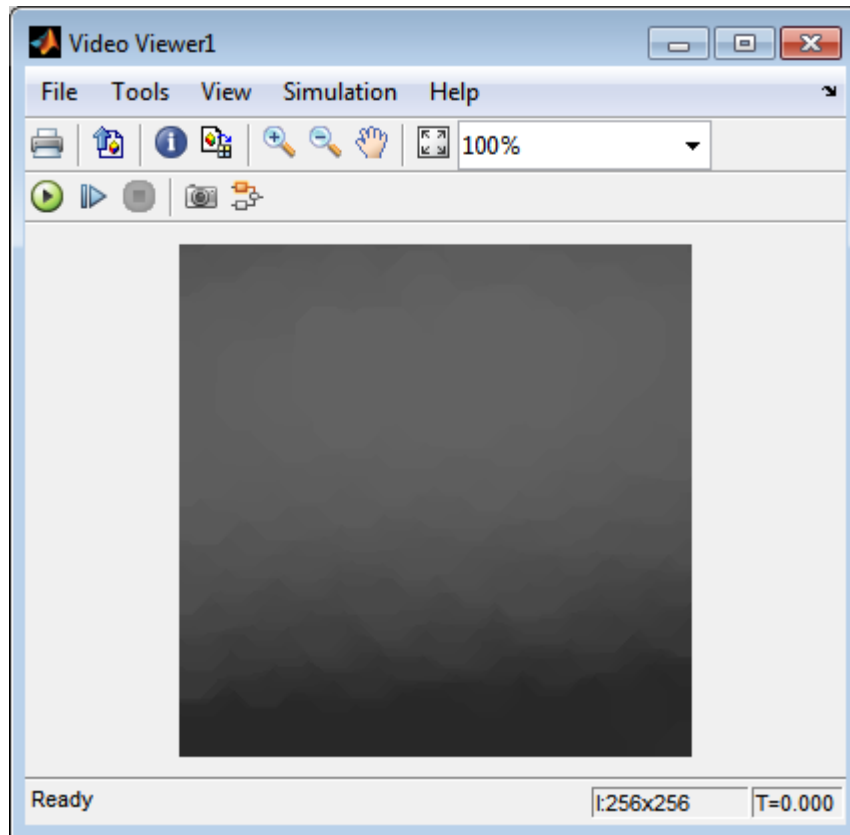
The image without the estimated background appears in the Video Viewer2 window.
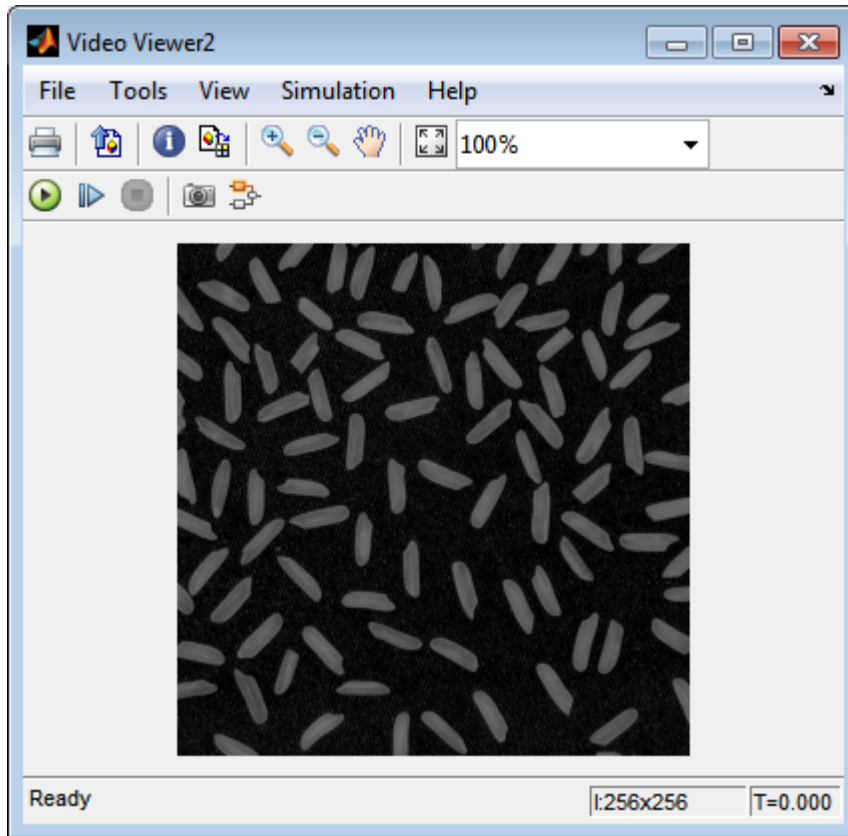
The preceding image is too dark. The Constant block provides an offset value that you used to brighten the image.

The corrected image, which has even lighting, appears in the Video Viewer3 window. The following image is shown at its true size.
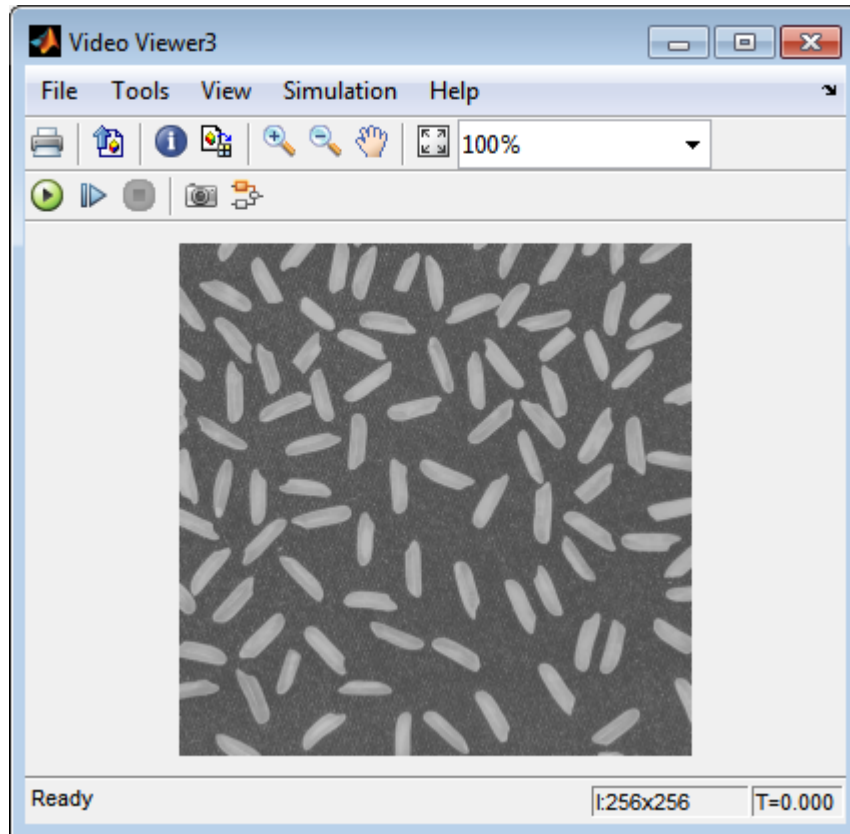
In this section, you have used the Opening block to remove irregular illumination from an image. For more information about this block, see the Opening reference page. For related information, see the Top-hat block reference page. For more information about STREL objects, see the `strel` object in the 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	Simulink > 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` object 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.
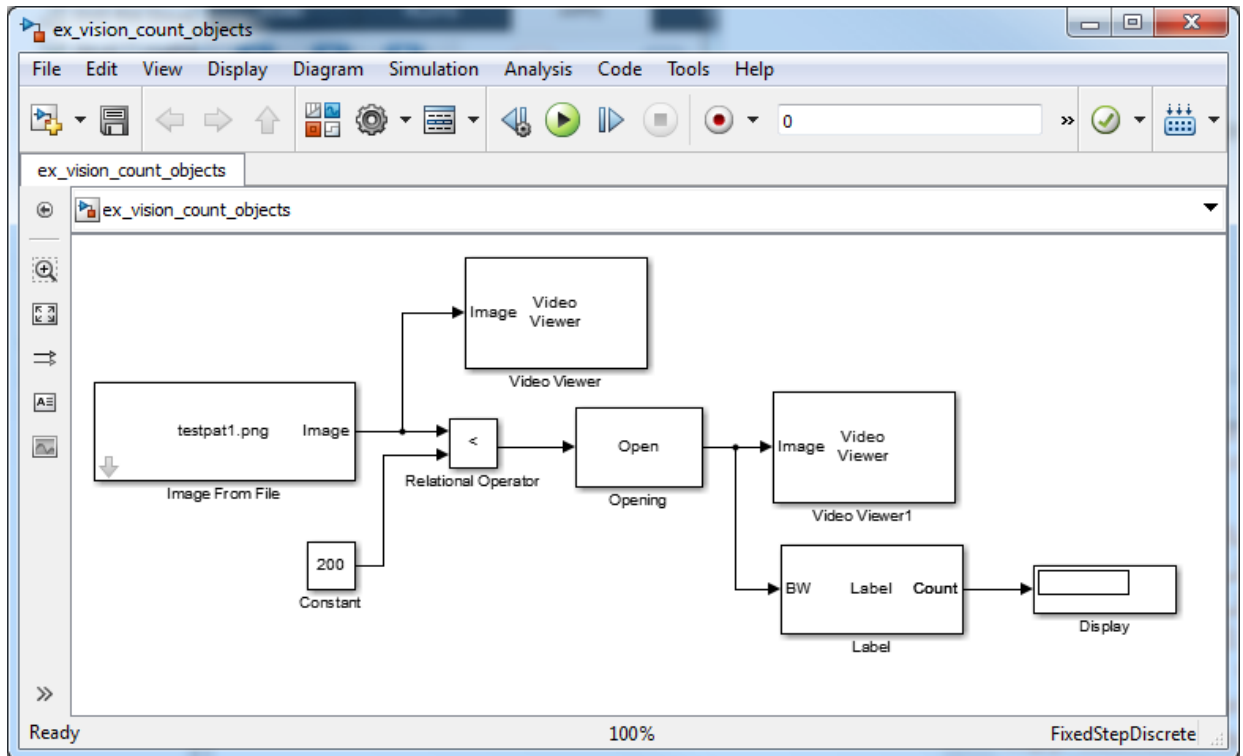
11. Set the configuration parameters. Open the Configuration dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. Set the parameters as follows:

   - **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**.
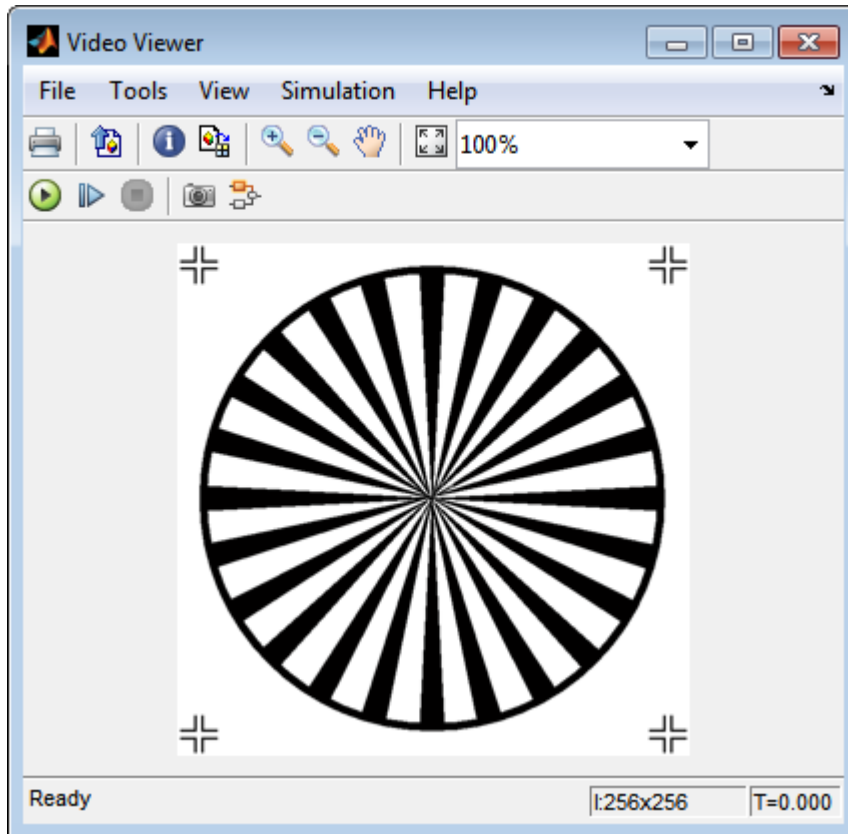
The opened image appears in the Video Viewer window. The following image is shown at its true size.
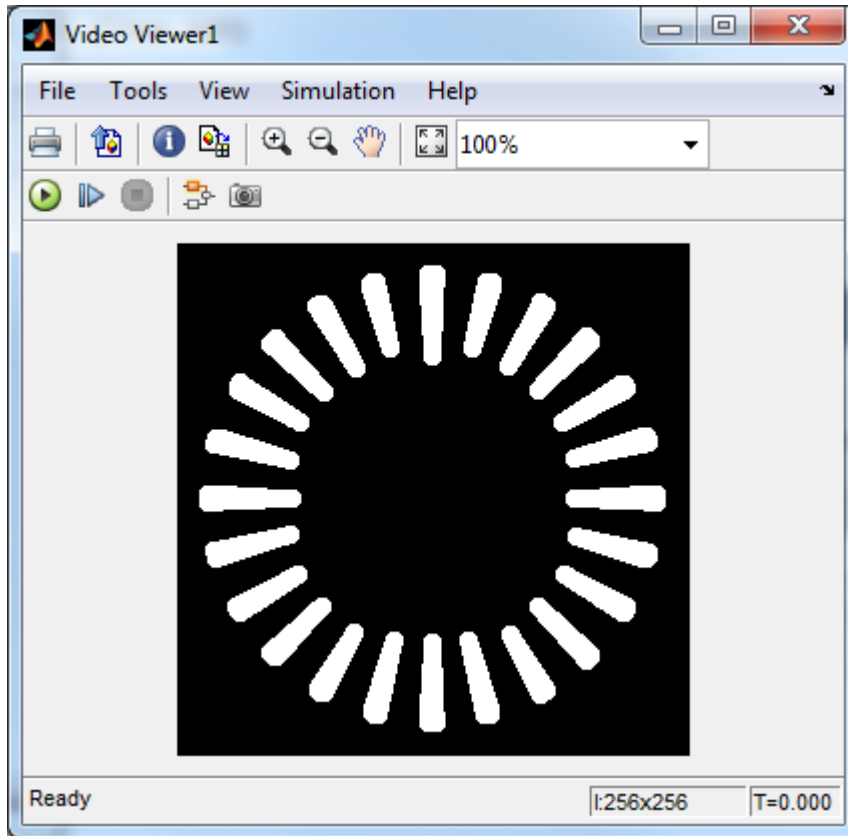
As you can see in the preceding figure, the spokes are now separate white objects. In the model, the Display block correctly indicates that there are 24 distinct spokes.
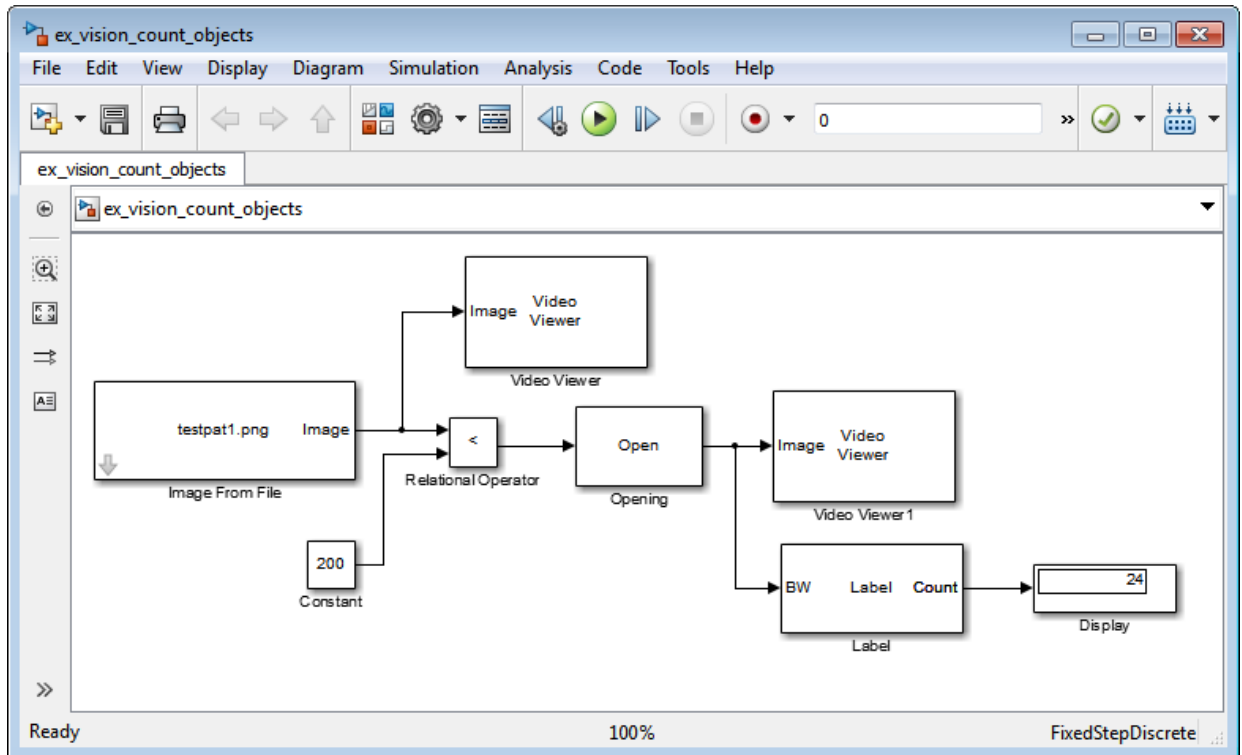
You have used the Opening and Label blocks to count the number of spokes in an image. For more information about these blocks, see the Opening and Label block reference pages in the *Computer Vision System Toolbox Reference*. If you want to send the number of spokes to the MATLAB workspace, use the To Workspace block in Simulink. For more information about STREL objects, see `strel` in the Image Processing Toolbox documentation.

12

Fixed-Point Design

Fixed-Point Signal Processing

In this section...
"Fixed-Point Features" on page 12-2
"Benefits of Fixed-Point Hardware" on page 12-2
"Benefits of Fixed-Point Design with System Toolboxes Software" on page 12-3

Note To take full advantage of fixed-point support in System Toolbox software, you must install Fixed-Point Designer™ 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 12-4
"Scaling" on page 12-5
"Precision and Range" on page 12-6

Note The Glossary (DSP System Toolbox) defines much of the vocabulary used in these sections. For more information on these subjects, see "Fixed-Point Designer".

Fixed-Point Data Types

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

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

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

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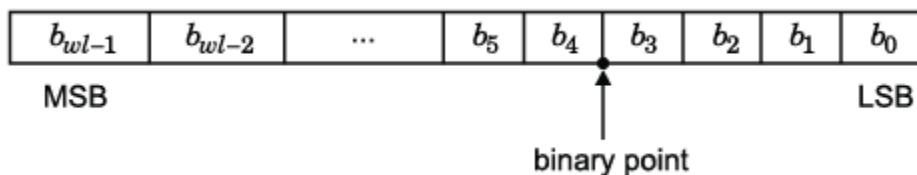
where

- b_i is the ith binary digit.

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

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

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

Two's complement is the most common representation of signed fixed-point numbers and is used by System Toolbox software. See "Two's Complement" on page 12-10 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

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

where the slope can be expressed as

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

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

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in the Fixed-Point Designer [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:

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

or

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

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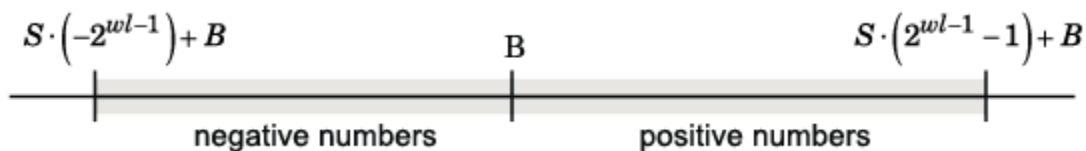
For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

For example, in two's complement, negative numbers must be represented as well as zero, so the maximum value is 2^{wl-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 -2^{wl-1} but not for 2^{wl-1}:

For slope = 1 and bias = 0:
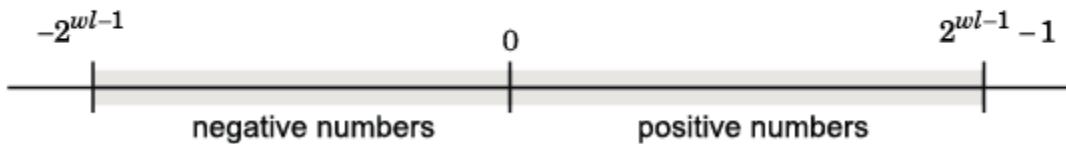
Overflow Handling

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 12-9 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" (Fixed-Point Designer).

- `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" (Fixed-Point Designer).

For a direct comparison of the rounding modes, see "Choosing a Rounding Method" (Fixed-Point Designer).

Arithmetic Operations

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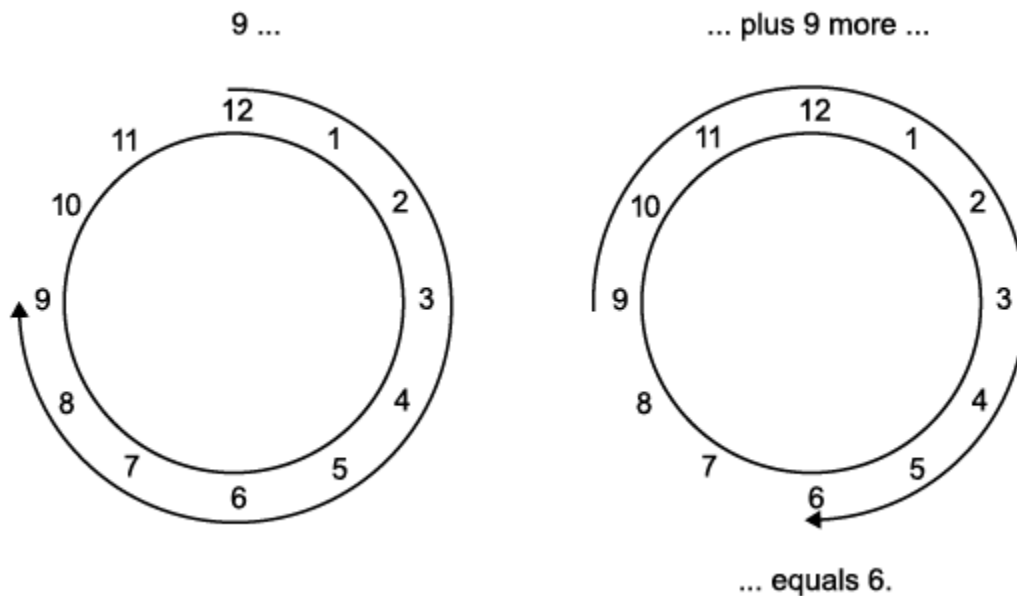
Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped "around the circle" to either 0 or 1.

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:

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

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):

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

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

Most fixed-point DSP System Toolbox blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting

is necessary during the addition to line up the binary points. See "Casts" on page 12-14 for more information.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):

Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages to determine whether a particular block accepts complex fixed-point inputs.

In most cases, you can set the data types used during multiplication in the block mask. For details, see "Casts" on page 12-14.

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.
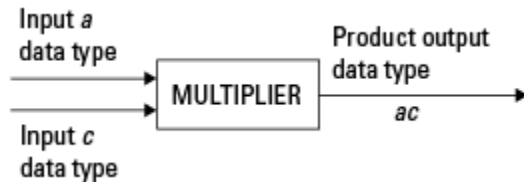
Real-Complex Multiplication

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.
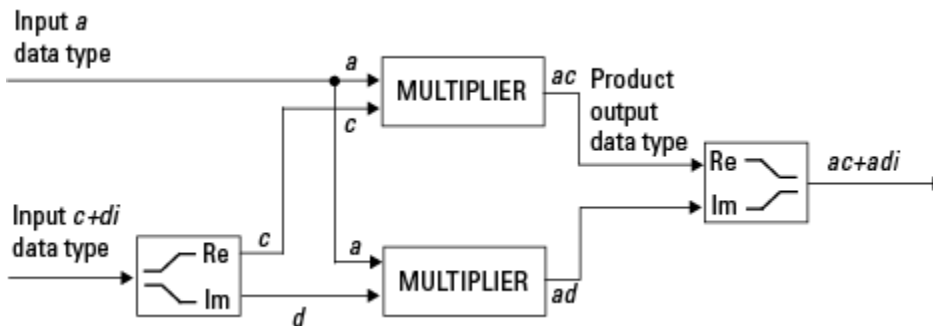
Complex-Complex Multiplication

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.
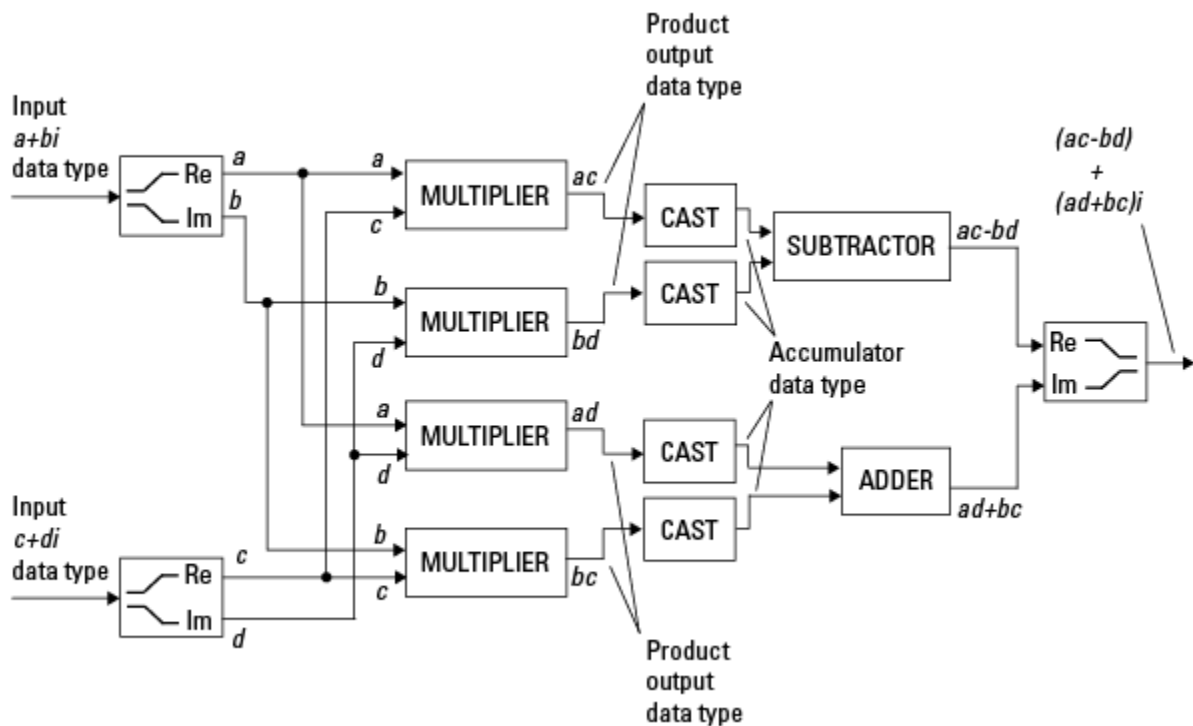
System Toolbox blocks cast to the accumulator data type before performing addition or subtraction operations. In the preceding diagram, this is equivalent to the C code

```
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. For details, see the description for **Accumulator** data type parameter in "Specify Fixed-Point Attributes for Blocks" (DSP System Toolbox). 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. For details, see the description for **Intermediate Product** and **Product Output** data type parameters in "Specify Fixed-Point Attributes for Blocks" (DSP System Toolbox).

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 cannot 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:
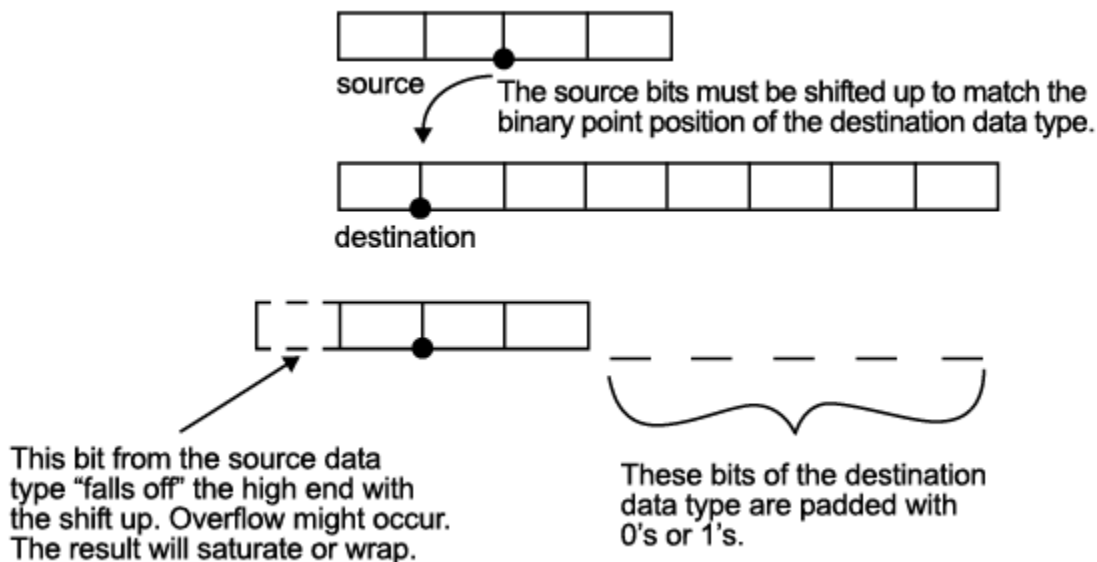
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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:
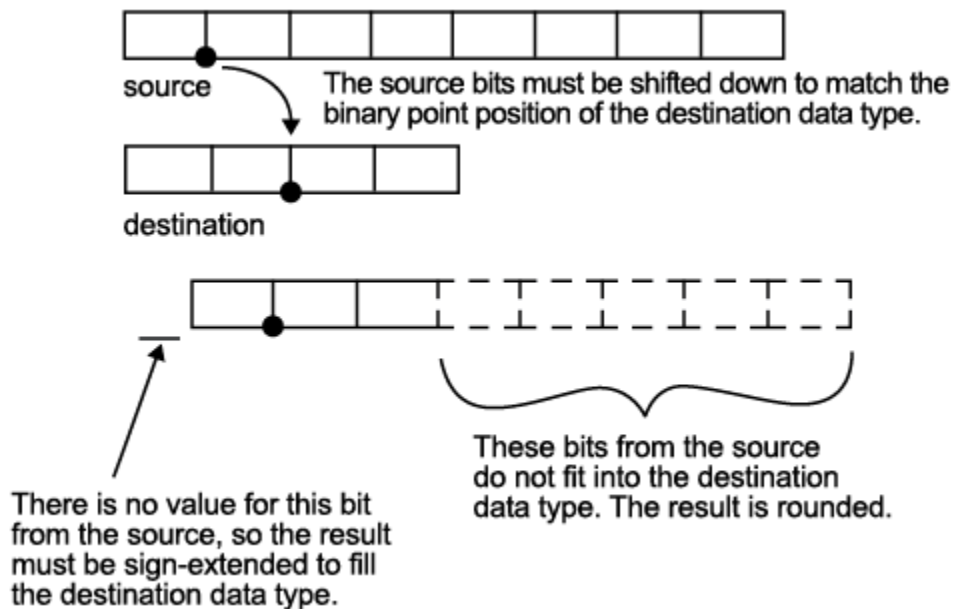
As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so the result is sign extended to fill the integer portion of the destination data type. The bottom five bits of the source 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

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. When you display the properties of a System object, click `Show all properties` at the end of the property list to display the fixed-point properties for that object. You can also display the fixed-point properties for a particular object by typing `vision.<ObjectName>.helpFixedPoint` at the command line.

The following Computer Vision System Toolbox objects support fixed-point data processing.

Fixed-Point Data Processing Support
vision.AlphaBlender
vision.Autocorrelator
vision.BlobAnalysis
vision.BlockMatcher
vision.Convolver
vision.Crosscorrelator
vision.DCT
vision.Deinterlacer
vision.DemosaicInterpolator
vision.FFT
vision.HoughLines
vision.IDCT
vision.IFFT
vision.Maximum
vision.Mean
vision.Median
vision.Minimum
vision.Pyramid

`vision.Variance`

Setting System Object Fixed-Point Properties

Several 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. You also use the Fixed-Point Designer `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, you will get a warning message.

You must set the property that activates a dependent property before attempting to change the dependent property. If you do not set the activating property, you will get a warning message.

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

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.

Main	Data Types

Fixed-point operational parameters

Rounding mode: Floor ▾   ☐ Saturate on integer overflow

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. When the block input is fixed point, all internal data types are signed fixed point.

	Data Type		Minimum	Maximum
Sine table:	Inherit: Same word length as i ▾	>>	N/A	N/A
Product output:	Inherit: Inherit via internal rule ▾	>>	N/A	N/A
Accumulator:	Inherit: Inherit via internal rule ▾	>>	N/A	N/A
Output:	Inherit: Inherit via internal rule ▾	>>	[]	[]

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

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 12-7 for more information on the available options.
Saturate on integer overflow	When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see "Overflow Handling" on page 12-7 for fixed-point operations.

Fixed-Point Data Type Parameter	Description
Intermediate Product	Specifies the data type and scaling of the intermediate product for fixed-point blocks. Blocks that feed multiplication results back to the input of the multiplier use the intermediate product data type. See the reference page of a specific block to learn about the intermediate product data type for that block.
Product Output	Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results. See the reference page of a specific block to learn about the product output data type for that block. For or complex-complex multiplication, the multiplication result is in the accumulator data type. See "Multiplication Data Types" on page 12-12 for more information on complex fixed-point multiplication in System toolbox software.
Accumulator	Specifies the data type and scaling of the accumulator (sum) for fixed-point blocks that must hold summation results for further calculation. Most such blocks cast to the accumulator data type before performing the add operations (summation). See the reference page of a specific block for details on the accumulator data type of that block.
Output	Specifies the output data type and scaling for blocks.

Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool available on the **Data Types** pane of some fixed-point System Toolbox blocks.

To learn more about using the **Data Type Assistant** to help you specify block data type parameters, see "Specify Data Types Using Data Type Assistant" (Simulink).

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" (Simulink).

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 more information, see `fxptdlg` and "Fixed-Point Tool" (Fixed-Point Designer).

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 12-25
- "Internal Rule for Product Data Types" on page 12-25

- "Internal Rule for Output Data Types" on page 12-26
- "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 12-26
- "Internal Rule Examples" on page 12-28

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 12-26 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 12-26 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 12-26.

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.
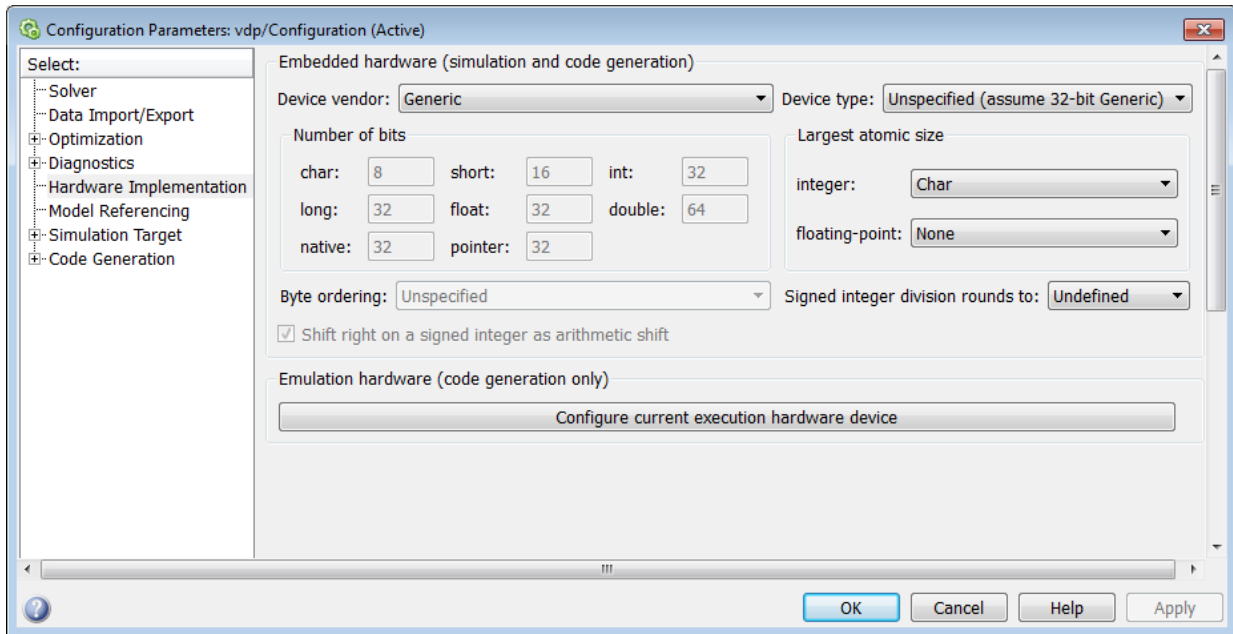
ASIC/FPGA

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.

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.

The largest word length allowed for Simulink and System Toolbox software on any target is 128 bits.

Internal Rule Examples

The following sections show examples of how the internal rule interacts with the **Hardware Implementation** pane to calculate accumulator data types on page 12-28 and product data types on page 12-31.

Accumulator Data Types

Consider the following model ex_internalRule_accumExp.
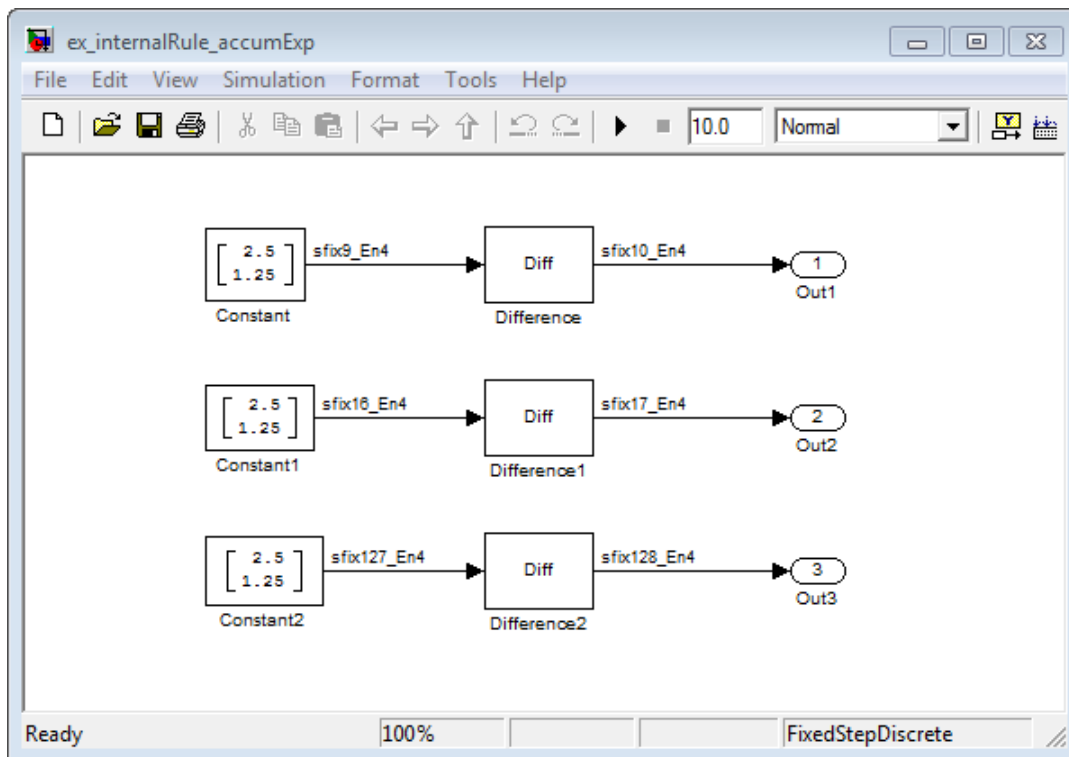
In the Difference blocks, the **Accumulator** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit: Same as accumulator`. Therefore, you can see the accumulator data type calculated by the internal rule on the output signal in the model.

In the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to ASIC/FPGA. Therefore, the accumulator data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Difference blocks in the model:

$$WL_{ideal\,accumulator} = WL_{input\,to\,accumulator} + \text{floor}(\log_2(number\,of\,accumulations)) + 1$$
$$WL_{ideal\,accumulator} = 9 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\,accumulator} = 9 + 0 + 1 = 10$$

$$WL_{ideal\,accumulator1} = WL_{input\,to\,accumulator1} + \text{floor}(\log_2(number\,of\,accumulations)) + 1$$
$$WL_{ideal\,accumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\,accumulator1} = 16 + 0 + 1 = 17$$

$$WL_{ideal\,accumulator2} = WL_{input\,to\,accumulator2} + \text{floor}(\log_2(number\,of\,accumulations)) + 1$$
$$WL_{ideal\,accumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\,accumulator2} = 127 + 0 + 1 = 128$$

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

$$FL_{ideal\,accumulator} = FL_{input\,to\,accumulator}$$
$$FL_{ideal\,accumulator} = 4$$

Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to 32—bit Embedded Processor, by changing the parameters as shown in the following figure.
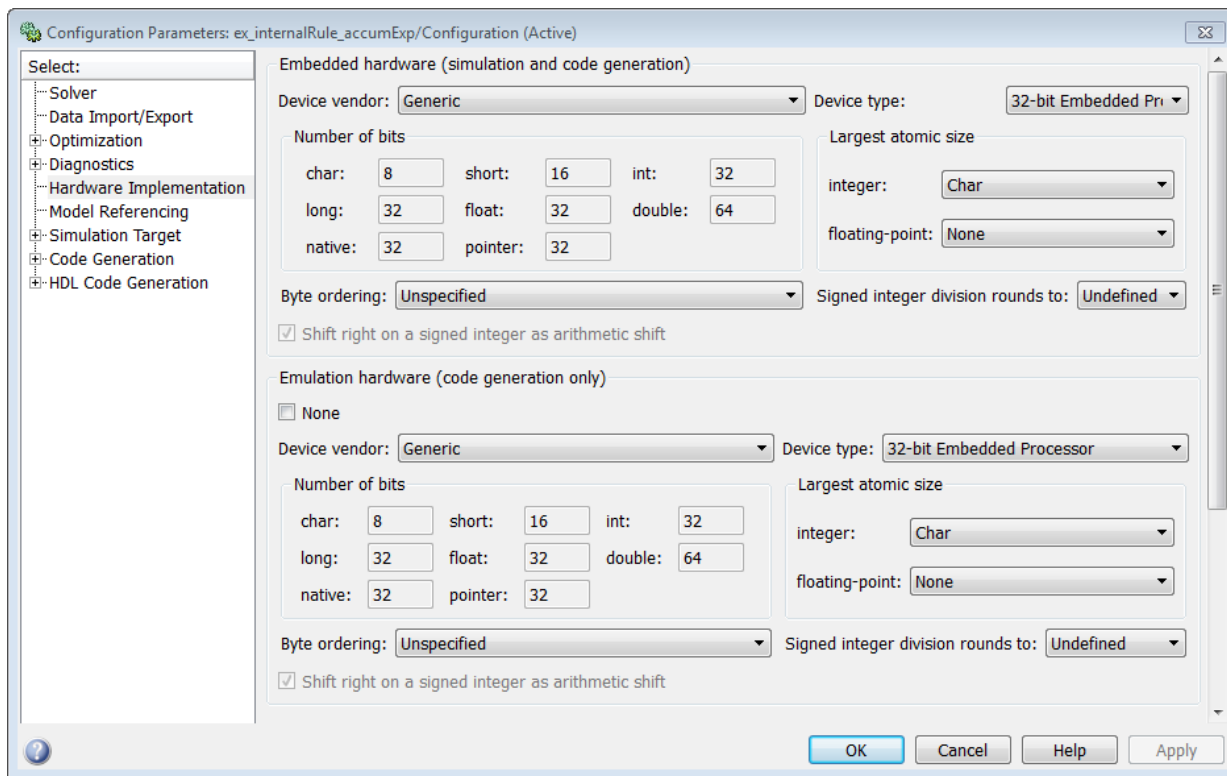
As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 10, 17, and 128 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case You can see this if you rerun the model, as shown in the following figure.

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:
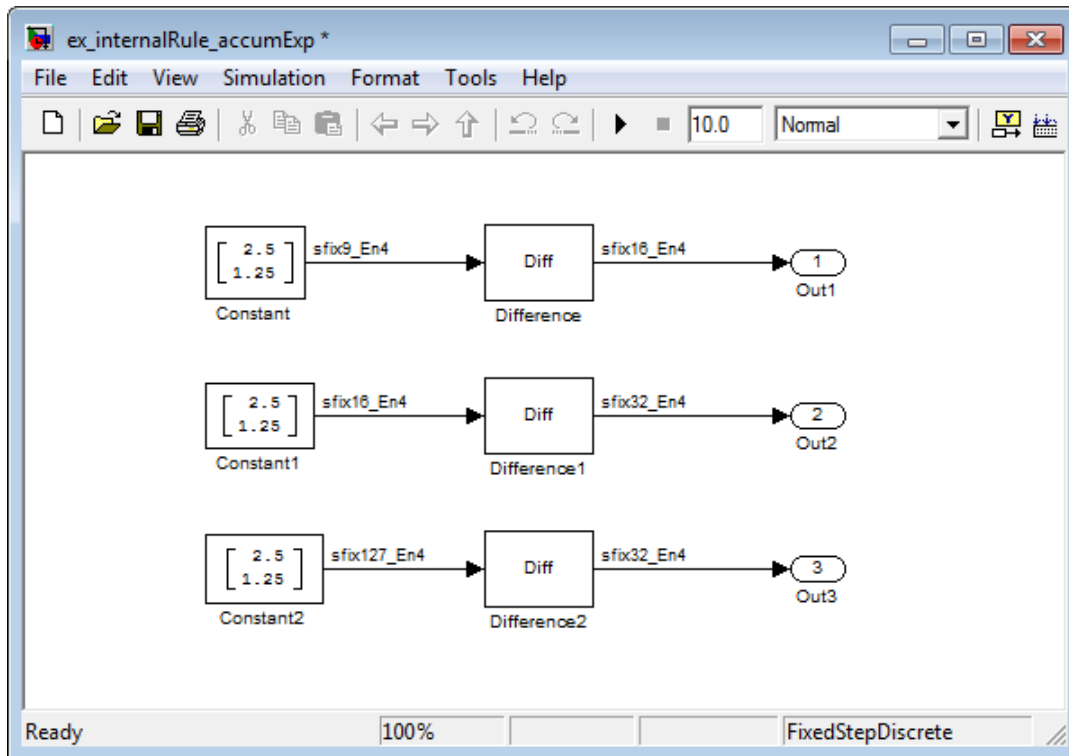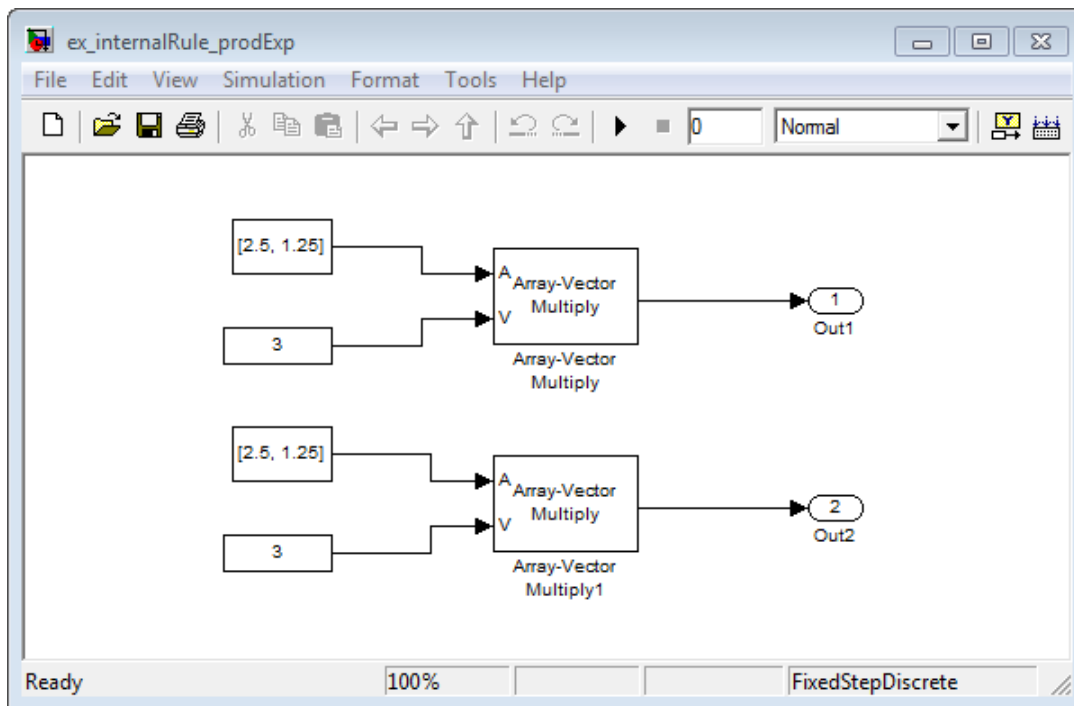
$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$
$$FL_{ideal\ accumulator} = 4$$

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.
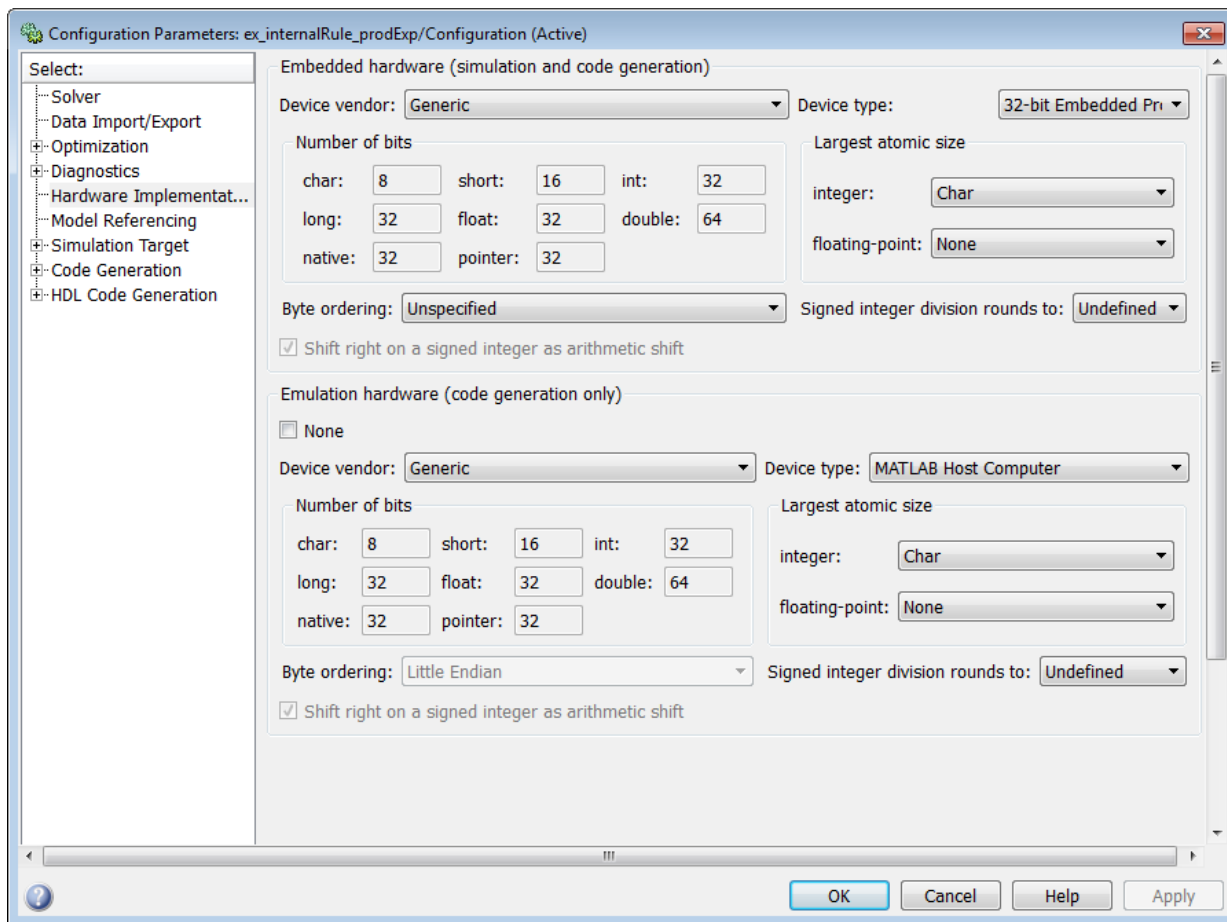
As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 12 and 31 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case. You can see this if you rerun the model, as shown in the following figure.
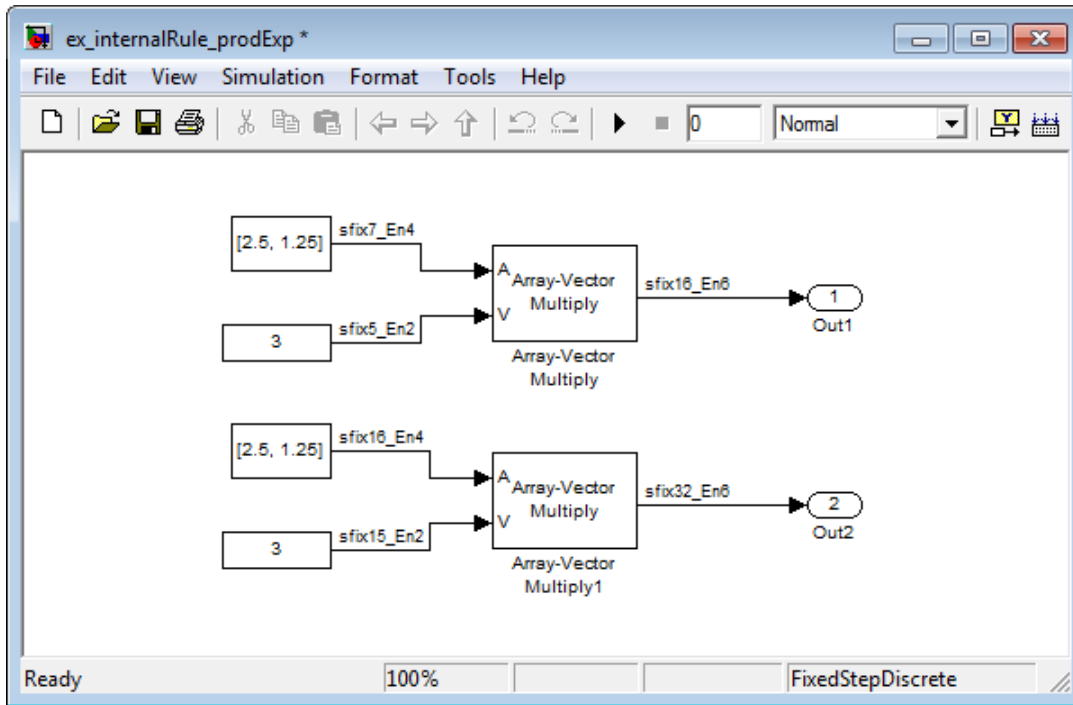
Specify Data Types for Fixed-Point Blocks

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 12-35
- "Use Data Type Override to Find a Floating-Point Benchmark" on page 12-40
- "Use the Fixed-Point Tool to Propose Fraction Lengths" on page 12-40
- "Examine the Results and Accept the Proposed Scaling" on page 12-41

Prepare the Model

1   Open the model by typing ex_fixedpoint_tut at the MATLAB command line.
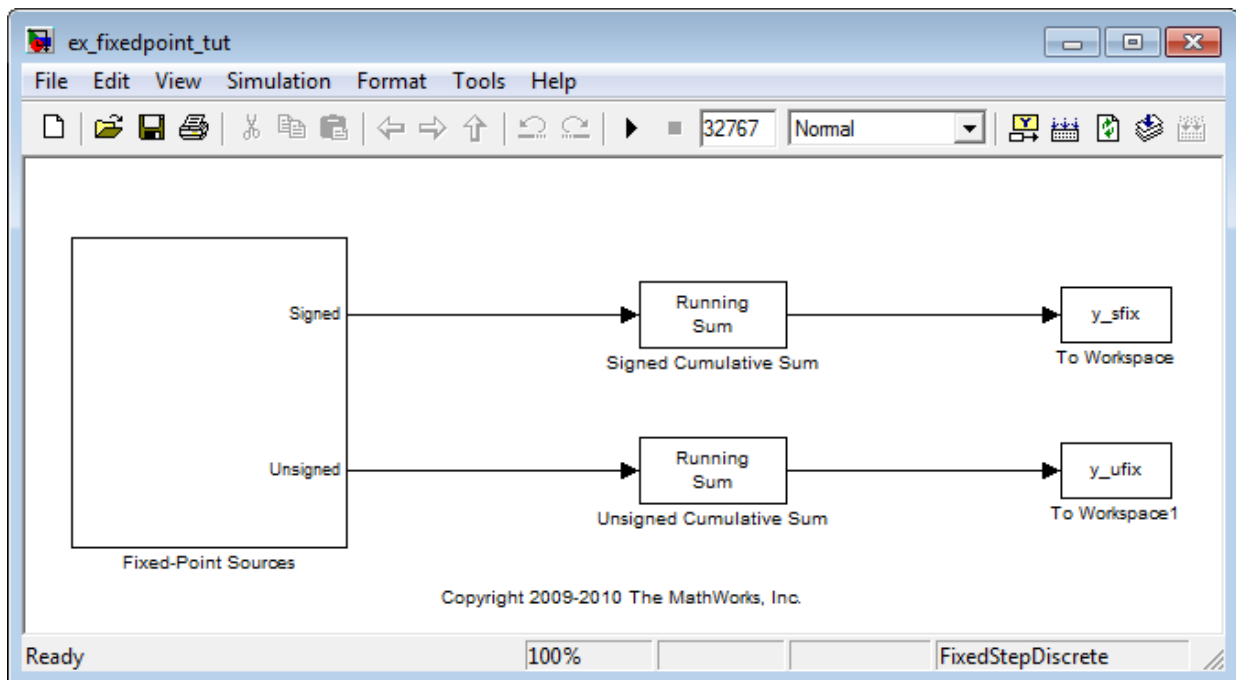
This model uses the Cumulative Sum block to sum the input coming from the Fixed-Point Sources subsystem. The Fixed-Point Sources subsystem outputs two signals with different data types:

- 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 Simulation button.
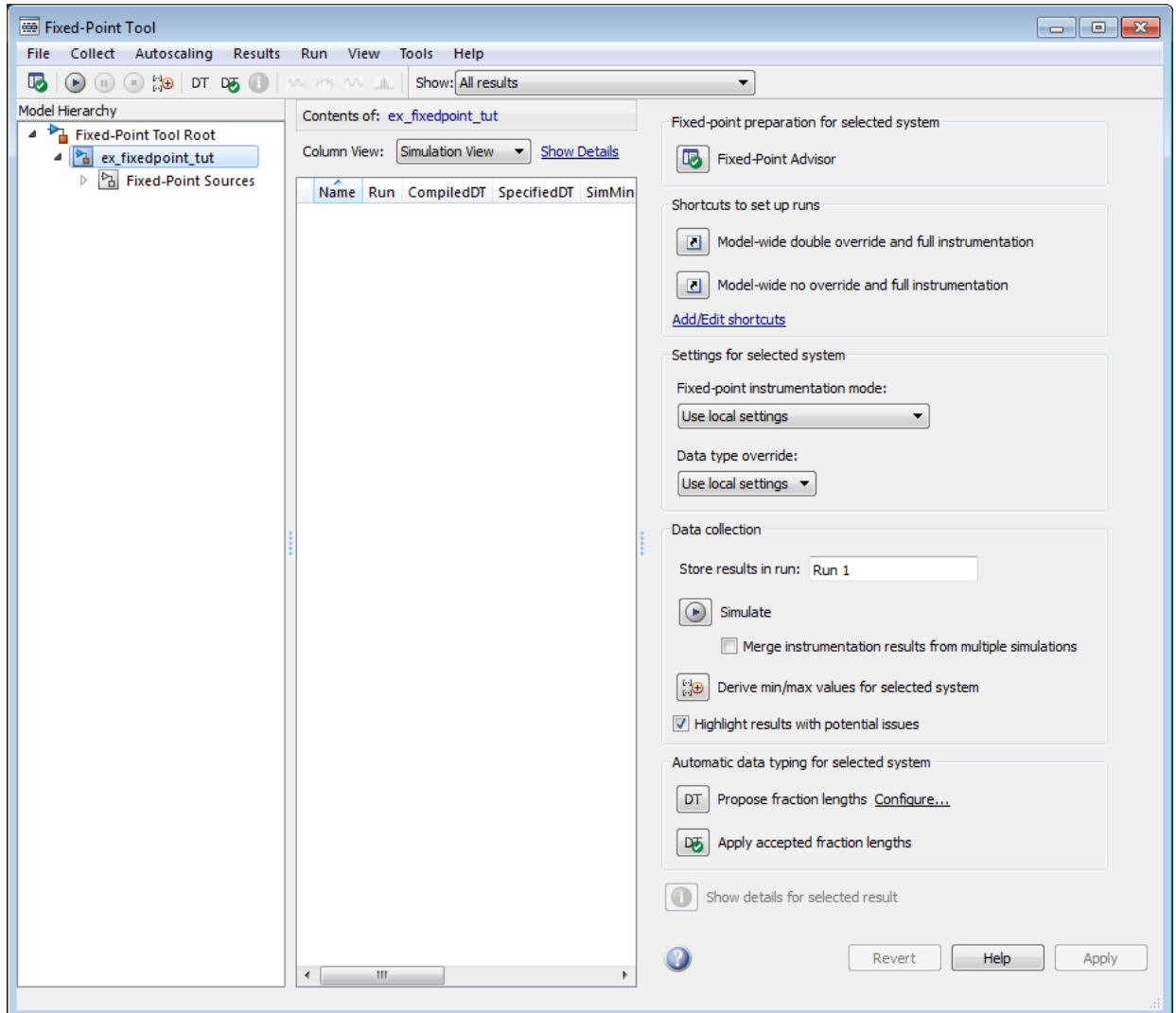
5 The results of the simulation appear in a table in the central **Contents** pane of the Fixed-Point Tool. Review the following columns:

- **Name** — Provides the name of each signal in the following format: `Subsystem Name/Block Name: Signal Name`.

- **SimDT** — The simulation data type of each logged signal.

- **SpecifiedDT** — The data type specified on the block dialog for each signal.

- **SimMin** — The smallest representable value achieved during simulation for each logged signal.

- **SimMax** — The largest representable value achieved during simulation for each logged signal.

- **OverflowWraps** — The number of overflows that wrap during simulation.

For more information on each of the columns in this table, see the "Contents Pane" (Simulink) 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 (![info icon])

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:

   1 Right-click the `Signed Cumulative Sum: Accumulator` row in the Fixed-Point Tool pane, and select **Highlight Block In Model**.

   2 Double-click the block in the model, and select the **Data Types** pane of the dialog box.

   3 Open the **Data Type Assistant for Accumulator** by clicking the Assistant button ([>>]) in the Accumulator data type row.

   4 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.
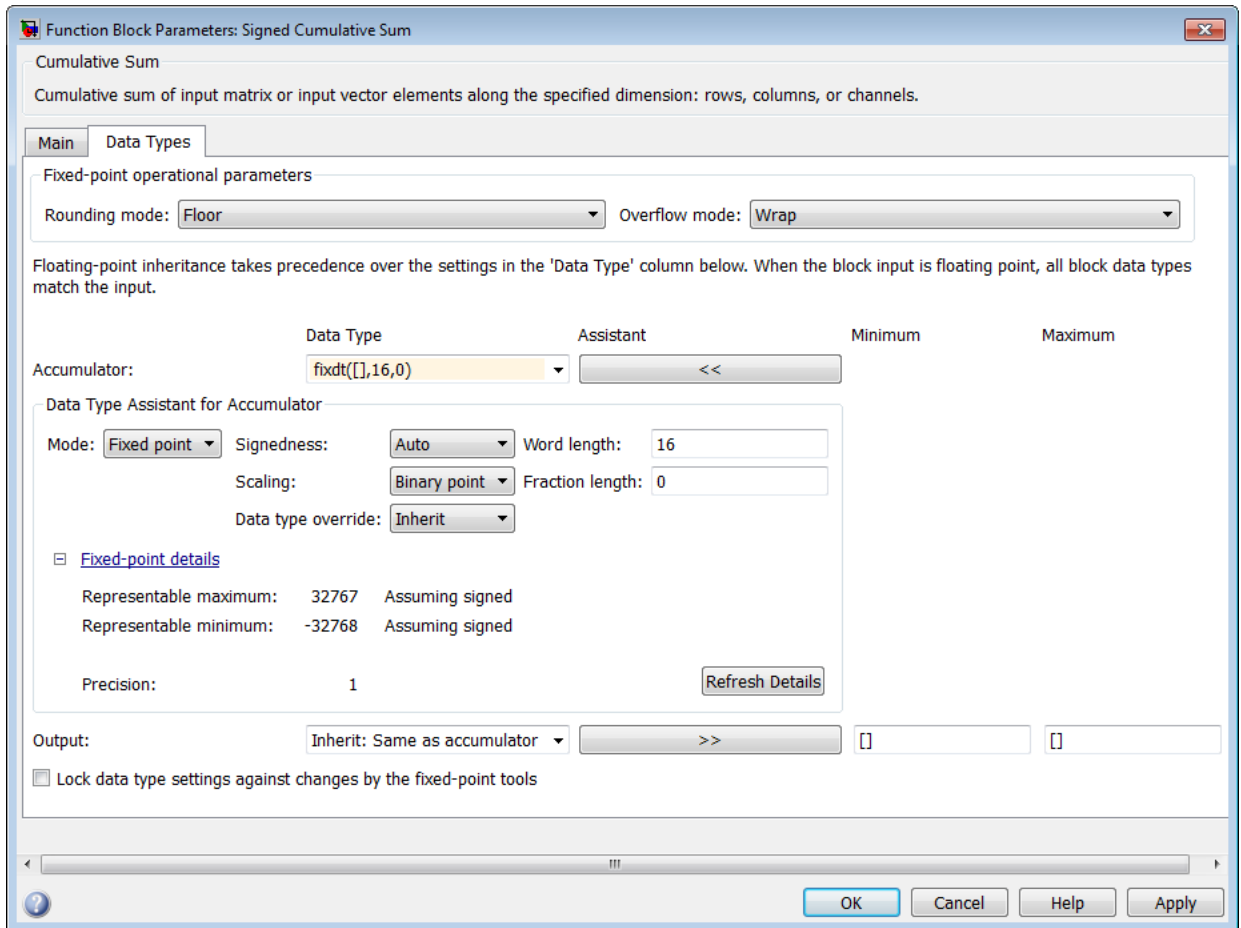
5   Change the **Word length** to 32, and click the **Refresh details** button in the **Fixed-point details** section to see the updated representable range. When you change the value of the **Word length** parameter, the **Data Type** edit box automatically updates.

6   Click **OK** on the block dialog box to save your changes and close the window.

7   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 `fixdt([],32,0)` directly into **Data Type** edit box for the Accumulator data type parameter.

- Perform the same steps you used to set the word length of the Accumulator data type of the Signed Cumulative Sum block to 32 bits.

7 To verify your changes in word length and check for overflow, rerun your model. To do so, click the **Simulate** button in the Fixed-Point Tool.

The **Contents** pane of the Fixed-Point Tool updates, and you can see that no overflows occurred in the most recent simulation. However, you can also see that the **SimMin** and **SimMax** values range from 0 to 0. This underflow happens because the fraction length of the Accumulator data type is too small. The **SpecifiedDT** cannot represent the precision of the data values. The following sections discuss how to find a floating-point benchmark and use the Fixed-Point Tool to propose fraction lengths.

Use Data Type Override to Find a Floating-Point Benchmark

The **Data type override** feature of the Fixed-Point tool allows you to override the data types specified in your model with floating-point types. Running your model in `Double` override mode gives you a reference range to help you select appropriate fraction lengths for your fixed-point data types. To do so, perform the following steps:

1 Open the Fixed-Point Tool and set **Data type override** to `Double`.
2 Run your model by clicking the **Run simulation and store active results** button.
3 Examine the results in the **Contents** pane of the Fixed-Point Tool. Because you ran the model in `Double` override mode, you get an accurate, idealized representation of the simulation minimums and maximums. These values appear in the **SimMin** and **SimMax** parameters.
4 Now that you have an accurate reference representation of the simulation minimum and maximum values, you can more easily choose appropriate fraction lengths. Before making these choices, save your active results to reference so you can use them as your floating-point benchmark. To do so, select **Results > Move Active Results To Reference** from the Fixed-Point Tool menu. The status displayed in the **Run** column changes from `Active` to `Reference` for all signals in your model.

Use the Fixed-Point Tool to Propose Fraction Lengths

Now that you have your `Double` override results saved as a floating-point reference, you are ready to propose fraction lengths.

1 To propose fraction lengths for your data types, you must have a set of `Active` results available in the Fixed-Point Tool. To produce an active set of results, simply rerun your model. The tool now displays both the `Active` results and the `Reference` results for each signal.

2 Select the **Use simulation min/max if design min/max is not available** check box. You did not specify any design minimums or maximums for the data types in this model. Thus, the tool uses the logged information to compute and propose fraction lengths. For information on specifying design minimums and maximums, see "Signal Ranges" (Simulink).

3

Click the **Propose fraction lengths** button ( ). The tool populates the proposed data types in the **ProposedDT** column of the **Contents** pane. The corresponding proposed minimums and maximums are displayed in the **ProposedMin** and **ProposedMax** columns.

Examine the Results and Accept the Proposed Scaling

Before accepting the fraction lengths proposed by the Fixed-Point Tool, it is important to look at the details of that data type. Doing so allows you to see how much of your data the suggested data type can represent. To examine the suggested data types and accept the proposed scaling, perform the following steps:

1 In the **Contents** pane of the Fixed-Point Tool, you can see the proposed fraction lengths for the data types in your model.

- The proposed fraction length for the Accumulator data type of both the Signed and Unsigned Cumulative Sum blocks is 17 bits.

- To get more details about the proposed scaling for a particular data type, highlight the data type in the **Contents** pane of the Fixed-Point Tool.

- Open the Autoscale Information window for the highlighted data type by clicking

  the **Show autoscale information for the selected result** button ( ).

2 When the Autoscale Information window opens, check the **Value** and **Percent Proposed Representable** columns for the **Simulation Minimum** and **Simulation Maximum** parameters. You can see that the proposed data type can represent 100% of the range of simulation data.

3 To accept the proposed data types, select the check box in the **Accept** column for each data type whose proposed scaling you want to keep. Then, click the **Apply**

accepted fraction lengths button ( ). The tool updates the specified data types on the block dialog boxes and the **SpecifiedDT** column in the **Contents** pane.

4 To verify the newly accepted scaling, set the **Data type override** parameter back to **Use local settings**, and run the model. Looking at **Contents** pane of the Fixed-Point Tool, you can see the following details:

- The **SimMin** and **SimMax** values of the `Active` run match the **SimMin** and **SimMax** values from the floating-point `Reference` run.

- There are no longer any overflows.

- The **SimDT** does not match the **SpecifiedDT** for the Accumulator data type of either Cumulative Sum block. This difference occurs because the Cumulative Sum block always inherits its **Signedness** from the input signal and only allows you to specify a **Signedness** of `Auto`. Therefore, the **SpecifiedDT** for both Accumulator data types is `fixdt([],32,17)`. However, because the Signed Cumulative Sum block has a signed input signal, the **SimDT** for the Accumulator parameter of that block is also signed (`fixdt(1,32,17)`). Similarly, the **SimDT** for the Accumulator parameter of the Unsigned Cumulative Sum block inherits its **Signedness** from its input signal and thus is unsigned (`fixdt(0,32,17)`).

13

Code Generation

Code Generation in MATLAB

Several Computer Vision System Toolbox functions have been enabled to generate C/C++ code. To use code generation with computer vision functions, follow these steps:

- Write your Computer Vision System Toolbox function or application as you would normally, using functions from the Computer Vision System Toolbox.

- Add the `%#codegen` compiler directive to your MATLAB code.

- Open the MATLAB Coder app, create a project, and add your file to the project. Once in MATLAB Coder, you can check the readiness of your code for code generation. For example, your code may contain functions that are not enabled for code generation. Make any modifications required for code generation.

- Generate code by clicking **Generate** in the Generate Code dialog box. You can choose to build a MEX file, a C/C++ shared library, a C/C++ dynamic library, or a C/C++ executable.

  Even if you addressed all readiness issues identified by MATLAB Coder, you might still encounter build issues. The readiness check only looks at function dependencies. When you try to generate code, MATLAB Coder might discover coding patterns that are not supported for code generation. View the error report and modify your MATLAB code until you get a successful build.

For more information about code generation, see the MATLAB Coder documentation and the "Introduction to Code Generation with Feature Matching and Registration" example.

Note To generate code from MATLAB code that contains Computer Vision System Toolbox functionality, you must have the MATLAB Coder software.

When working with generated code, note the following:

- For some Computer Vision System Toolbox functions, code generation includes creation of a shared library.

- Refer to the "Code Generation Support, Usage Notes, and Limitations" on page 13-3 for supported functionality, usages, and limitations.

Code Generation Support, Usage Notes, and Limitations

Code Generation Support, Usage Notes, and Limitations for Functions, Classes, and System Objects

To generate code from MATLAB code that contains Computer Vision System Toolbox functions, classes, or System objects, you must have the MATLAB Coder software.

An asterisk (*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

Name
Feature Detection, Extraction, and Matching
BRISKPoints*
cornerPoints*
detectBRISKFeatures*
detectFASTFeatures*
detectHarrisFeatures*
detectMinEigenFeatures*
detectMSERFeatures*
detectSURFFeatures*
extractFeatures
extractHOGFeatures*
extractLBPFeatures*
matchFeatures*
MSERRegions*
SURFPoints*
Image Registration and Geometric Transformations
estimateGeometricTransform*
Object Detection and Recognition
ocr*
ocrText*

Name
vision.PeopleDetector*
vision.CascadeObjectDetector*
Tracking and Motion Estimation
assignDetectionsToTracks
opticalFlowFarneback*
opticalFlowHS*
opticalFlowLKDoG*
opticalFlowLK*
vision.ForegroundDetector*
vision.HistogramBasedTracker*
vision.KalmanFilter*
vision.PointTracker*
vision.TemplateMatcher*
Camera Calibration and Stereo Vision
bboxOverlapRatio*
bbox2points
disparity*
cameraPoseToExtrinsics
cameraMatrix*
cameraPose*
cameraParameters*
detectCheckerboardPoints*
epipolarLine
estimateEssentialMatrix*
estimateFundamentalMatrix*
estimateUncalibratedRectification
estimateWorldCameraPose*
extrinsics*

Name
extrinsicsToCameraPose
generateCheckerboardPoints*
isEpipoleInImage
lineToBorderPoints
reconstructScene*
rectifyStereoImages*
relativeCameraPose*
rotationMatrixToVector
rotationVectorToMatrix
selectStrongestBbox*
stereoAnaglyph
stereoParameters*
triangulate*
undistortImage*
Statistics
vision.Autocorrelator*
vision.BlobAnalysis*
vision.Crosscorrelator*
vision.LocalMaximaFinder*
vision.Maximum*
vision.Mean*
vision.Median*
vision.Minimum*
vision.StandardDeviation*
vision.Variance*
Filters, Transforms, and Enhancements
integralImage
vision.Convolver*

Name
vision.DCT*
vision.Deinterlacer*
vision.FFT*
vision.IDCT*
vision.IFFT*
vision.Pyramid*
Video Loading, Saving, and Streaming
vision.DeployableVideoPlayer*
vision.VideoFileReader*
vision.VideoFileWriter*
Color Space Formatting and Conversions
vision.ChromaResampler*
vision.DemosaicInterpolator*
vision.GammaCorrector*
Graphics
insertMarker*
insertShape*
insertObjectAnnotation*
insertText*
vision.AlphaBlender*

Simulink 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" (Simulink Coder).

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` Simulink Desktop Real-Time™ `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.
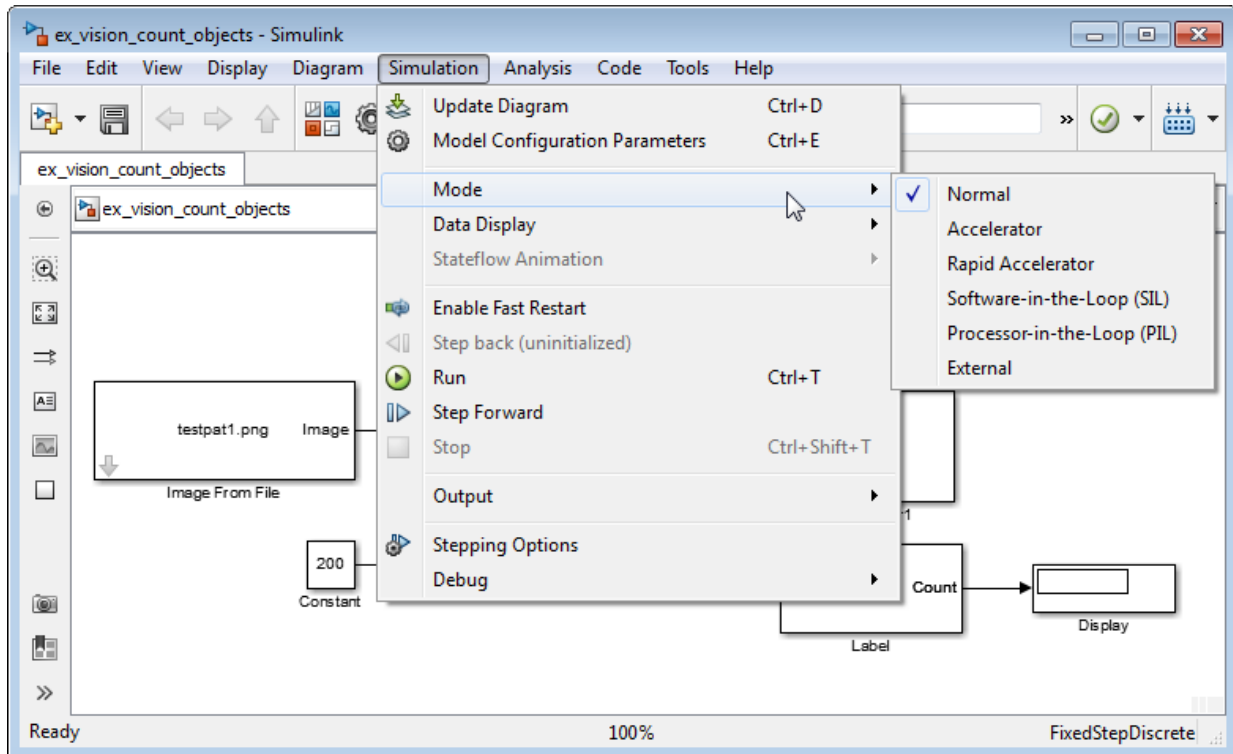
For more information on the accelerator modes in Simulink, see "Choosing a Simulation Mode" (Simulink).

Portable C Code Generation for Functions That Use OpenCV Library
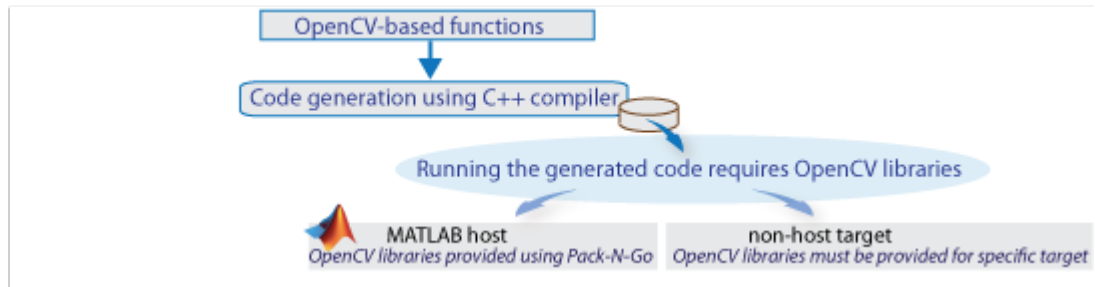
The generated binary uses prebuilt OpenCV libraries that ship with the Computer Vision System Toolbox product. Your compiler must be compatible with the one used to build the libraries. The following compilers are used to build the OpenCV libraries for MATLAB host:

Operating System	Compatible Compiler
Windows 64 bit	Microsoft Visual Studio 2015 Professional
Linux 64 bit	gcc-4.9.3 (g++)
Mac 64 bit	Xcode 6.2.0 (Clang++)

Limitations

Computer Vision System Toolbox functions that use the OpenCV library do not support target code generation from Simulink.