

Computer Vision Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2020b



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.
1 Apple Hill Drive
Natick, MA 01760-2098

Computer Vision Toolbox™ User's Guide

© COPYRIGHT 2004–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

July 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Second printing	Revised for Version 1.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.1 (Release 14SP2)
September 2005	Online only	Revised for Version 1.2 (Release 14SP3)
November 2005	Online only	Revised for Version 2.0 (Release 14SP3+)
March 2006	Online only	Revised for Version 2.1 (Release 2006a)
September 2006	Online only	Revised for Version 2.2 (Release 2006b)
March 2007	Online only	Revised for Version 2.3 (Release 2007a)
September 2007	Online only	Revised for Version 2.4 (Release 2007b)
March 2008	Online only	Revised for Version 2.5 (Release 2008a)
October 2008	Online only	Revised for Version 2.6 (Release 2008b)
March 2009	Online only	Revised for Version 2.7 (Release 2009a)
September 2009	Online only	Revised for Version 2.8 (Release 2009b)
March 2010	Online only	Revised for Version 3.0 (Release 2010a)
September 2010	Online only	Revised for Version 3.1 (Release 2010b)
April 2011	Online only	Revised for Version 4.0 (Release 2011a)
September 2011	Online only	Revised for Version 4.1 (Release 2011b)
March 2012	Online only	Revised for Version 5.0 (Release 2012a)
September 2012	Online only	Revised for Version 5.1 (Release R2012b)
March 2013	Online only	Revised for Version 5.2 (Release R2013a)
September 2013	Online only	Revised for Version 5.3 (Release R2013b)
March 2014	Online only	Revised for Version 6.0 (Release R2014a)
October 2014	Online only	Revised for Version 6.1 (Release R2014b)
March 2015	Online only	Revised for Version 6.2 (Release R2015a)
September 2015	Online only	Revised for Version 7.0 (Release R2015b)
March 2016	Online only	Revised for Version 7.1 (Release R2016a)
September 2016	Online only	Revised for Version 7.2 (Release R2016b)
March 2017	Online only	Revised for Version 7.3 (Release R2017a)
September 2017	Online only	Revised for Version 8.0 (Release R2017b)
March 2018	Online only	Revised for Version 8.1 (Release R2018a)
September 2018	Online only	Revised for Version 8.2 (Release R2018b)
March 2019	Online only	Revised for Version 9.0 (Release R2019a)
September 2019	Online only	Revised for Version 9.1 (Release R2019b)
March 2020	Online only	Revised for Version 9.2 (Release R2020a)
September 2020	Online only	Revised for Version 9.3 (Release R2020b)

1	Camera Calibration Examples	
	Camera Calibration Using AprilTag Markers	1-2
	Configure Monocular Fisheye Camera	1-13
	Monocular Visual Simultaneous Localization and Mapping	1-18
	Structure From Motion From Two Views	1-37
	Evaluating the Accuracy of Single Camera Calibration	1-47
	Measuring Planar Objects with a Calibrated Camera	1-52
	Depth Estimation From Stereo Video	1-61
	Structure From Motion From Multiple Views	1-70
	Uncalibrated Stereo Image Rectification	1-78

2	Code Generation and Third-Party Examples	
	Code Generation for Object Detection by Using Single Shot Multibox Detector	2-2
	Code Generation for Object Detection by Using YOLO v2	2-5
	Introduction to Code Generation with Feature Matching and Registration	2-8
	Code Generation for Face Tracking with PackNGo	2-14
	Code Generation for Depth Estimation From Stereo Video	2-22
	Detect Face (Raspberry Pi2)	2-27
	Track Face (Raspberry Pi2)	2-33
	Video Display in a Custom User Interface	2-39

Deep Learning, Semantic Segmentation, and Detection Examples

3

Point Cloud Classification Using PointNet Deep Learning	3-2
Object Detection Using SSD Deep Learning	3-23
Object Detection in a Cluttered Scene Using Point Feature Matching ..	3-32
Semantic Segmentation Using Deep Learning	3-43
Calculate Segmentation Metrics in Block-Based Workflow	3-59
Semantic Segmentation of Multispectral Images Using Deep Learning	3-64
3-D Brain Tumor Segmentation Using Deep Learning	3-81
Image Category Classification Using Bag of Features	3-93
Image Category Classification Using Deep Learning	3-100
Image Retrieval Using Customized Bag of Features	3-109
Create SSD Object Detection Network	3-115
Train YOLO v2 Network for Vehicle Detection	3-118
Object Detection Using YOLO v2 Deep Learning	3-123
Import Pretrained ONNX YOLO v2 Object Detector	3-133
Export YOLO v2 Object Detector to ONNX	3-140
Estimate Anchor Boxes From Training Data	3-146
Object Detection Using YOLO v3 Deep Learning	3-150
Object Detection Using YOLO v2 Deep Learning	3-170
Create YOLO v2 Object Detection Network	3-180
Train Object Detector Using R-CNN Deep Learning	3-183
Object Detection Using Faster R-CNN Deep Learning	3-197
Train Classification Network to Classify Object in 3-D Point Cloud ...	3-207
Estimate Body Pose Using Deep Learning	3-217
Generate Image from Segmentation Map Using Deep Learning	3-224

Create Simple Semantic Segmentation Network in Deep Network Designer	3-242
Train ACF-Based Stop Sign Detector	3-247
Activity Recognition from Video and Optical Flow Data Using Deep Learning	3-249
Train Fast R-CNN Stop Sign Detector	3-273

Feature Detection and Extraction Examples

4

Automatically Detect and Recognize Text in Natural Images	4-2
Digit Classification Using HOG Features	4-14
Find Image Rotation and Scale Using Automated Feature Matching ...	4-22
Feature Based Panoramic Image Stitching	4-27
Cell Counting	4-33
Object Counting	4-36
Pattern Matching	4-38
Recognize Text Using Optical Character Recognition (OCR)	4-43
Cell Counting	4-56

Lidar and Point Cloud Processing Examples

5

Augment Point Cloud Data For Deep Learning	5-2
Import Point Cloud Data For Deep Learning	5-7
Encode Point Cloud Data For Deep Learning	5-11
Build a Map from Lidar Data	5-17
Build a Map from Lidar Data Using SLAM	5-38
3-D Point Cloud Registration and Stitching	5-54

Multicore Simulation of Video Processing System	6-2
Concentricity Inspection	6-7
Object Counting	6-9
Video Focus Assessment	6-11
Video Compression	6-13
Barcode Recognition	6-15
Motion Detection	6-17
Pattern Matching	6-19
Scene Change Detection	6-22
Surveillance Recording	6-24
Traffic Warning Sign Recognition	6-26
Abandoned Object Detection	6-29
Color-based Road Tracking	6-32
Detect and Track Face	6-36
Lane Departure Warning System	6-43
Tracking Cars Using Foreground Detection	6-47
Tracking Cars Using Optical Flow	6-50
Tracking Based on Color	6-52
Video Mosaicking	6-54
Video Stabilization	6-59
Periodic Noise Reduction	6-61
Rotation Correction	6-63
Barcode Recognition Using Live Video Acquisition	6-66
Edge Detection Using Live Video Acquisition	6-68
Noise Removal and Image Sharpening	6-73

Tracking and Motion Estimation Examples

7

Video Stabilization	7-2
Video Stabilization Using Point Feature Matching	7-5
Face Detection and Tracking Using CAMShift	7-15
Face Detection and Tracking Using the KLT Algorithm	7-20
Face Detection and Tracking Using Live Video Acquisition	7-26
Motion-Based Multiple Object Tracking	7-31
Tracking Pedestrians from a Moving Car	7-40
Use Kalman Filter for Object Tracking	7-50
Detect Cars Using Gaussian Mixture Models	7-61

Featured Examples

8

Localize and Read Multiple Barcodes in Image	8-2
Monocular Visual Odometry	8-22
Detect and Track Vehicles Using Lidar Data	8-35
Semantic Segmentation Using Dilated Convolutions	8-54
Define Custom Pixel Classification Layer with Tversky Loss	8-58
Track a Face in Scene	8-65
Create 3-D Stereo Display	8-70
Measure Distance from Stereo Camera to a Face	8-71
Reconstruct 3-D Scene from Disparity Map	8-72
Visualize Stereo Pair of Camera Extrinsic Parameters	8-75
Remove Distortion from an Image Using the Camera Parameters Object	8-78

9

Getting Started with Point Clouds Using Deep Learning	9-2
Import Point Cloud Data	9-2
Augment Data	9-2
Encode Point Cloud Data to Image-like Format	9-3
Train a Deep Learning Classification Network with Encoded Point Cloud Data	9-3
Point Cloud Registration and Mapping Overview	9-4
Registration and Mapping Workflow	9-4
Manage Point Cloud Registration and Mapping Data	9-6
Preprocess Point Clouds	9-6
Register Point Clouds	9-6
Detect Loops	9-8
Correct Drift	9-8
Assemble Map	9-8
Tips	9-8
The PLY Format	9-10
File Header	9-10
Data	9-11
Common Elements and Properties	9-12

**Using the Installer for Computer Vision System Toolbox
Product**

10

Install Computer Vision Toolbox Add-on Support Files	10-2
Install OCR Language Data Files	10-3
Installation	10-3
Pretrained Language Data and the ocr function	10-3
Install and Use Computer Vision Toolbox Interface for OpenCV in MATLAB	10-6
Installation	10-6
Support Package Contents	10-6
Create MEX-File from OpenCV C++ file	10-7
Use the Computer Vision Toolbox Interface for OpenCV in MATLAB C++ API	10-7
Create Your Own OpenCV MEX-files	10-8
Run OpenCV Examples	10-8
Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink	10-11
Installation	10-11
Import OpenCV Code into Simulink	10-11
Limitations	10-18

Smile Detection by Using OpenCV Code in Simulink	10-19
Required Products	10-19
Set Up Your C++ Compiler	10-19
Model Description	10-19
Copy Example Folder to a Writable Location	10-20
Step 1: Import OpenCV Function to Create a Simulink Library	10-21
Step 2: Use Generated Subsystem in Simulink Model	10-25
Step 3: Simulate the Smile Detector	10-26
Step 4: Generate C++ Code from the Smile Detector Model	10-27
Deploy the Smile Detector on the Raspberry Pi Hardware	10-28
Convert RGB Image to Grayscale Image by Using OpenCV Importer ..	10-29
Required Products	10-29
Set Up Your C++ Compiler	10-29
Model Description	10-29
Copy Example Folder to a Writable Location	10-30
Step 1: Import OpenCV Function to Create a Simulink Library	10-30
Step 2: Use Generated Subsystem in Simulink Model	10-33
Step 3: Simulate the RGB to Gray Convertor	10-34
Draw Different Shapes by Using OpenCV Code in Simulink	10-36
Required Products	10-36
Set Up Your C++ Compiler	10-36
Model Description	10-36
Copy Example Folder to a Writable Location	10-37
Step 1: Import OpenCV Function to Create a Simulink Library	10-37
Step 2: Use Generated Subsystem in Simulink Model	10-38
Draw Atom on Image by Using C Caller Block	10-39

Input, Output, and Conversions

11

Export to Video Files	11-2
Setting Block Parameters for this Example	11-2
Configuration Parameters	11-3
Import from Video Files	11-4
Setting Block Parameters for this Example	11-4
Configuration Parameters	11-5
Batch Process Image Files	11-6
Configuration Parameters	11-6
Convert R'G'B' to Intensity Images	11-7
Process Multidimensional Color Video Signals	11-10
Video Formats	11-12
Defining Intensity and Color	11-12
Video Data Stored in Column-Major Format	11-12

Image Formats	11-13
Binary Images	11-13
Intensity Images	11-13
RGB Images	11-13

Display and Graphics

12

Choose Function to Visualize Detected Objects	12-2
Display, Stream, and Preview Videos	12-5
View Streaming Video in MATLAB	12-5
Preview Video in MATLAB	12-5
View Video in Simulink	12-5
Draw Shapes and Lines	12-7
Rectangle	12-7
Line and Polyline	12-7
Polygon	12-9
Circle	12-9

Registration and Stereo Vision

13

Fisheye Calibration Basics	13-2
Fisheye Camera Model	13-3
Fisheye Camera Calibration in MATLAB	13-4
Single Camera Calibrator App	13-8
Camera Calibrator Overview	13-8
Single Camera Calibration	13-8
Open the Camera Calibrator	13-9
Prepare the Pattern, Camera, and Images	13-9
Add Images and Select Camera Model	13-12
Calibrate	13-15
Evaluate Calibration Results	13-17
Improve Calibration	13-20
Export Camera Parameters	13-23
Stereo Camera Calibrator App	13-25
Stereo Camera Calibrator Overview	13-25
Stereo Camera Calibration	13-25
Open the Stereo Camera Calibrator	13-26
Prepare Pattern, Camera, and Images	13-26
Add Image Pairs	13-29
Calibrate	13-31
Evaluate Calibration Results	13-31
Improve Calibration	13-35
Export Camera Parameters	13-37

What Is Camera Calibration?	13-39
Camera Models	13-39
Pinhole Camera Model	13-40
Camera Calibration Parameters	13-41
Distortion in Camera Calibration	13-42
Structure from Motion	13-45
Structure from Motion from Two Views	13-45
Structure from Motion from Multiple Views	13-46

Object Detection

14

Choose an Object Detector	14-2
Getting Started with SSD Multibox Detection	14-9
Predict Objects in the Image	14-9
Transfer Learning	14-10
Design an SSD Detection Network	14-10
Train an Object Detector and Detect Objects with an SSD Model	14-11
Code Generation	14-11
Label Training Data for Deep Learning	14-11
Getting Started with Object Detection Using Deep Learning	14-13
Create Training Data for Object Detection	14-13
Create Object Detection Network	14-14
Train Detector and Evaluate Results	14-14
Detect Objects Using Deep Learning Detectors	14-14
How Labeler Apps Store Exported Pixel Labels	14-16
Location of Pixel Label Data Folder	14-16
View Exported Pixel Label Data	14-16
Examples	14-17
Anchor Boxes for Object Detection	14-21
What Is an Anchor Box?	14-21
Advantage of Using Anchor Boxes	14-21
How Do Anchor Boxes Work?	14-22
Anchor Box Size	14-25
Getting Started with YOLO v2	14-26
Predicting Objects in the Image	14-26
Transfer Learning	14-27
Design a YOLO v2 Detection Network	14-27
Train an Object Detector and Detect Objects with a YOLO v2 Model	14-28
Code Generation	14-28
Label Training Data for Deep Learning	14-28
Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN	14-30
Object Detection Using R-CNN Algorithms	14-30
Comparison of R-CNN Object Detectors	14-32
Transfer Learning	14-32

Design an R-CNN, Fast R-CNN, and a Faster R-CNN Model	14-33
Label Training Data for Deep Learning	14-34
Getting Started with Mask R-CNN for Instance Segmentation	14-36
Training Data	14-36
Visualize Training Data	14-38
Preprocess Data	14-40
Mask R-CNN Network Architecture	14-40
Train Mask R-CNN Network	14-41
Getting Started with Semantic Segmentation Using Deep Learning	14-43
Train a Semantic Segmentation Network	14-43
Label Training Data for Semantic Segmentation	14-43
Training Data for Object Detection and Semantic Segmentation	14-45
Create Automation Algorithm for Labeling	14-49
Create New Algorithm	14-49
Import Existing Algorithm	14-50
Custom Algorithm Execution	14-50
Label Pixels for Semantic Segmentation	14-53
Start Pixel Labeling	14-53
Label Pixels Using Flood Fill Tool	14-54
Label Pixels Using Smart Polygon Tool	14-55
Label Pixels Using Polygon Tool	14-57
Label Pixels Using Assisted Freehand Tool	14-58
Replace Pixel Labels	14-59
Refine Labels Using Brush Tool	14-59
Visualize Pixel Labels	14-60
Tips	14-61
Get Started with the Image Labeler	14-63
Load Unlabeled Data	14-63
Create Label Definitions	14-63
Label Ground Truth	14-71
Export Labeled Ground Truth	14-72
Save App Session	14-74
Choose an App to Label Ground Truth Data	14-75
Get Started with the Video Labeler	14-78
Load Unlabeled Data	14-78
Set Time Interval to Label	14-79
Create Label Definitions	14-80
Label Ground Truth	14-87
Export Labeled Ground Truth	14-89
Label Data	14-91
Save App Session	14-92
Use Custom Image Source Reader for Labeling	14-94
Create Custom Reader Function	14-94
Import Data Source into Video Labeler App	14-94
Import Data Source into Ground Truth Labeler App	14-95

Use Sublabels and Attributes to Label Ground Truth Data	14-96
When to Use Sublabels vs. Attributes	14-96
Draw Sublabels	14-97
Copy and Paste Sublabels	14-97
Delete Sublabels	14-98
Sublabel Limitations	14-99
Temporal Automation Algorithms	14-100
Create Temporal Automation Algorithm	14-100
Run Temporal Automation Algorithm	14-100
View Summary of Ground Truth Labels	14-102
View Label Summary	14-102
Compare Selected Labels	14-104
Share and Store Labeled Ground Truth Data	14-106
Share Ground Truth	14-106
Move Ground Truth	14-109
Store Ground Truth	14-110
Keyboard Shortcuts and Mouse Actions for Image Labeler	14-112
Label Definitions	14-112
Image Browsing and Selection	14-112
Labeling Window	14-112
Polyline Drawing	14-113
Polygon Drawing	14-114
Zooming	14-114
Zooming and Panning	14-115
App Sessions	14-115
Keyboard Shortcuts and Mouse Actions for Video Labeler	14-116
Label Definitions	14-116
Frame Navigation and Time Interval Settings	14-116
Labeling Window	14-116
Polyline Drawing	14-117
Polygon Drawing	14-118
Zooming and Panning	14-118
App Sessions	14-118
Point Feature Types	14-120
Functions That Return Points Objects	14-120
Functions That Accept Points Objects	14-122
Local Feature Detection and Extraction	14-126
What Are Local Features?	14-126
Benefits and Applications of Local Features	14-126
What Makes a Good Local Feature?	14-127
Feature Detection and Feature Extraction	14-127
Choose a Feature Detector and Descriptor	14-128
Use Local Features	14-129
Image Registration Using Multiple Features	14-135
Train a Cascade Object Detector	14-143
Why Train a Detector?	14-143
What Kinds of Objects Can You Detect?	14-143

How Does the Cascade Classifier Work?	14-143
Create a Cascade Classifier Using the trainCascadeObjectDetector ..	14-144
Troubleshooting	14-147
Examples	14-149
Train Stop Sign Detector	14-153
Train Optical Character Recognition for Custom Fonts	14-156
Open the OCR Trainer App	14-156
Train OCR	14-156
App Controls	14-158
Troubleshoot ocr Function Results	14-160
Performance Options with the ocr Function	14-160
Create a Custom Feature Extractor	14-161
Example of a Custom Feature Extractor	14-161
Image Retrieval with Bag of Visual Words	14-164
Retrieval System Workflow	14-165
Evaluate Image Retrieval	14-166
Image Classification with Bag of Visual Words	14-167
Step 1: Set Up Image Category Sets	14-167
Step 2: Create Bag of Features	14-167
Step 3: Train an Image Classifier With Bag of Visual Words	14-168
Step 4: Classify an Image or Image Set	14-169

Motion Estimation and Tracking

15

Multiple Object Tracking	15-2
Detection	15-2
Prediction	15-2
Data Association	15-3
Track Management	15-4
Video Mosaicking	15-5
Pattern Matching	15-10
Pattern Matching	15-15

Geometric Transformations

16

Nearest Neighbor, Bilinear, and Bicubic Interpolation Methods	16-2
Nearest Neighbor Interpolation	16-2
Bilinear Interpolation	16-3
Bicubic Interpolation	16-3

Filters, Transforms, and Enhancements

17

Adjust the Contrast of Intensity Images	17-2
Adjust the Contrast of Color Images	17-6
Remove Salt and Pepper Noise from Images	17-10
Sharpen an Image	17-14

Statistics and Morphological Operations

18

Correct Nonuniform Illumination	18-2
Count Objects in an Image	18-8

Fixed-Point Design

19

Fixed-Point Signal Processing	19-2
Fixed-Point Features	19-2
Benefits of Fixed-Point Hardware	19-2
Benefits of Fixed-Point Design with System Toolboxes Software	19-2
Fixed-Point Concepts and Terminology	19-4
Fixed-Point Data Types	19-4
Scaling	19-5
Precision and Range	19-6
Arithmetic Operations	19-8
Modulo Arithmetic	19-8
Two's Complement	19-8
Addition and Subtraction	19-9
Multiplication	19-10
Casts	19-12
Fixed-Point Support for MATLAB System Objects	19-15
Getting Information About Fixed-Point System Objects	19-15
Setting System Object Fixed-Point Properties	19-15
Specify Fixed-Point Attributes for Blocks	19-17
Fixed-Point Block Parameters	19-17
Specify System-Level Settings	19-19
Inherit via Internal Rule	19-19
Specify Data Types for Fixed-Point Blocks	19-26

Code Generation and Shared Library

20

Simulink Shared Library Dependencies	20-2
Accelerating Simulink Models	20-3
Portable C Code Generation for Functions That Use OpenCV Library ..	20-4
Limitations	20-4

Vision Blocks Examples

21

Generate Image Histogram	21-2
Export Image to MATLAB Workspace	21-4
Import Video from MATLAB Workspace	21-7
Find Minimum Value in ROI	21-9
Write Image to Binary File	21-13
Compute Standard Deviation of ROIs	21-14
Read Video Stored as Binary Data	21-17
Compare Image Quality Using PSNR	21-21
Compute Autocorrelation of Input Matrix	21-23
Compute Correlation between Two Matrices	21-24
Find Statistics of Circular Blobs in Image	21-25
Replace Intensity Values in ROI with its Maximum Value	21-29
Median based Image Thresholding	21-33
Import Image From MATLAB Workspace	21-36
Import Image from Specified Location	21-38
Remove Interlacing Effect From Image	21-42
Estimate Motion between Two Images	21-45

Camera Calibration Examples

- “Camera Calibration Using AprilTag Markers” on page 1-2
- “Configure Monocular Fisheye Camera” on page 1-13
- “Monocular Visual Simultaneous Localization and Mapping” on page 1-18
- “Structure From Motion From Two Views” on page 1-37
- “Evaluating the Accuracy of Single Camera Calibration” on page 1-47
- “Measuring Planar Objects with a Calibrated Camera” on page 1-52
- “Depth Estimation From Stereo Video” on page 1-61
- “Structure From Motion From Multiple Views” on page 1-70
- “Uncalibrated Stereo Image Rectification” on page 1-78

Camera Calibration Using AprilTag Markers

AprilTags are widely used as visual markers for applications in object detection, localization, and as a target for camera calibration [1]. AprilTags are similar to QR codes, but are designed to encode less data, and can therefore be decoded faster which is useful, for example, for real-time robotics applications.

This example uses the `readAprilTag` function to detect and localize AprilTags in a calibration pattern. The `readAprilTag` function supports all official tag families. The example also uses additional Computer Vision Toolbox™ functions to perform end-to-end camera calibration. The default checkerboard pattern is replaced by a grid of evenly spaced AprilTags. For an example of using a checkerboard pattern for calibration, refer to “Single Camera Calibrator App” on page 13-8.

The advantages of using AprilTags as a calibration pattern are: more robust feature point detection, consistent and repeatable detections. This example can also serve as a template for using other custom calibration patterns such as a field of circles instead of a typical checkerboard pattern.

Step 1: Generate the calibration pattern

Download and prepare tag images

Pre-generated tags for all the supported families can be downloaded from here using a web browser or by running the following code:

```
downloadURL = 'https://github.com/AprilRobotics/apriltag-imgs/archive/master.zip';
dataFolder = fullfile(tempdir, 'apriltag-imgs', filesep);
options     = weboptions('Timeout', Inf);
zipFileName = [dataFolder, 'apriltag-imgs-master.zip'];
folderExists = exist(dataFolder, 'dir');

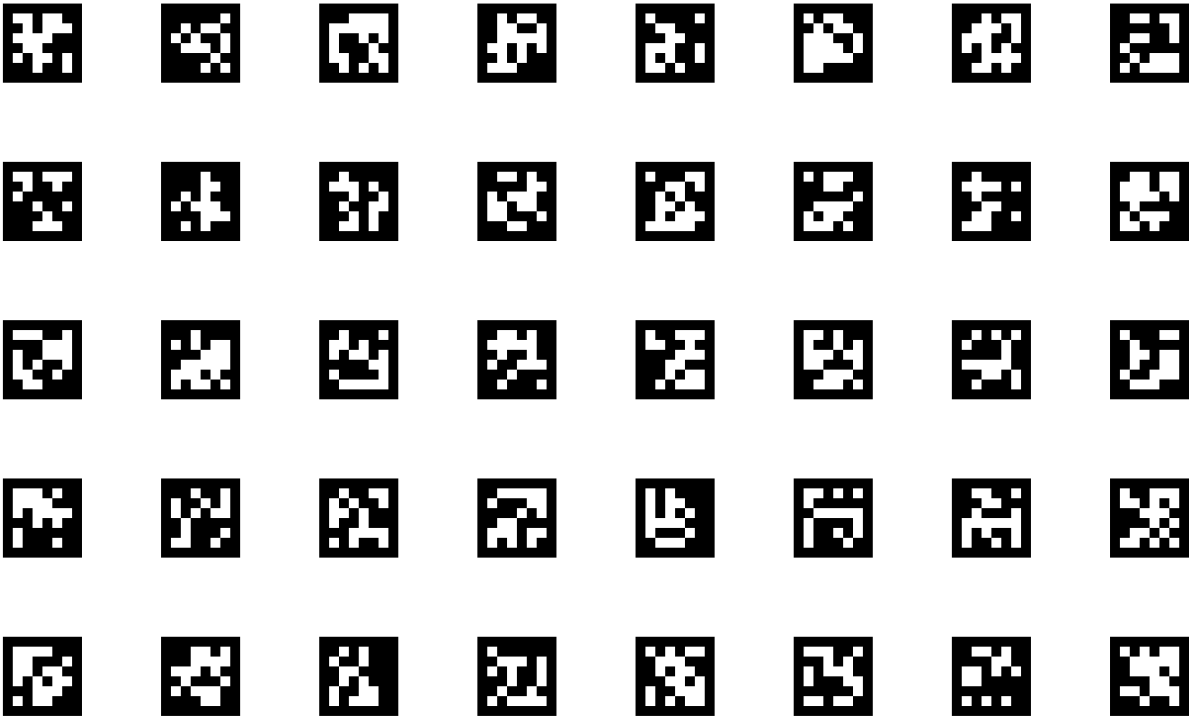
% Create a folder in a temporary directory to save the downloaded file
if ~folderExists
    mkdir(dataFolder);
    disp('Downloading apriltag-imgs-master.zip (60.1 MB)...')
    websave(zipFileName, downloadURL, options);

    % Extract contents of the downloaded file
    disp('Extracting apriltag-imgs-master.zip...')
    unzip(zipFileName, dataFolder);
end
```

The `helperGenerateAprilTagPattern` function at the end of the example can be used to generate a calibration target using the tag images for a specific arrangement of tags. The pattern image is contained in `calibPattern`, which can be used to print the pattern (from MATLAB). The example uses the **tag36h11** family, which provides a reasonable trade-off between detection performance and robustness to false-positive detections.

```
% Set the properties of the calibration pattern.
tagArrangement = [5,8];
tagFamily = 'tag36h11';

% Generate the calibration pattern using AprilTags.
tagImageFolder = [dataFolder 'apriltag-imgs-master/' tagFamily];
imdsTags = imageDatastore(tagImageFolder);
calibPattern = helperGenerateAprilTagPattern(imdsTags, tagArrangement, tagFamily);
```



Using the `readAprilTag` function on this pattern results in detections with the corner locations of the individual tags grouped together. The `helperAprilTagToCheckerLocations` on page 1-0 function can be used to convert this arrangement to a column-major arrangement similar to a checkerboard.

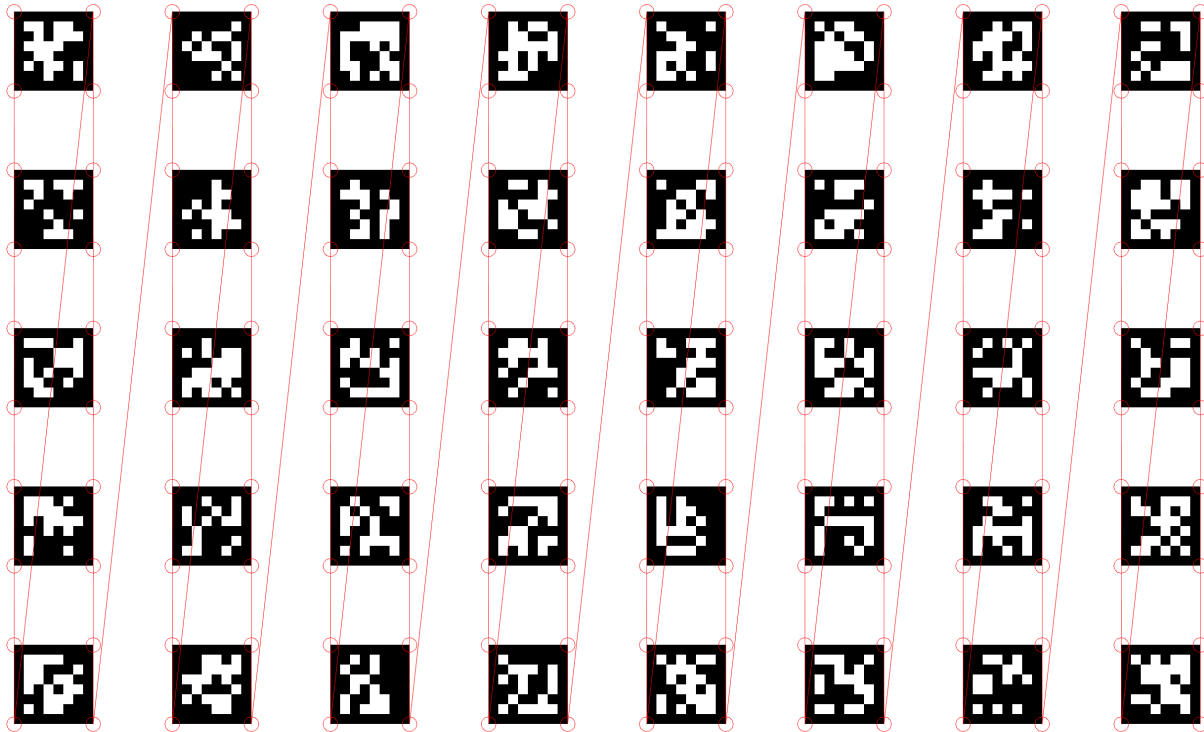
```
% Read and localize the tags in the calibration pattern.
[tagIds, tagLocs] = readAprilTag(calibPattern, tagFamily);

% Sort the tags based on their ID values.
[~, sortIdx] = sort(tagIds);
tagLocs = tagLocs(:, :, sortIdx);

% Reshape the tag corner locations into an M-by-2 array.
tagLocs = reshape(permute(tagLocs, [1,3,2]), [], 2);

% Convert the AprilTag corner locations to checkerboard corner locations.
checkerIdx = helperAprilTagToCheckerLocations(tagArrangement);
imagePoints = tagLocs(checkerIdx(:), :);

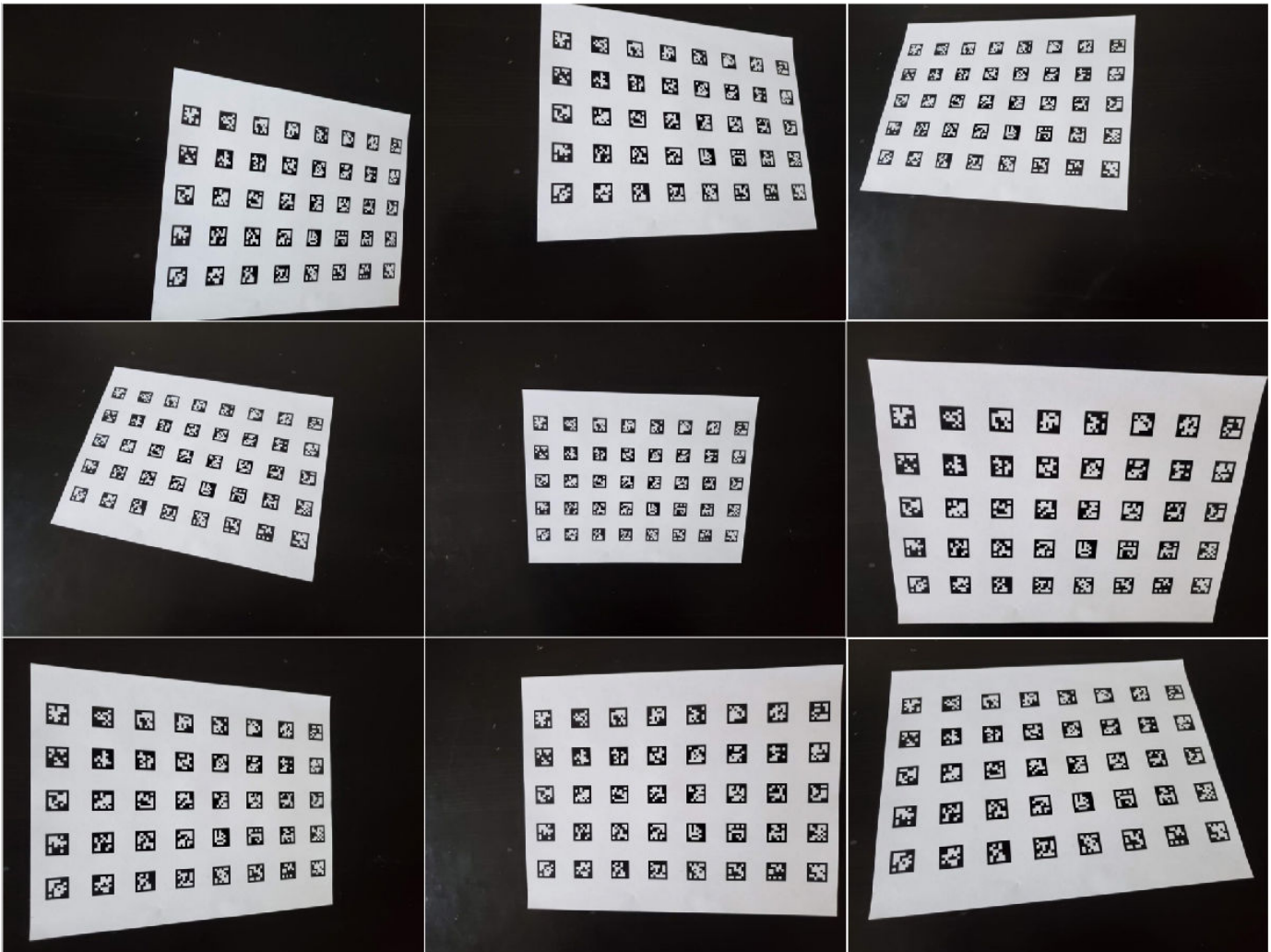
% Display corner locations.
figure; imshow(calibPattern); hold on
plot(imagePoints(:,1), imagePoints(:,2), 'ro-', 'MarkerSize', 15)
```



Prepare images for calibration

The generated calibration pattern must be printed on a flat surface and the camera that needs to be calibrated is used to capture images of the pattern. A few points to note while preparing images for calibration:

- While the pattern is printed on a paper in this example, consider printing it on a surface that remains flat, and is not subject to deformations due to moisture, etc.
- Since the calibration procedure assumes that the pattern is planar, any imperfections in the pattern (eg. an uneven surface) can reduce the accuracy of the calibration.
- The calibration procedure requires at least 2 images of the pattern but using between 10 and 20 images produces more accurate results.
- Capture a variety of images of the pattern such that the pattern fills most of the image, thus covering the entire field of view. For example, to best capture the lens distortion, have images of the pattern at all edges of the image frame.
- Make sure the pattern is completely visible in the captured images, since images with partially visible patterns will be rejected.
- For more information on preparing images of the calibration pattern, “Prepare the Pattern, Camera, and Images” on page 13-9.



Step 2: Detect and localize the AprilTags

The `helperDetectAprilTagCorners` function, included at the end of the example, is used to detect and localize the tags from the captured images and arrange them in a checkerboard fashion to be used as key points in the calibration procedure.

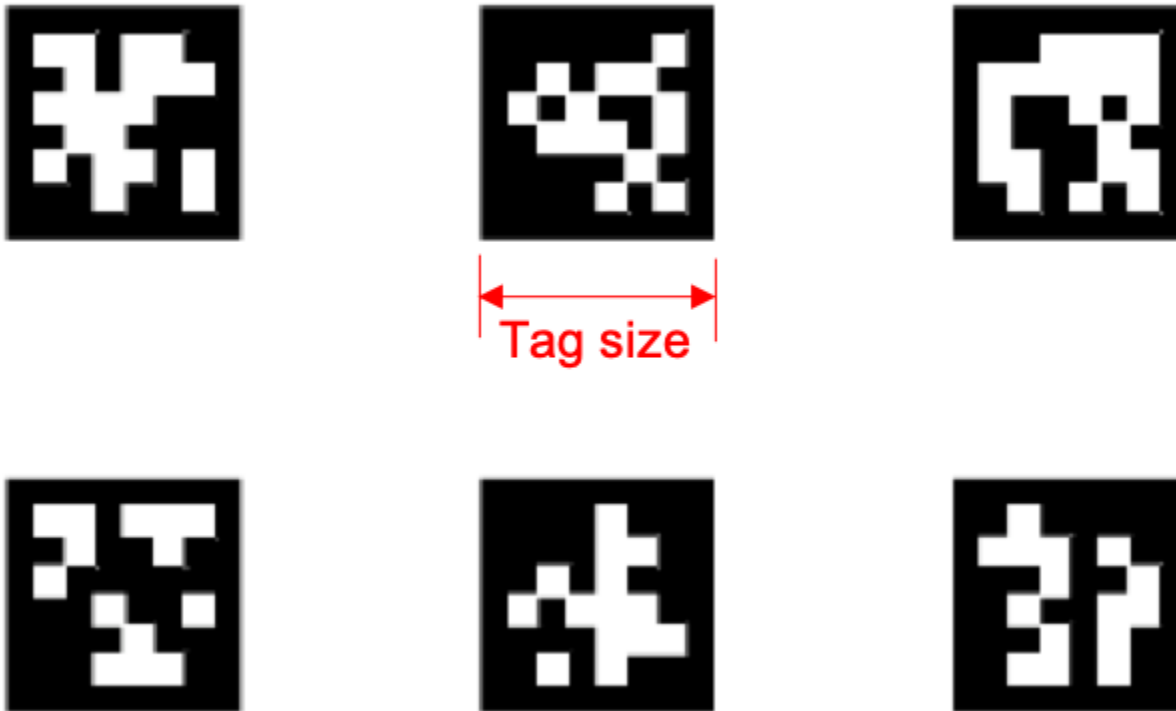
```
% Create an imageDatastore object to store the captured images.
imdsCalib = imageDatastore("calibImages/");

% Detect the calibration pattern from the images.
[imagePoints, boardSize] = helperDetectAprilTagCorners(imdsCalib, tagArrangement, tagFamily);
```

Step 3: Generate world points for the calibration pattern

The generated AprilTag pattern is such that the tags are located in a checkerboard fashion, and so the world coordinates for the corresponding image coordinates determined above (in `imagePoints`) can be obtained using the `generateCheckerboardPoints` function.

Here, the size of the square is replaced by the size of the tag and the size of the board is obtained from the previous step. Measure the tag size between the outer black edges of one side of the tag.



```
% Generate world point coordinates for the pattern.  
tagSize = 16; % in millimeters  
worldPoints = generateCheckerboardPoints(boardSize, tagSize);
```

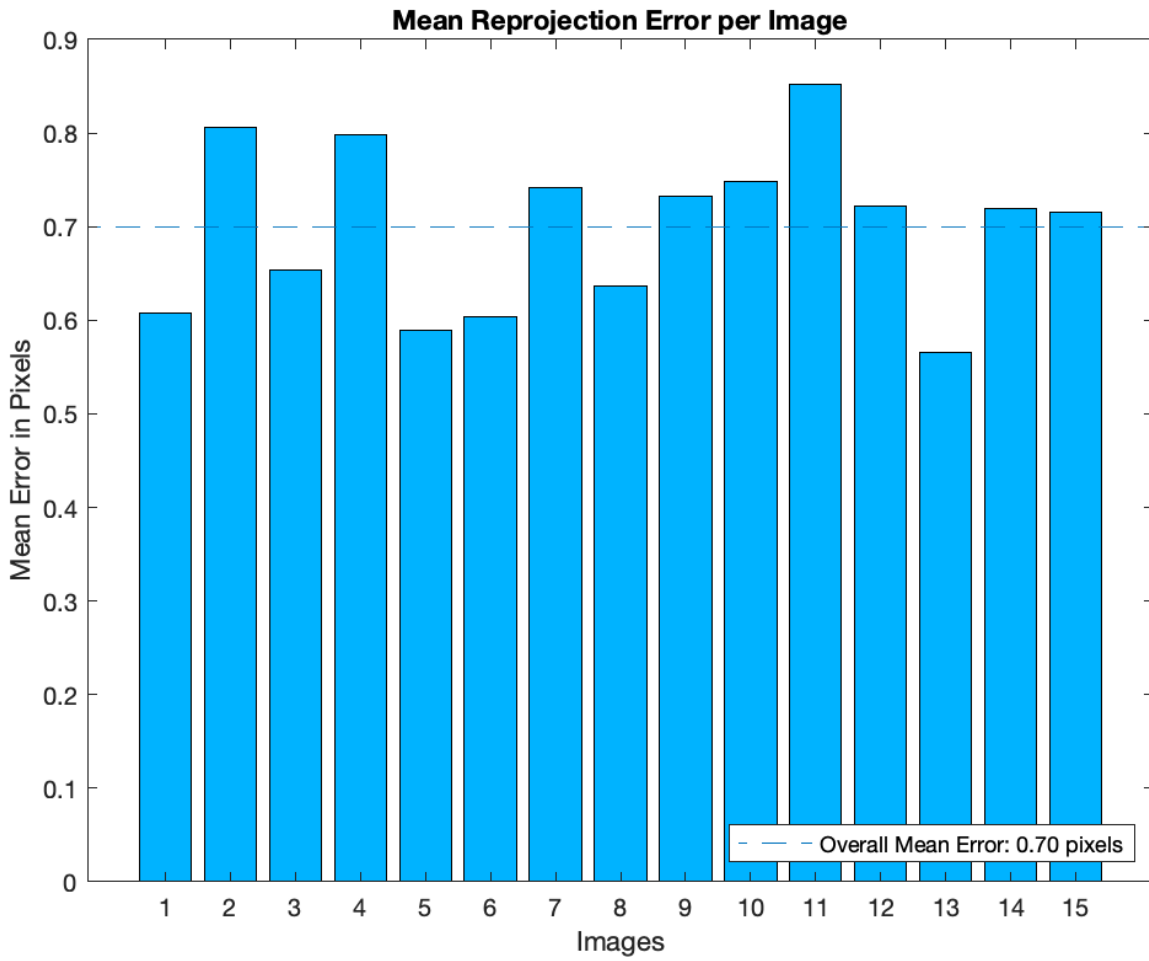
Step 4: Estimate camera parameters

With the image and world point correspondences, estimate the camera parameters using the `estimateCameraParameters` function.

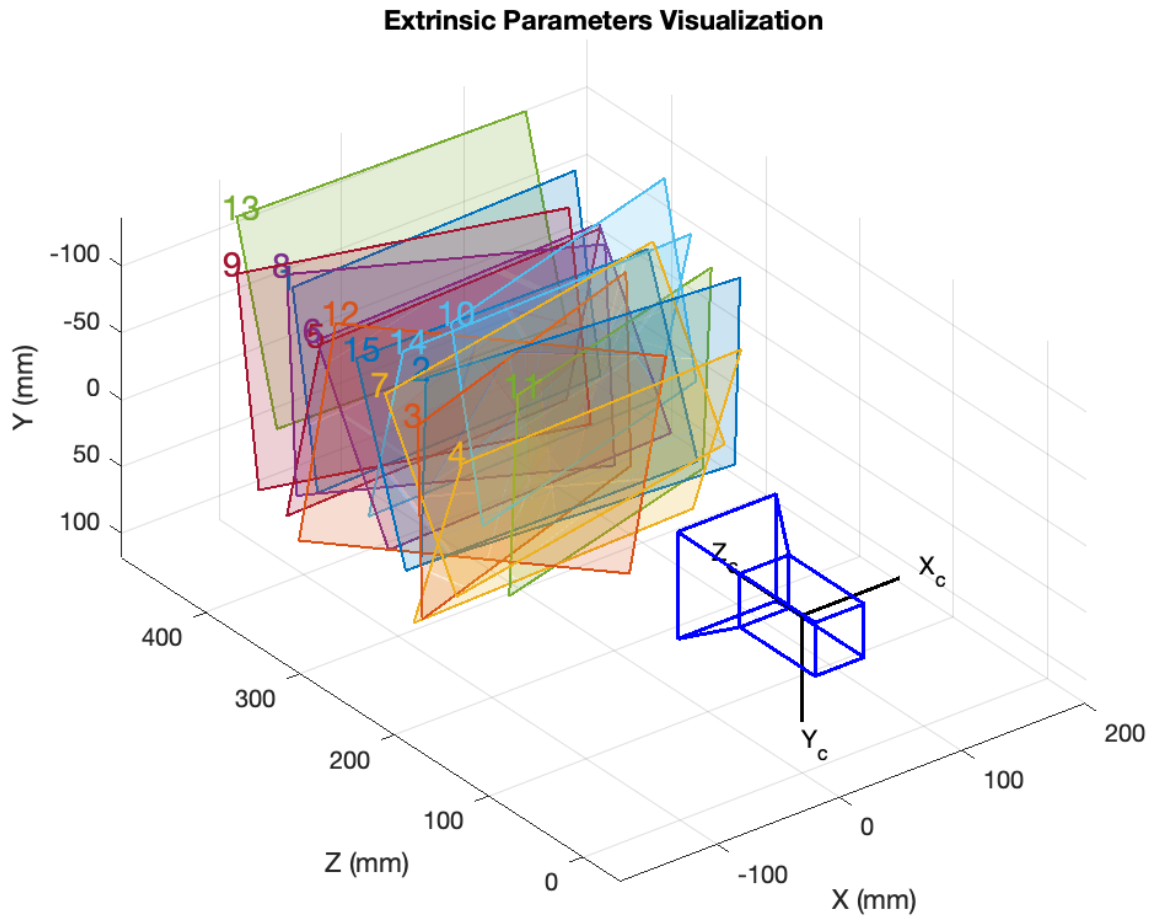
```
% Determine the size of the images.  
I = readimage(imdsCalib, 1);  
imageSize = [size(I,1), size(I,2)];  
  
% Estimate the camera parameters.  
params = estimateCameraParameters(imagePoints, worldPoints, 'ImageSize', imageSize);
```

Visualize the accuracy of the calibration and the extrinsic camera parameters showing the planes of the calibration pattern in the captured images.

```
% Display the reprojection errors.  
figure  
showReprojectionErrors(params)
```

```
% Display the extrinsics.  
figure  
showExtrinsics(params)
```

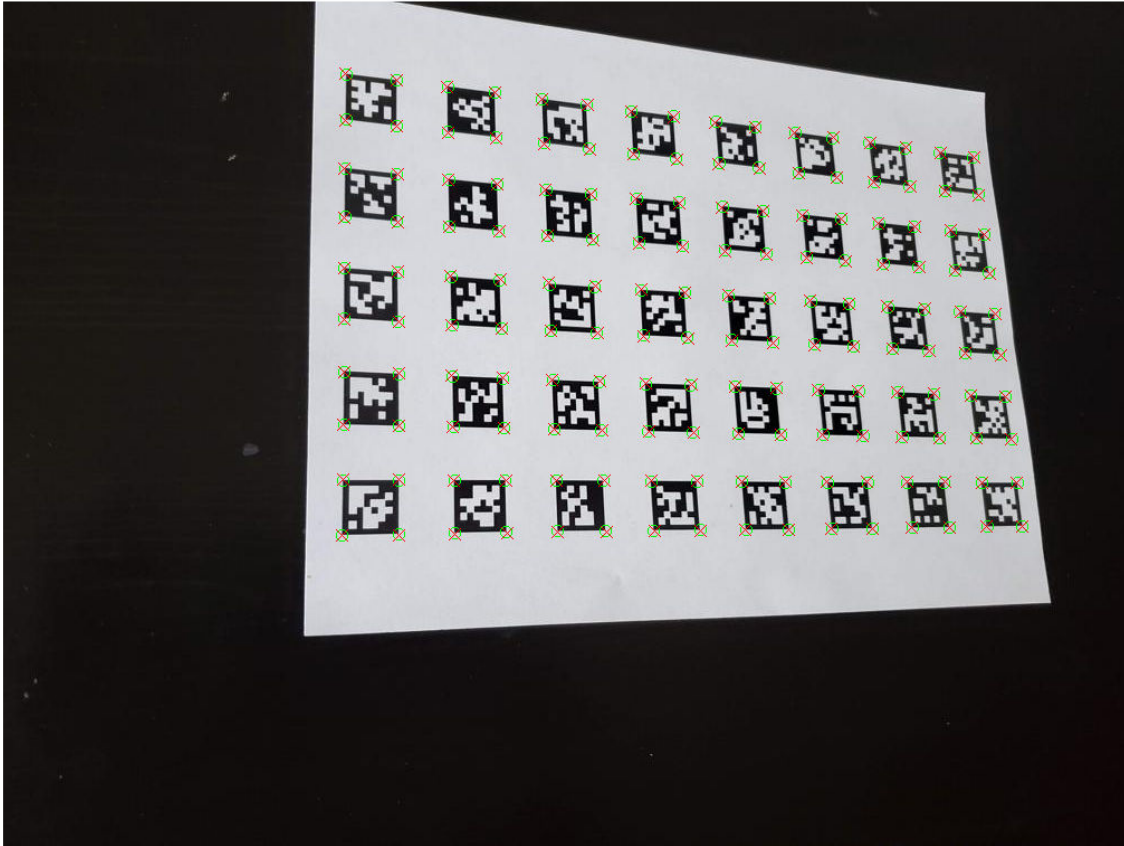


Inspect the locations of the detected image points and the reprojected points obtained using the estimated camera parameters.

```
% Read a calibration image.
I = readimage(imdsCalib, 10);

% Insert markers for the detected and reprojected points.
I = insertMarker(I, imagePoints(:,:,10), 'o', 'Color', 'g', 'Size', 5);
I = insertMarker(I, params.ReprojectedPoints(:,:,10), 'x', 'Color', 'r', 'Size', 5);

% Display the image.
figure
imshow(I)
```



Using other Calibration Patterns

While this example uses AprilTags markers in the calibration pattern, the same workflow can be extended to other planar patterns as well. The `estimateCameraParameters` used to obtain the camera parameters requires:

- **imagePoints:** Key points in the calibration pattern in image coordinates obtained from the captured images.
- **worldPoints:** Corresponding world point coordinates of the key points in the calibration pattern.

Provided there is a way to obtain these key points, the rest of the calibration workflow remains the same.

Supporting functions

`helperGenerateAprilTagPattern` generates an AprilTag based calibration pattern.

```
function calibPattern = helperGenerateAprilTagPattern(imdsTags, tagArrangement, tagFamily)
```

```
numTags = tagArrangement(1)*tagArrangement(2);
tagIds = zeros(1,numTags);
```

```
% Read the first image.
```

```
I = readimage(imdsTags, 3);
Igray = rgb2gray(I);

% Scale up the thumbnail tag image.
Ires = imresize(Igray, 15, 'nearest');

% Detect the tag ID and location (in image coordinates).
[tagIds(1), tagLoc] = readAprilTag(Ires, tagFamily);

% Pad image with white boundaries (ensures the tags replace the black
% portions of the checkerboard).
tagSize = round(max(tagLoc(:,2)) - min(tagLoc(:,2)));
padSize = round(tagSize/2 - (size(Ires,2) - tagSize)/2);
Ires = padarray(Ires, [padSize,padSize], 255);

% Initialize tagImages array to hold the scaled tags.
tagImages = zeros(size(Ires,1), size(Ires,2), numTags);
tagImages(:,:,1) = Ires;

for idx = 2:numTags

    I = readimage(imdsTags, idx + 2);
    Igray = rgb2gray(I);
    Ires = imresize(Igray, 15, 'nearest');
    Ires = padarray(Ires, [padSize,padSize], 255);

    tagIds(idx) = readAprilTag(Ires, tagFamily);

    % Store the tag images.
    tagImages(:,:,idx) = Ires;

end

% Sort the tag images based on their IDs.
[~, sortIdx] = sort(tagIds);
tagImages = tagImages(:,:,sortIdx);

% Reshape the tag images to ensure that they appear in column-major order
% (montage function places image in row-major order).
columnMajIdx = reshape(1:numTags, tagArrangement)';
tagImages = tagImages(:,:,columnMajIdx(:));

% Create the pattern using 'montage'.
imgData = montage(tagImages, 'Size', tagArrangement);
calibPattern = imgData.CData;

end
```

helperDetectAprilTagCorners detects AprilTag calibration pattern in images.

```
function [imagePoints, boardSize, imagesUsed] = helperDetectAprilTagCorners(imdsCalib, tagArrangement)

% Get the pattern size from tagArrangement.
boardSize = tagArrangement*2 + 1;

% Initialize number of images and tags.
numImages = length(imdsCalib.Files);
numTags = tagArrangement(1)*tagArrangement(2);
```

```

% Initialize number of corners in AprilTag pattern.
imagePoints = zeros(numTags*4,2,numImages);
imagesUsed = zeros(1, numImages);

% Get checkerboard corner indices from AprilTag corners.
checkerIdx = helperAprilTagToCheckerLocations(tagArrangement);

for idx = 1:numImages

    % Read and detect AprilTags in image.
    I = readimage(imdsCalib, idx);
    [tagIds, tagLocs] = readAprilTag(I, tagFamily);

    % Accept images if all tags are detected.
    if numel(tagIds) == numTags
        % Sort detected tags using ID values.
        [~, sortIdx] = sort(tagIds);
        tagLocs = tagLocs(:, :, sortIdx);

        % Reshape tag corner locations into a M-by-2 array.
        tagLocs = reshape(permute(tagLocs, [1,3,2]), [], 2);

        % Populate imagePoints using checkerboard corner indices.
        imagePoints(:, :, idx) = tagLocs(checkerIdx(:, :), :);
        imagesUsed(idx) = true;
    else
        imagePoints(:, :, idx) = [];
    end
end

end
end

```

helperAprilTagToCheckerLocations converts AprilTag corners to checkerboard corners.

```

function checkerIdx = helperAprilTagToCheckerLocations(tagArrangement)

numTagRows = tagArrangement(1);
numTagCols = tagArrangement(2);
numTags = numTagRows * numTagCols;

% Row index offsets.
rowIdxOffset = [0:numTagRows - 1; 0:numTagRows - 1];

% Row indices for first and second columns in board.
col1Idx = repmat([4 1]', numTagRows, 1);
col2Idx = repmat([3 2]', numTagRows, 1);
col1Idx = col1Idx + rowIdxOffset(:)*4;
col2Idx = col2Idx + rowIdxOffset(:)*4;

% Column index offsets
colIdxOffset = 0:4*numTagRows:numTags*4 - 1;

% Implicit expansion to get all indices in order.
checkerIdx = [col1Idx;col2Idx] + colIdxOffset;

end

```

Reference

[1] E. Olson, "AprilTag: A robust and flexible visual fiducial system," *2011 IEEE International Conference on Robotics and Automation*, Shanghai, 2011, pp. 3400-3407, doi: 10.1109/ICRA.2011.5979561.

Configure Monocular Fisheye Camera

This example shows how to convert a fisheye camera model to a pinhole model and construct a corresponding monocular camera sensor simulation. In this example, you learn how to calibrate a fisheye camera and configure a `monoCamera` (Automated Driving Toolbox) object.

Overview

To simulate a monocular camera sensor mounted in a vehicle, follow these steps:

- 1 Estimate the intrinsic camera parameters by calibrating the camera using a checkerboard. The intrinsic parameters describe the properties of the fisheye camera itself.
- 2 Estimate the extrinsic camera parameters by calibrating the camera again, using the same checkerboard from the previous step. The extrinsic parameters describe the mounting position of the fisheye camera in the vehicle coordinate system.
- 3 Remove image distortion by converting the fisheye camera intrinsics to pinhole camera intrinsics. These intrinsics describe a synthetic pinhole camera that can hypothetically generate undistorted images.
- 4 Use the intrinsic pinhole camera parameters and the extrinsic parameters to configure the monocular camera sensor for simulation. You can then use this sensor to detect objects and lane boundaries.

Estimate Fisheye Camera Intrinsics

To estimate the intrinsic parameters, use a checkerboard for camera calibration. Alternatively, to better visualize the results, use the Camera Calibrator app. For fisheye camera, it is useful to place the checkerboard close to the camera, in order to capture large noticeable distortion in the image.

```
% Gather a set of calibration images.
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', ...
    'calibration', 'gopro'));
imageFileNames = images.Files;

% Detect calibration pattern.
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);

% Generate world coordinates of the corners of the squares.
squareSize = 0.029; % Square size in meters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Calibrate the camera.
I = readimage(images, 1);
imageSize = [size(I, 1), size(I, 2)];
params = estimateFisheyeParameters(imagePoints, worldPoints, imageSize);
```

Estimate Fisheye Camera Extrinsics

To estimate the extrinsic parameters, use the same checkerboard to estimate the mounting position of the camera in the vehicle coordinate system. The following step estimates the parameters from one image. You can also take multiple checkerboard images to obtain multiple estimations, and average the results.

```
% Load a different image of the same checkerboard, where the checkerboard
% is placed on the flat ground. Its X-axis is pointing to the right of the
% vehicle, and its Y-axis is pointing to the camera. The image includes
```

```
% noticeable distortion, such as along the wall next to the checkerboard.  
  
imageFileName = fullfile(toolboxdir('driving'), 'drivingdata', 'checkerboard.png');  
I = imread(imageFileName);  
imshow(I)  
title('Distorted Checkerboard Image');
```

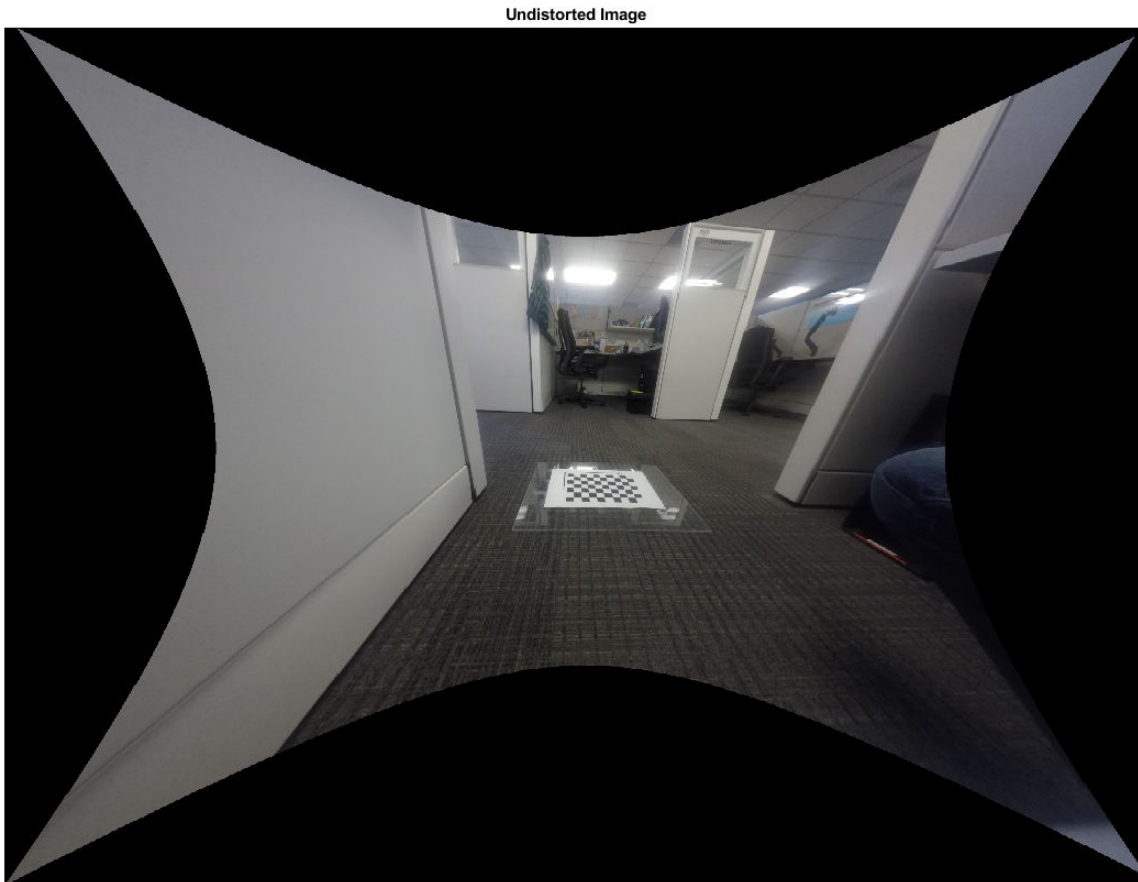
Distorted Checkerboard Image



```
[imagePoints, boardSize] = detectCheckerboardPoints(I);  
  
% Generate coordinates of the corners of the squares.  
squareSize = 0.029; % Square size in meters  
worldPoints = generateCheckerboardPoints(boardSize, squareSize);  
  
% Estimate the parameters for configuring the monoCamera object.  
% Height of the checkerboard is zero here, since the pattern is  
% directly on the ground.  
originHeight = 0;  
[pitch, yaw, roll, height] = estimateMonoCameraParameters(params.Intrinsics, ...  
    imagePoints, worldPoints, originHeight);
```


Construct a Synthetic Pinhole Camera for the Undistorted Image

```
% Undistort the image and extract the synthetic pinhole camera intrinsics.
[J1, camIntrinsics] = undistortFisheyeImage(I, params.Intrinsics, 'Output', 'full');
imshow(J1)
title('Undistorted Image');
```



```
% Set up monoCamera with the synthetic pinhole camera intrinsics.
% Note that the synthetic camera has removed the distortion.
sensor = monoCamera(camIntrinsics, height, 'pitch', pitch, 'yaw', yaw, 'roll', roll);
```

Plot Bird's Eye View

Now you can validate the `monoCamera` (Automated Driving Toolbox) by plotting a bird's-eye view.

```
% Define bird's-eye-view transformation parameters
distAheadOfSensor = 6; % in meters
spaceToOneSide = 2.5; % look 2.5 meters to the right and 2.5 meters to the left
bottomOffset = 0.2; % look 0.2 meters ahead of the sensor
outView = [bottomOffset, distAheadOfSensor, -spaceToOneSide, spaceToOneSide];
outImageSize = [NaN, 1000]; % output image width in pixels

birdsEyeConfig = birdsEyeView(sensor, outView, outImageSize);
```

```
% Transform input image to bird's-eye-view image and display it
B = transformImage(birdsEyeConfig, J1);

% Place a 2-meter marker ahead of the sensor in bird's-eye view
imagePoint0 = vehicleToImage(birdsEyeConfig, [2, 0]);
offset = 5; % offset marker from text label by 5 pixels
annotatedB = insertMarker(B, imagePoint0 - offset);
annotatedB = insertText(annotatedB, imagePoint0, '2 meters');

figure
imshow(annotatedB)
title('Bird''s-Eye View')
```

Bird's-Eye View



The plot above shows that the camera measures distances accurately. Now you can use the monocular camera for object and lane boundary detection. See the “Visual Perception Using Monocular Camera” (Automated Driving Toolbox) example.

Monocular Visual Simultaneous Localization and Mapping

Visual simultaneous localization and mapping (vSLAM), refers to the process of calculating the position and orientation of a camera with respect to its surroundings, while simultaneously mapping the environment. The process uses only visual inputs from the camera. Applications for vSLAM include augmented reality, robotics, and autonomous driving.

This example shows how to process image data from a monocular camera to build a map of an indoor environment and estimate the trajectory of the camera. The example uses ORB-SLAM [1] on page 1-0 , which is a feature-based vSLAM algorithm.

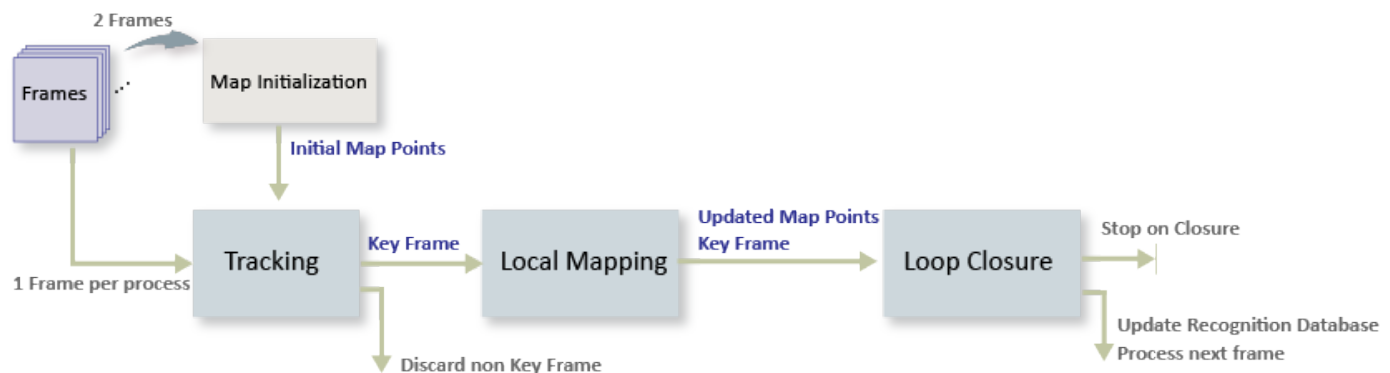
To speed up computations, you can enable parallel computing from the “Computer Vision Toolbox Preferences” dialog box. To open Computer Vision Toolbox™ preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Computer Vision Toolbox**.

Glossary

The following terms are frequently used in this example:

- **Key Frames:** A subset of video frames that contain cues for localization and tracking. Two consecutive key frames usually involve sufficient visual change.
- **Map Points:** A list of 3-D points that represent the map of the environment reconstructed from the key frames.
- **Covisibility Graph:** A graph consisting of key frame as nodes. Two key frames are connected by an edge if they share common map points. The weight of an edge is the number of shared map points.
- **Essential Graph:** A subgraph of covisibility graph containing only edges with high weight, i.e. more shared map points.
- **Recognition Database:** A database used to recognize whether a place has been visited in the past. The database stores the visual word-to-image mapping based on the input bag of features. It is used to search for an image that is visually similar to a query image.

Overview of ORB-SLAM



The ORB-SLAM pipeline includes:

- **Map Initialization:** ORB-SLAM starts by initializing the map of 3-D points from two video frames. The 3-D points and relative camera pose are computed using triangulation based on 2-D ORB feature correspondences.

- **Tracking:** Once a map is initialized, for each new frame, the camera pose is estimated by matching features in the current frame to features in the last key frame. The estimated camera pose is refined by tracking the local map.
- **Local Mapping:** The current frame is used to create new 3-D map points if it is identified as a key frame. At this stage, bundle adjustment is used to minimize reprojection errors by adjusting the camera pose and 3-D points.
- **Loop Closure:** Loops are detected for each key frame by comparing it against all previous key frames using the bag-of-features approach. Once a loop closure is detected, the pose graph is optimized to refine the camera poses of all the key frames.

Download and Explore the Input Image Sequence

The data used in this example are from the TUM RGB-D benchmark [2] on page 1-0 . You can download the data to a temporary directory using a web browser or by running the following code:

```
baseDownloadURL = 'https://vision.in.tum.de/rgbd/dataset/freiburg3/rgbd_dataset_freiburg3_long_office_household';
dataFolder      = fullfile(tempdir, 'tum_rgb_d_dataset', filesep);
options         = weboptions('Timeout', Inf);
tgzFileName     = [dataFolder, 'fr3_office.tgz'];
folderExists    = exist(dataFolder, 'dir');

% Create a folder in a temporary directory to save the downloaded file
if ~folderExists
    mkdir(dataFolder);
    disp('Downloading fr3_office.tgz (1.38 GB). This download can take a few minutes.')
    websave(tgzFileName, baseDownloadURL, options);

    % Extract contents of the downloaded file
    disp('Extracting fr3_office.tgz (1.38 GB) ...')
    untar(tgzFileName, dataFolder);
end
```

Create an `imageDatastore` object to inspect the RGB images.

```
imageFolder = [dataFolder, 'rgbd_dataset_freiburg3_long_office_household/rgb/'];
imds        = imageDatastore(imageFolder);

% Inspect the first image
currFrameIdx = 1;
currI = readimage(imds, currFrameIdx);
himage = imshow(currI);
```

Map Initialization

The ORB-SLAM pipeline starts by initializing the map that holds 3-D world points. This step is crucial and has a significant impact on the accuracy of final SLAM result. Initial ORB feature point correspondences are found using `matchFeatures` between a pair of images. After the correspondences are found, two geometric transformation models are used to establish map initialization:

- **Homography:** If the scene is planar, a homography projective transformation is a better choice to describe feature point correspondences.
- **Fundamental Matrix:** If the scene is non-planar, a fundamental matrix must be used instead.

The homography and the fundamental matrix can be computed using `estimateGeometricTransform2D` and `estimateFundamentalMatrix`, respectively. The model

that results in a smaller reprojection error is selected to estimate the relative rotation and translation between the two frames using `relativeCameraPose`. Since the RGB images are taken by a monocular camera which does not provide the depth information, the relative translation can only be recovered up to a specific scale factor.

Given the relative camera pose and the matched feature points in the two images, the 3-D locations of the matched points are determined using `triangulate` function. A triangulated map point is valid when it is located in the front of both cameras, when its reprojection error is low, and when the parallax of the two views of the point is sufficiently large.

```
% Set random seed for reproducibility
rng(0);

% Create a cameraIntrinsics object to store the camera intrinsic parameters.
% The intrinsics for the dataset can be found at the following page:
% https://vision.in.tum.de/data/datasets/rgbd-dataset/file_formats
% Note that the images in the dataset are already undistorted, hence there
% is no need to specify the distortion coefficients.
focalLength    = [535.4, 539.2];    % in units of pixels
principalPoint = [320.1, 247.6];    % in units of pixels
imageSize      = size(currI,[1 2]); % in units of pixels
intrinsics     = cameraIntrinsics(focalLength, principalPoint, imageSize);

% Detect and extract ORB features
scaleFactor = 1.2;
numLevels   = 8;
[preFeatures, prePoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels);

currFrameIdx = currFrameIdx + 1;
firstI       = currI; % Preserve the first frame

isMapInitialized = false;

% Map initialization loop
while ~isMapInitialized && currFrameIdx < numel(imds.Files)
    currI = readimage(imds, currFrameIdx);

    [currFeatures, currPoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels);

    currFrameIdx = currFrameIdx + 1;

    % Find putative feature matches
    indexPairs = matchFeatures(preFeatures, currFeatures, 'Unique', true, ...
        'MaxRatio', 0.9, 'MatchThreshold', 40);

    preMatchedPoints = prePoints(indexPairs(:,1),:);
    currMatchedPoints = currPoints(indexPairs(:,2),:);

    % If not enough matches are found, check the next frame
    minMatches = 100;
    if size(indexPairs, 1) < minMatches
        continue
    end

    preMatchedPoints = prePoints(indexPairs(:,1),:);
    currMatchedPoints = currPoints(indexPairs(:,2),:);
end
```

```

% Compute homography and evaluate reconstruction
[tformH, scoreH, inliersIdxH] = helperComputeHomography(preMatchedPoints, currMatchedPoints)

% Compute fundamental matrix and evaluate reconstruction
[tformF, scoreF, inliersIdxF] = helperComputeFundamentalMatrix(preMatchedPoints, currMatchedPoints)

% Select the model based on a heuristic
ratio = scoreH/(scoreH + scoreF);
ratioThreshold = 0.45;
if ratio > ratioThreshold
    inlierTformIdx = inliersIdxH;
    tform          = tformH;
else
    inlierTformIdx = inliersIdxF;
    tform          = tformF;
end

% Computes the camera location up to scale. Use half of the
% points to reduce computation
inlierPrePoints = preMatchedPoints(inlierTformIdx);
inlierCurrPoints = currMatchedPoints(inlierTformIdx);
[relOrient, relLoc, validFraction] = relativeCameraPose(tform, intrinsics, ...
    inlierPrePoints(1:2:end), inlierCurrPoints(1:2:end));

% If not enough inliers are found, move to the next frame
if validFraction < 0.9 || numel(size(relOrient))==3
    continue
end

% Triangulate two views to obtain 3-D map points
relPose = rigid3d(relOrient, relLoc);
minParallax = 3; % In degrees
[isValid, xyzWorldPoints, inlierTriangulationIdx] = helperTriangulateTwoFrames(...
    rigid3d, relPose, inlierPrePoints, inlierCurrPoints, intrinsics, minParallax);

if ~isValid
    continue
end

% Get the original index of features in the two key frames
indexPairs = indexPairs(inlierTformIdx(inlierTriangulationIdx),:);

isMapInitialized = true;

disp(['Map initialized with frame 1 and frame ', num2str(currFrameIdx-1)])
end % End of map initialization loop

Map initialized with frame 1 and frame 44

if isMapInitialized
    close(himage.Parent.Parent); % Close the previous figure
    % Show matched features
    hfeature = showMatchedFeatures(firstI, currI, prePoints(indexPairs(:,1)), ...
        currPoints(indexPairs(:, 2)), 'Montage');
else
    error('Unable to initialize map.')
end

```

Store Initial Key Frames and Map Points

After the map is initialized using two frames, you can use `imageviewset`, `worldpointset` and `helperViewDirectionAndDepth` to store the two key frames and the corresponding map points:

- `imageviewset` stores the key frames and their attributes, such as ORB descriptors, feature points and camera poses, and connections between the key frames, such as feature points matches and relative camera poses. The pose of the camera is stored as a `rigid3d` object.
- `worldpointset` stores 3-D positions of the map points and the 3-D into 2-D projection correspondences: which map points are observed in a key frame and which key frames observe a map point.
- `helperViewDirectionAndDepth` stores other attributes of map points, such as the mean view direction, the representative ORB descriptors, and the range of distance at which the map point can be observed.

```
% Create an empty imageviewset object to store key frames
vSetKeyFrames = imageviewset;
```

```
% Create an empty worldpointset object to store 3-D map points
mapPointSet   = worldpointset;
```

```
% Create a helperViewDirectionAndDepth object to store view direction and depth
directionAndDepth = helperViewDirectionAndDepth(size(xyzWorldPoints, 1));
```

```
% Add the first key frame. Place the camera associated with the first
% key frame at the origin, oriented along the Z-axis
preViewId      = 1;
vSetKeyFrames = addView(vSetKeyFrames, preViewId, rigid3d, 'Points', prePoints,...
    'Features', preFeatures.Features);
```

```
% Add the second key frame
currViewId     = 2;
vSetKeyFrames = addView(vSetKeyFrames, currViewId, relPose, 'Points', currPoints,...
    'Features', currFeatures.Features);
```

```
% Add connection between the first and the second key frame
vSetKeyFrames = addConnection(vSetKeyFrames, preViewId, currViewId, relPose, 'Matches', indexPairs);
```

```
% Add 3-D map points
[mapPointSet, newPointIdx] = addWorldPoints(mapPointSet, xyzWorldPoints);
```

```
% Add observations of the map points
preLocations  = prePoints.Location;
currLocations = currPoints.Location;
preScales     = prePoints.Scale;
currScales    = currPoints.Scale;
```

```
% Add image points corresponding to the map points in the first key frame
mapPointSet = addCorrespondences(mapPointSet, preViewId, newPointIdx, indexPairs(:,1));
```

```
% Add image points corresponding to the map points in the second key frame
mapPointSet = addCorrespondences(mapPointSet, currViewId, newPointIdx, indexPairs(:,2));
```

Refine and Visualize the Initial Reconstruction

Refine the initial reconstruction using `bundleAdjustment`, that optimizes both camera poses and world points to minimize the overall reprojection errors. After the refinement, the attributes of the

map points including 3-D locations, view direction, and depth range are updated. You can use `helperVisualizeMotionAndStructure` to visualize the map points and the camera locations.

```
% Run full bundle adjustment on the first two key frames
tracks      = findTracks(vSetKeyFrames);
cameraPoses = poses(vSetKeyFrames);

[refinedPoints, refinedAbsPoses] = bundleAdjustment(xyzWorldPoints, tracks, ...
    cameraPoses, intrinsics, 'FixedViewIDs', 1, ...
    'PointsUndistorted', true, 'AbsoluteTolerance', 1e-7,...
    'RelativeTolerance', 1e-15, 'MaxIteration', 50);

% Scale the map and the camera pose using the median depth of map points
medianDepth = median(vecnorm(refinedPoints.'));
refinedPoints = refinedPoints / medianDepth;

refinedAbsPoses.AbsolutePose(currViewId).Translation = ...
    refinedAbsPoses.AbsolutePose(currViewId).Translation / medianDepth;
relPose.Translation = relPose.Translation/medianDepth;

% Update key frames with the refined poses
vSetKeyFrames = updateView(vSetKeyFrames, refinedAbsPoses);
vSetKeyFrames = updateConnection(vSetKeyFrames, prevViewId, currViewId, relPose);

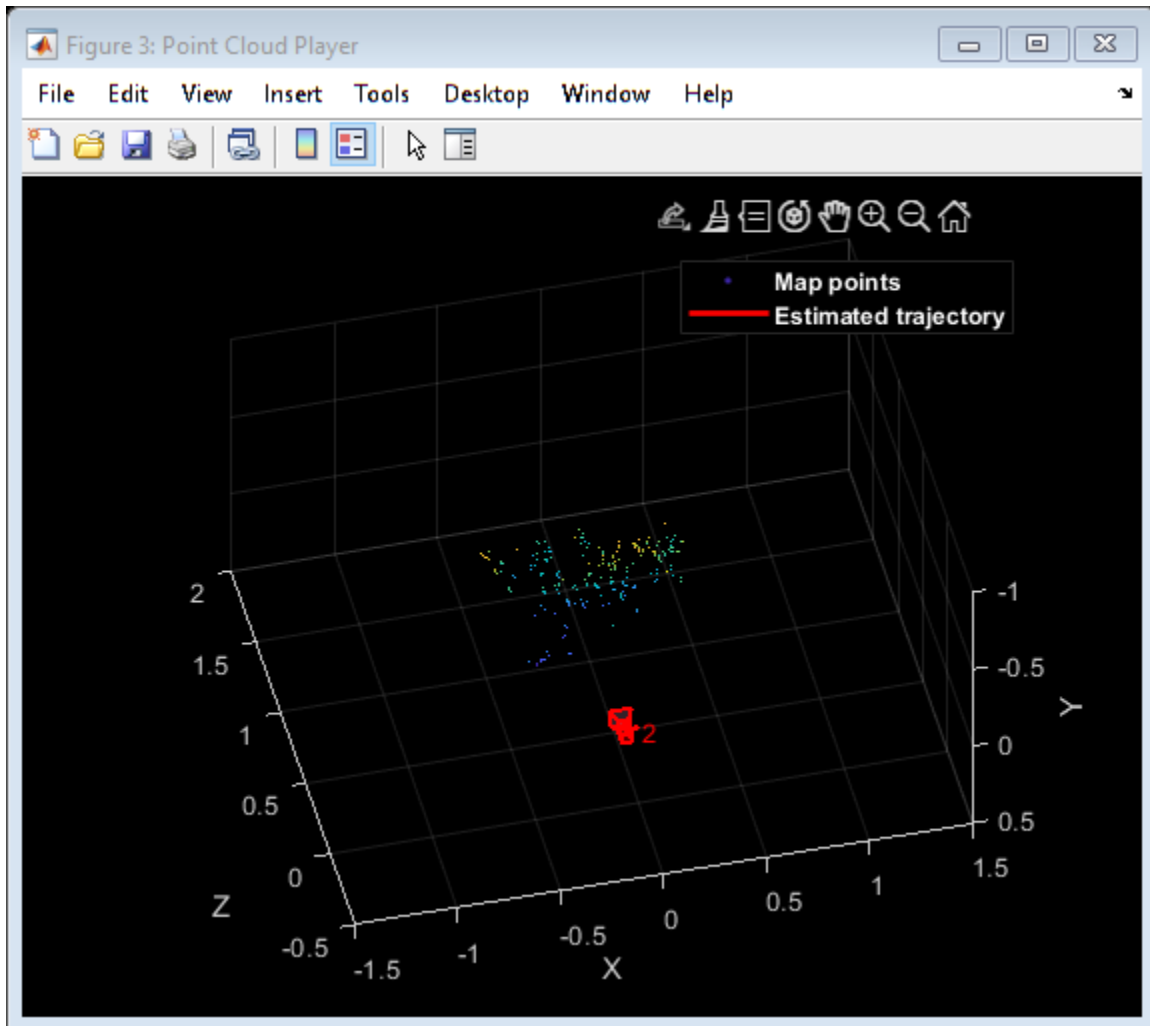
% Update map points with the refined positions
mapPointSet = updateWorldPoints(mapPointSet, newPointIdx, refinedPoints);

% Update view direction and depth
directionAndDepth = update(directionAndDepth, mapPointSet, vSetKeyFrames.Views, newPointIdx, true);

% Visualize matched features in the current frame
close(hfeature.Parent.Parent);
featurePlot = helperVisualizeMatchedFeatures(currI, currPoints(indexPairs(:,2))));
```



```
% Visualize initial map points and camera trajectory  
mapPlot      = helperVisualizeMotionAndStructure(vSetKeyFrames, mapPointSet);  
  
% Show legend  
showLegend(mapPlot);
```



Tracking

The tracking process is performed using every frame and determines when to insert a new key frame. To simplify this example, we will terminate the tracking process once a loop closure is found.

```
% ViewId of the current key frame
currKeyFrameId = currViewId;

% ViewId of the last key frame
lastKeyFrameId = currViewId;

% ViewId of the reference key frame that has the most co-visible
% map points with the current key frame
refKeyFrameId = currViewId;

% Index of the last key frame in the input image sequence
lastKeyFrameIdx = currFrameIdx - 1;

% Indices of all the key frames in the input image sequence
addedFramesIdx = [1; lastKeyFrameIdx];
```

```
isLoopClosed      = false;
```

Each frame is processed as follows:

- 1 ORB features are extracted for each new frame and then matched (using `matchFeatures`), with features in the last key frame that have known corresponding 3-D map points.
- 2 Estimate the camera pose with the Perspective-n-Point algorithm using `estimateWorldCameraPose`.
- 3 Given the camera pose, project the map points observed by the last key frame into the current frame and search for feature correspondences using `helperMatchFeaturesInRadius`.
- 4 With 3-D to 2-D correspondence in the current frame, refine the camera pose by performing a motion-only bundle adjustment using `bundleAdjustmentMotion`.
- 5 Project the local map points into the current frame to search for more feature correspondences using `helperMatchFeaturesInRadius` and refine the camera pose again using `bundleAdjustmentMotion`.
- 6 The last step of tracking is to decide if the current frame is a new key frame. If the current frame is a key frame, continue to the **Local Mapping** process. Otherwise, start **Tracking** for the next frame.

```
% Main loop
```

```
while ~isLoopClosed && currFrameIdx < numel(imds.Files)
    currI = readimage(imds, currFrameIdx);

    [currFeatures, currPoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels);

    % Track the last key frame
    % mapPointsIdx:  Indices of the map points observed in the current frame
    % featureIdx:   Indices of the corresponding feature points in the
    %               current frame
    [currPose, mapPointsIdx, featureIdx] = helperTrackLastKeyFrame(mapPointSet, ...
        vSetKeyFrames.Views, currFeatures, currPoints, lastKeyFrameId, intrinsics, scaleFactor);

    % Track the local map
    % refKeyFrameId:  ViewId of the reference key frame that has the most
    %                co-visible map points with the current frame
    % localKeyFrameIds:  ViewId of the connected key frames of the current frame
    [refKeyFrameId, localKeyFrameIds, currPose, mapPointsIdx, featureIdx] = ...
        helperTrackLocalMap(mapPointSet, directionAndDepth, vSetKeyFrames, mapPointsIdx, ...
            featureIdx, currPose, currFeatures, currPoints, intrinsics, scaleFactor, numLevels);

    % Check if the current frame is a key frame.
    % A frame is a key frame if both of the following conditions are satisfied:
    %
    % 1. At least 20 frames have passed since the last key frame or the
    %    current frame tracks fewer than 80 map points
    % 2. The map points tracked by the current frame are fewer than 90% of
    %    points tracked by the reference key frame
    isKeyFrame = helperIsKeyFrame(mapPointSet, refKeyFrameId, lastKeyFrameIdx, ...
        currFrameIdx, mapPointsIdx);

    % Visualize matched features
    updatePlot(featurePlot, currI, currPoints(featureIdx));

    if ~isKeyFrame
```

```

        currFrameIdx = currFrameIdx + 1;
        continue
    end

    % Update current key frame ID
    currKeyFrameId = currKeyFrameId + 1;

```

Local Mapping

Local mapping is performed for every key frame. When a new key frame is determined, add it to the key frames and update the attributes of the map points observed by the new key frame. To ensure that `mapPointSet` contains as few outliers as possible, a valid map point must be observed in at least 3 key frames.

New map points are created by triangulating ORB feature points in the current key frame and its connected key frames. For each unmatched feature point in the current key frame, search for a match with other unmatched points in the connected key frames using `matchFeatures`. The local bundle adjustment refines the pose of the current key frame, the poses of connected key frames, and all the map points observed in these key frames.

```

% Add the new key frame
[mapPointSet, vSetKeyFrames] = helperAddNewKeyFrame(mapPointSet, vSetKeyFrames, ...
    currPose, currFeatures, currPoints, mapPointsIdx, featureIdx, localKeyFrameIds);

% Remove outlier map points that are observed in fewer than 3 key frames
[mapPointSet, directionAndDepth, mapPointsIdx] = helperCullRecentMapPoints(mapPointSet, dire

% Create new map points by triangulation
minNumMatches = 20;
minParallax = 3;
[mapPointSet, vSetKeyFrames, newPointIdx] = helperCreateNewMapPoints(mapPointSet, vSetKeyFrames,
    currKeyFrameId, intrinsics, scaleFactor, minNumMatches, minParallax);

% Update view direction and depth
directionAndDepth = update(directionAndDepth, mapPointSet, vSetKeyFrames.Views, [mapPointsIdx

% Local bundle adjustment
[mapPointSet, directionAndDepth, vSetKeyFrames, newPointIdx] = helperLocalBundleAdjustment(m
    currKeyFrameId, intrinsics, newPointIdx);

% Visualize 3D world points and camera trajectory
updatePlot(mapPlot, vSetKeyFrames, mapPointSet);

```

Loop Closure

The loop closure step takes the current key frame processed by the local mapping process and tries to detect and close the loop. Loop detection is performed using the bags-of-words approach. A visual vocabulary represented as a `bagOfFeatures` object is created offline with the SURF descriptors extracted from a large set of images in the dataset by calling:

```

bag = bagOfFeatures(imds, 'CustomExtractor',
    @helperSURFFeatureExtractorFunction);

```

where `imds` is an `imageDatastore` object storing the training images and `helperSURFFeatureExtractorFunction` is the SURF feature extractor function. See “Image Retrieval with Bag of Visual Words” on page 14-164 for more information.

The loop closure process incrementally builds a database, represented as an `invertedImageIndex` object, that stores the visual word-to-image mapping based on the bag of SURF features. Loop candidates are identified by querying images in the database that are visually similar to the current key frame using `evaluateImageRetrieval`. A candidate key frame is valid if it is not connected to the last key frame and three of its neighbor key frames are loop candidates.

When a valid loop candidate is found, compute the relative pose between the loop candidate frame and the current key frame using the same strategy as the one used in the **Tracking** process. Then add the loop connection with the relative pose and update `mapPointSet` and `vSetKeyFrames`.

```
% Initialize the loop closure database
if currKeyFrameId == 3
    % Load the bag of features data created offline
    bofData      = load('bagOfFeaturesData.mat');
    loopDatabase = invertedImageIndex(bofData.bof);
    loopCandidates = [1; 2];

% Check loop closure after some key frames have been created
elseif currKeyFrameId > 20

    % Minimum number of feature matches of loop edges
    loopEdgeNumMatches = 50;

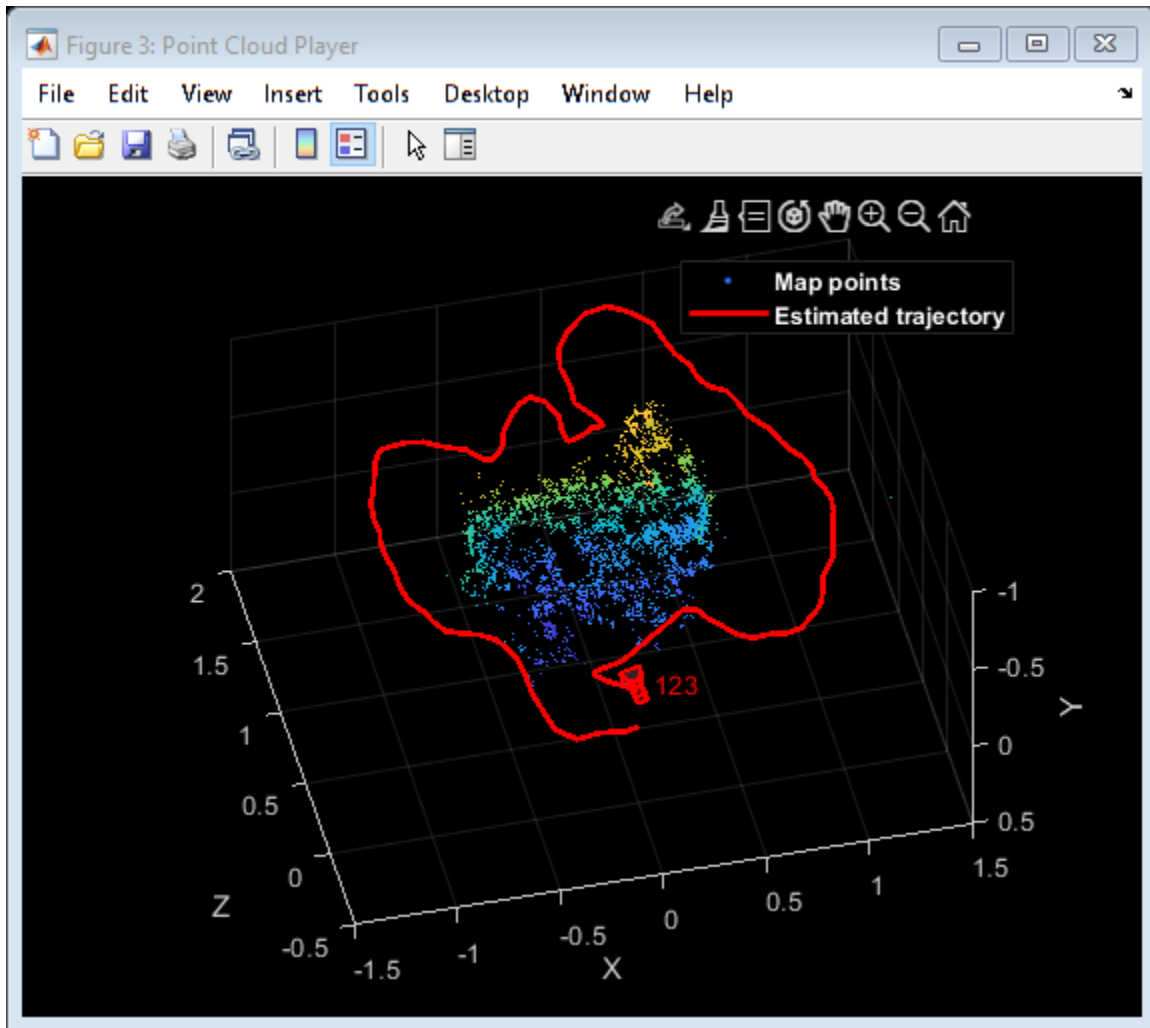
    % Detect possible loop closure key frame candidates
    [isDetected, validLoopCandidates] = helperCheckLoopClosure(vSetKeyFrames, currKeyFrameId,
        loopDatabase, currI, loopCandidates, loopEdgeNumMatches);

    if isDetected
        % Add loop closure connections
        [isLoopClosed, mapPointSet, vSetKeyFrames] = helperAddLoopConnections(...
            mapPointSet, vSetKeyFrames, validLoopCandidates, currKeyFrameId, ...
            currFeatures, currPoints, intrinsics, scaleFactor, loopEdgeNumMatches);
    end
end

% If no loop closure is detected, add the image into the database
if ~isLoopClosed
    currds = imageDatastore(imds.Files{currFrameIdx});
    addImages(loopDatabase, currds, 'Verbose', false);
    loopCandidates = [loopCandidates; currKeyFrameId]; %#ok<AGROW>
end

% Update IDs and indices
lastKeyFrameId = currKeyFrameId;
lastKeyFrameIdx = currFrameIdx;
addedFramesIdx = [addedFramesIdx; currFrameIdx]; %#ok<AGROW>
currFrameIdx = currFrameIdx + 1;
end % End of main loop
```





Loop edge added between keyframe: 1 and 123

Finally, a pose graph optimization is performed over the essential graph in `vSetKeyFrames` to correct the drift in rotation and translation. The essential graph is created internally by removing connections with fewer than `minNumMatches` matches in the covisibility graph. After pose graph optimization, update the 3-D locations of the map points using the optimized poses.

```
% Optimize the poses
```

```
vSetKeyFramesOptim = optimizePoses(vSetKeyFrames, minNumMatches, 'Tolerance', 1e-16, 'Verbose', 1)
```

```
Iteration 1, residual error 0.047601
```

```
Iteration 2, residual error 0.046330
```

```
Iteration 3, residual error 0.046329
```

```
Iteration 4, residual error 0.046329
```

```
Iteration 5, residual error 0.046329
```

```
Iteration 6, residual error 0.046329
```

```
Iteration 7, residual error 0.046329
```

```
Iteration 8, residual error 0.046329
```

```
Iteration 9, residual error 0.046329
```

```
Iteration 10, residual error 0.046329
```

```
Solver stopped because change in function value was less than specified function tolerance.
```



```
% Update map points after optimizing the poses
mapPointSet = helperUpdateGlobalMap(mapPointSet, directionAndDepth, ...
    vSetKeyFrames, vSetKeyFramesOptim);

updatePlot(mapPlot, vSetKeyFrames, mapPointSet);

% Plot the optimized camera trajectory
optimizedPoses = poses(vSetKeyFramesOptim);
plotOptimizedTrajectory(mapPlot, optimizedPoses)

% Update legend
showLegend(mapPlot);
```

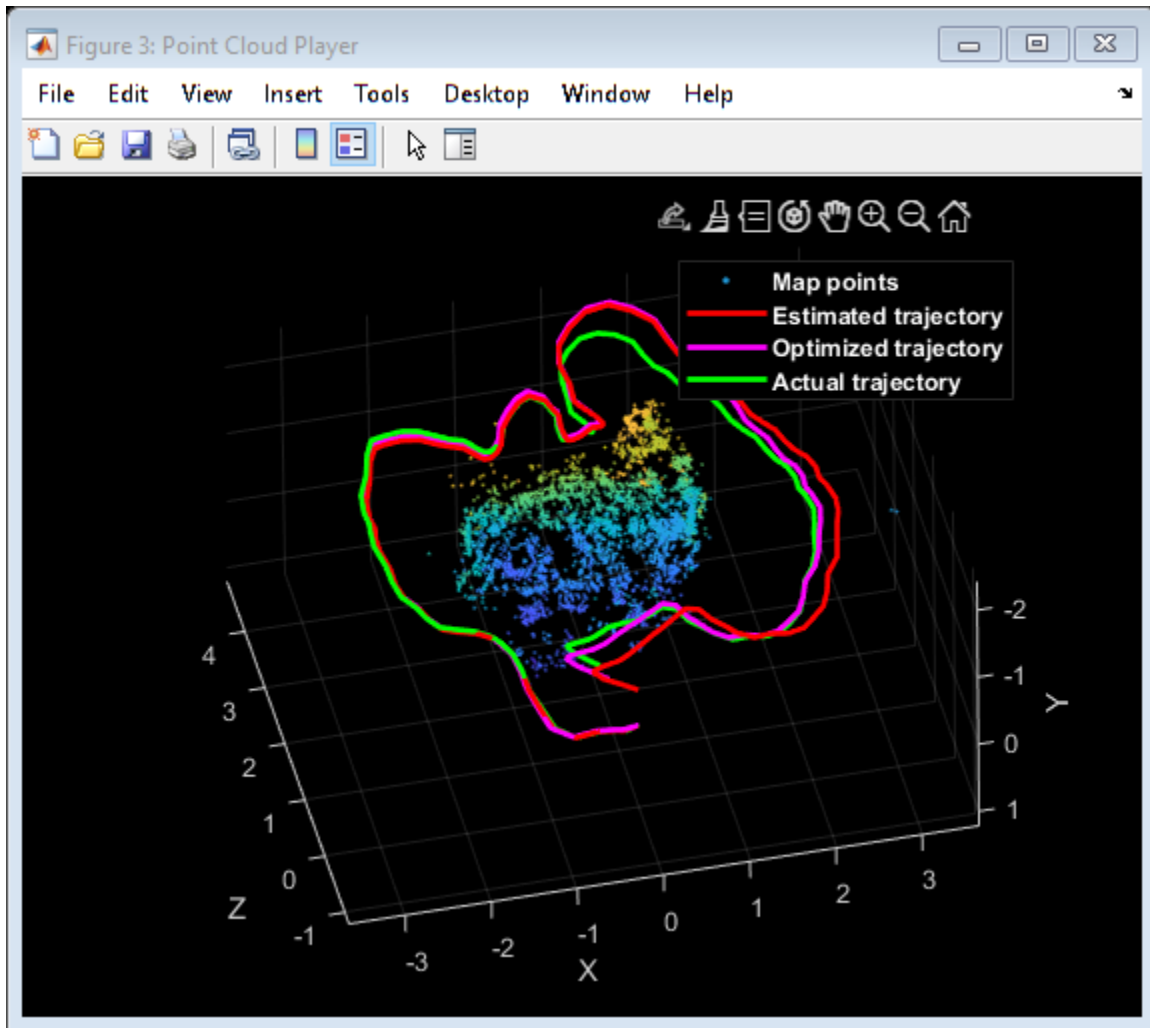
Compare with the Ground Truth

You can compare the optimized camera trajectory with the ground truth to evaluate the accuracy of ORB-SLAM. The downloaded data contains a `groundtruth.txt` file that stores the ground truth of camera pose of each frame. The data has been saved in the form of a MAT-file. You can also calculate the root-mean-square-error (RMSE) of trajectory estimates.

```
% Load ground truth
gTruthData = load('orb_slamGroundTruth.mat');
gTruth      = gTruthData.gTruth;

% Plot the actual camera trajectory
plotActualTrajectory(mapPlot, gTruth(addedFramesIdx), optimizedPoses);

% Show legend
showLegend(mapPlot);
```



```
% Evaluate tracking accuracy
helperEstimateTrajectoryError(gTruth(addedFramesIdx), optimizedPoses);
```

```
Absolute RMSE for key frame trajectory (m): 0.17784
```

This concludes an overview of how to build a map of an indoor environment and estimate the trajectory of the camera using ORB-SLAM.

Supporting Functions

Short helper functions are included below. Larger function are included in separate files.

helperAddLoopConnections add connections between the current keyframe and the valid loop candidate.

helperAddNewKeyFrame add key frames to the key frame set.

helperCheckLoopClosure detect loop candidates key frames by retrieving visually similar images from the database.

helperCreateNewMapPoints create new map points by triangulation.

helperFindProjectedPointsInImage check if projected world points are within an image.

helperHammingDistance compute hamming distance between two groups of binary feature vectors.

helperLocalBundleAdjustment refine the pose of the current key frame and the map of the surrounding scene.

helperMatchFeaturesInRadius match features within a radius.

helperSelectStrongConnections select strong connections with more than a specified number of matches.

helperSURFFeatureExtractorFunction implements the SURF feature extraction used in `bagOfFeatures`.

helperTrackLastKeyFrame estimate the current camera pose by tracking the last key frame.

helperTrackLocalMap refine the current camera pose by tracking the local map.

helperViewDirectionAndDepth store the mean view direction and the predicted depth of map points

helperVisualizeMatchedFeatures show the matched features in a frame.

helperVisualizeMotionAndStructure show map points and camera trajectory.

helperDetectAndExtractFeatures detect and extract and ORB features from the image.

```
function [features, validPoints] = helperDetectAndExtractFeatures(Irgb, ...
    scaleFactor, numLevels, varargin)

numPoints    = 1000;

% In this example, the images are already undistorted. In a general
% workflow, uncomment the following code to undistort the images.
%
% if nargin > 3
%     intrinsics = varargin{1};
% end
% Irgb = undistortImage(Irgb, intrinsics);

% Detect ORB features
Igray = rgb2gray(Irgb);

points = detectORBFeatures(Igray, 'ScaleFactor', scaleFactor, 'NumLevels', numLevels);

% Select a subset of features, uniformly distributed throughout the image
points = selectUniform(points, numPoints, size(Igray, 1:2));

% Extract features
[features, validPoints] = extractFeatures(Igray, points);
end
```

helperHomographyScore compute homography and evaluate reconstruction.

```

function [H, score, inliersIndex] = helperComputeHomography(matchedPoints1, matchedPoints2)

[H, inliersLogicalIndex] = estimateGeometricTransform2D( ...
    matchedPoints1, matchedPoints2, 'projective', ...
    'MaxNumTrials', 1e3, 'MaxDistance', 4, 'Confidence', 90);

inlierPoints1 = matchedPoints1(inliersLogicalIndex);
inlierPoints2 = matchedPoints2(inliersLogicalIndex);

inliersIndex = find(inliersLogicalIndex);

locations1 = inlierPoints1.Location;
locations2 = inlierPoints2.Location;
xy1In2 = transformPointsForward(H, locations1);
xy2In1 = transformPointsInverse(H, locations2);
errorIn2 = sum((locations2 - xy1In2).^2, 2);
error2In1 = sum((locations1 - xy2In1).^2, 2);

outlierThreshold = 6;

score = sum(max(outlierThreshold-errorIn2, 0)) + ...
    sum(max(outlierThreshold-error2In1, 0));
end

```

helperFundamentalMatrixScore compute fundamental matrix and evaluate reconstruction.

```

function [F, score, inliersIndex] = helperComputeFundamentalMatrix(matchedPoints1, matchedPoints2)

[F, inliersLogicalIndex] = estimateFundamentalMatrix( ...
    matchedPoints1, matchedPoints2, 'Method', 'RANSAC', ...
    'NumTrials', 1e3, 'DistanceThreshold', 0.01);

inlierPoints1 = matchedPoints1(inliersLogicalIndex);
inlierPoints2 = matchedPoints2(inliersLogicalIndex);

inliersIndex = find(inliersLogicalIndex);

locations1 = inlierPoints1.Location;
locations2 = inlierPoints2.Location;

% Distance from points to epipolar line
lineIn1 = epipolarLine(F, locations2);
error2In1 = (sum([locations1, ones(size(locations1, 1),1)].* lineIn1, 2)).^2 ...
    ./ sum(lineIn1(:,1:2).^2, 2);
lineIn2 = epipolarLine(F, locations1);
errorIn2 = (sum([locations2, ones(size(locations2, 1),1)].* lineIn2, 2)).^2 ...
    ./ sum(lineIn2(:,1:2).^2, 2);

outlierThreshold = 4;

score = sum(max(outlierThreshold-errorIn2, 0)) + ...
    sum(max(outlierThreshold-error2In1, 0));
end

```

helperTriangulateTwoFrames triangulate two frames to initialize the map.

```

function [isValid, xyzPoints, inlierIdx] = helperTriangulateTwoFrames(...
    pose1, pose2, matchedPoints1, matchedPoints2, intrinsics, minParallax)

[R1, t1] = cameraPoseToExtrinsics(pose1.Rotation, pose1.Translation);
camMatrix1 = cameraMatrix(intrinsics, R1, t1);

[R2, t2] = cameraPoseToExtrinsics(pose2.Rotation, pose2.Translation);
camMatrix2 = cameraMatrix(intrinsics, R2, t2);

[xyzPoints, reprojectionErrors, isInFront] = triangulate(matchedPoints1, ...
    matchedPoints2, camMatrix1, camMatrix2);

% Filter points by view direction and reprojection error
minReprojError = 1;
inlierIdx = isInFront & reprojectionErrors < minReprojError;
xyzPoints = xyzPoints(inlierIdx, :);

% A good two-view with significant parallax
ray1 = xyzPoints - pose1.Translation;
ray2 = xyzPoints - pose2.Translation;
cosAngle = sum(ray1 .* ray2, 2) ./ (vecnorm(ray1, 2, 2) .* vecnorm(ray2, 2, 2));

% Check parallax
isValid = all(cosAngle < cosd(minParallax) & cosAngle > 0);
end

```

helperIsKeyFrame check if a frame is a key frame.

```

function isKeyFrame = helperIsKeyFrame(mapPoints, ...
    refKeyFrameId, lastKeyFrameIndex, currFrameIndex, mapPointsIndices)

numPointsRefKeyFrame = numel(findWorldPointsInView(mapPoints, refKeyFrameId));

% More than 20 frames have passed from last key frame insertion
tooManyNonKeyFrames = currFrameIndex >= lastKeyFrameIndex + 20;

% Track less than 90 map points
tooFewMapPoints = numel(mapPointsIndices) < 90;

% Tracked map points are fewer than 90% of points tracked by
% the reference key frame
tooFewTrackedPoints = numel(mapPointsIndices) < 0.9 * numPointsRefKeyFrame;

isKeyFrame = (tooManyNonKeyFrames || tooFewMapPoints) && tooFewTrackedPoints;
end

```

helperCullRecentMapPoints cull recently added map points.

```

function [mapPointSet, directionAndDepth, mapPointsIdx] = helperCullRecentMapPoints(mapPointSet,
    outlierIdx = setdiff(newPointIdx, mapPointsIdx);
if ~isempty(outlierIdx)
    mapPointSet = removeWorldPoints(mapPointSet, outlierIdx);
    directionAndDepth = remove(directionAndDepth, outlierIdx);
    mapPointsIdx = mapPointsIdx - arrayfun(@(x) nnz(x > outlierIdx), mapPointsIdx);
end
end

```

helperEstimateTrajectoryError calculate the tracking error.

```
function rmse = helperEstimateTrajectoryError(gTruth, cameraPoses)
locations      = vertcat(cameraPoses.AbsolutePose.Translation);
gLocations     = vertcat(gTruth.Translation);
scale         = median(vecnorm(gLocations, 2, 2))/ median(vecnorm(locations, 2, 2));
scaledLocations = locations * scale;

rmse = sqrt(mean( sum((scaledLocations - gLocations).^2, 2) ));
disp(['Absolute RMSE for key frame trajectory (m): ', num2str(rmse)]);
end
```

helperUpdateGlobalMap update 3-D locations of map points after pose graph optimization

```
function [mapPointSet, directionAndDepth] = helperUpdateGlobalMap(...
    mapPointSet, directionAndDepth, vSetKeyFrames, vSetKeyFramesOptim)
%helperUpdateGlobalMap update map points after pose graph optimization
posesOld      = vSetKeyFrames.Views.AbsolutePose;
posesNew      = vSetKeyFramesOptim.Views.AbsolutePose;
positionsOld  = mapPointSet.WorldPoints;
positionsNew  = positionsOld;
indices       = 1:mapPointSet.Count;

% Update world location of each map point based on the new absolute pose of
% the corresponding major view
for i = 1: mapPointSet.Count
    majorViewIds = directionAndDepth.MajorViewId(i);
    tform = posesOld(majorViewIds).T \ posesNew(majorViewIds).T ;
    positionsNew(i, :) = positionsOld(i, :) * tform(1:3,1:3) + tform(4, 1:3);
end
mapPointSet = updateWorldPoints(mapPointSet, indices, positionsNew);
end
```

Reference

- [1] Mur-Artal, Raul, Jose Maria Martinez Montiel, and Juan D. Tardos. "ORB-SLAM: a versatile and accurate monocular SLAM system." *IEEE Transactions on Robotics* 31, no. 5, pp 1147-116, 2015.
- [2] Sturm, Jürgen, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. "A benchmark for the evaluation of RGB-D SLAM systems". In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 573-580, 2012.

Structure From Motion From Two Views

Structure from motion (SfM) is the process of estimating the 3-D structure of a scene from a set of 2-D images. This example shows you how to estimate the poses of a calibrated camera from two images, reconstruct the 3-D structure of the scene up to an unknown scale factor, and then recover the actual scale factor by detecting an object of a known size.

Overview

This example shows how to reconstruct a 3-D scene from a pair 2-D images taken with a camera calibrated using the Camera Calibrator app. The algorithm consists of the following steps:

- 1 Match a sparse set of points between the two images. There are multiple ways of finding point correspondences between two images. This example detects corners in the first image using the `detectMinEigenFeatures` function, and tracks them into the second image using `vision.PointTracker`. Alternatively you can use `extractFeatures` followed by `matchFeatures`.
- 2 Estimate the fundamental matrix using `estimateFundamentalMatrix`.
- 3 Compute the motion of the camera using the `cameraPose` function.
- 4 Match a dense set of points between the two images. Re-detect the point using `detectMinEigenFeatures` with a reduced 'MinQuality' to get more points. Then track the dense points into the second image using `vision.PointTracker`.
- 5 Determine the 3-D locations of the matched points using `triangulate`.
- 6 Detect an object of a known size. In this scene there is a globe, whose radius is known to be 10cm. Use `pcfitsphere` to find the globe in the point cloud.
- 7 Recover the actual scale, resulting in a metric reconstruction.

Read a Pair of Images

Load a pair of images into the workspace.

```
imageDir = fullfile(toolboxdir('vision'), 'visiondata', 'upToScaleReconstructionImages');
images = imageDatastore(imageDir);
I1 = readimage(images, 1);
I2 = readimage(images, 2);
figure
imshowpair(I1, I2, 'montage');
title('Original Images');
```

Original Images



Load Camera Parameters

This example uses the camera parameters calculated by the cameraCalibrator app. The parameters are stored in the cameraParams object, and include the camera intrinsics and lens distortion coefficients.

```
% Load precomputed camera parameters  
load upToScaleReconstructionCameraParameters.mat
```

Remove Lens Distortion

Lens distortion can affect the accuracy of the final reconstruction. You can remove the distortion from each of the images using the undistortImage function. This process straightens the lines that are bent by the radial distortion of the lens.

```
I1 = undistortImage(I1, cameraParams);  
I2 = undistortImage(I2, cameraParams);  
figure  
imshowpair(I1, I2, 'montage');  
title('Undistorted Images');
```




Find Point Correspondences Between The Images

Detect good features to track. Reduce 'MinQuality' to detect fewer points, which would be more uniformly distributed throughout the image. If the motion of the camera is not very large, then tracking using the KLT algorithm is a good way to establish point correspondences.

```
% Detect feature points
```

```
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1), 'MinQuality', 0.1);
```

```
% Visualize detected points
```

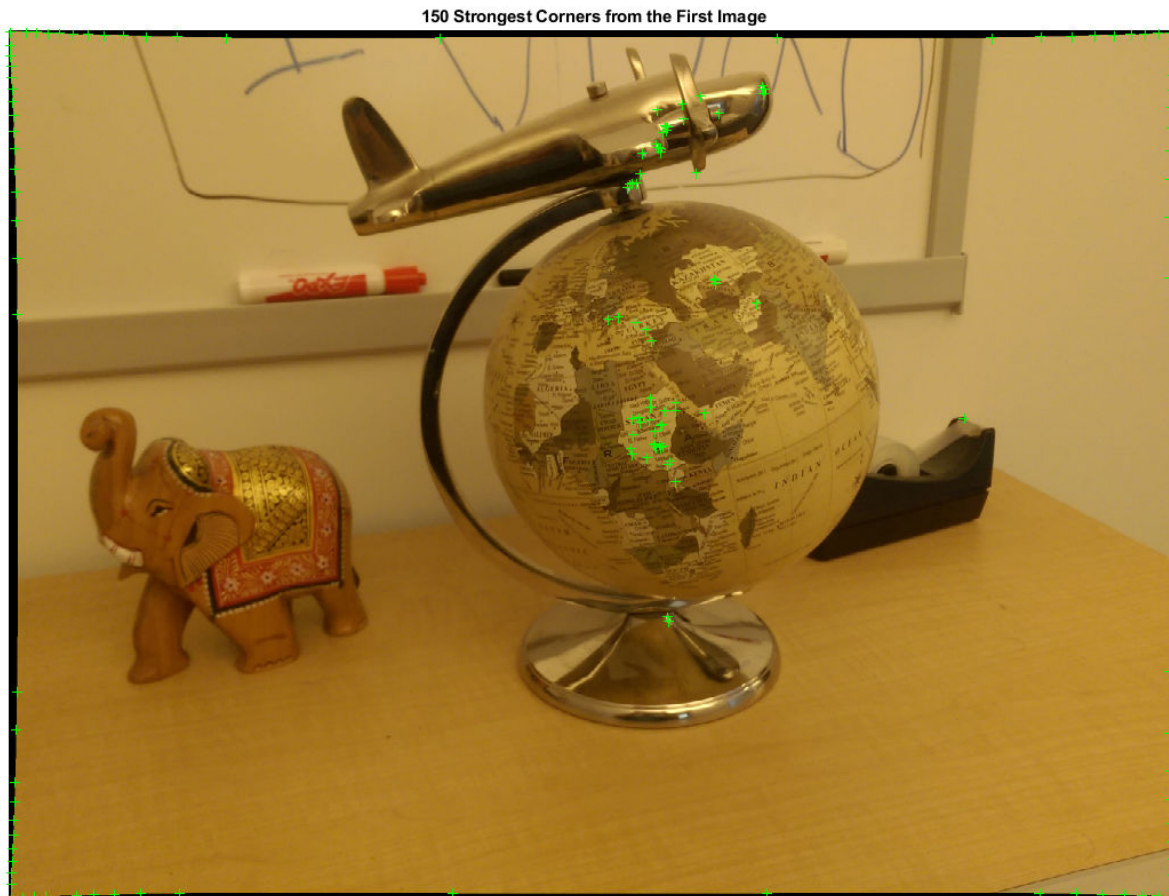
```
figure
```

```
imshow(I1, 'InitialMagnification', 50);
```

```
title('150 Strongest Corners from the First Image');
```

```
hold on
```

```
plot(selectStrongest(imagePoints1, 150));
```

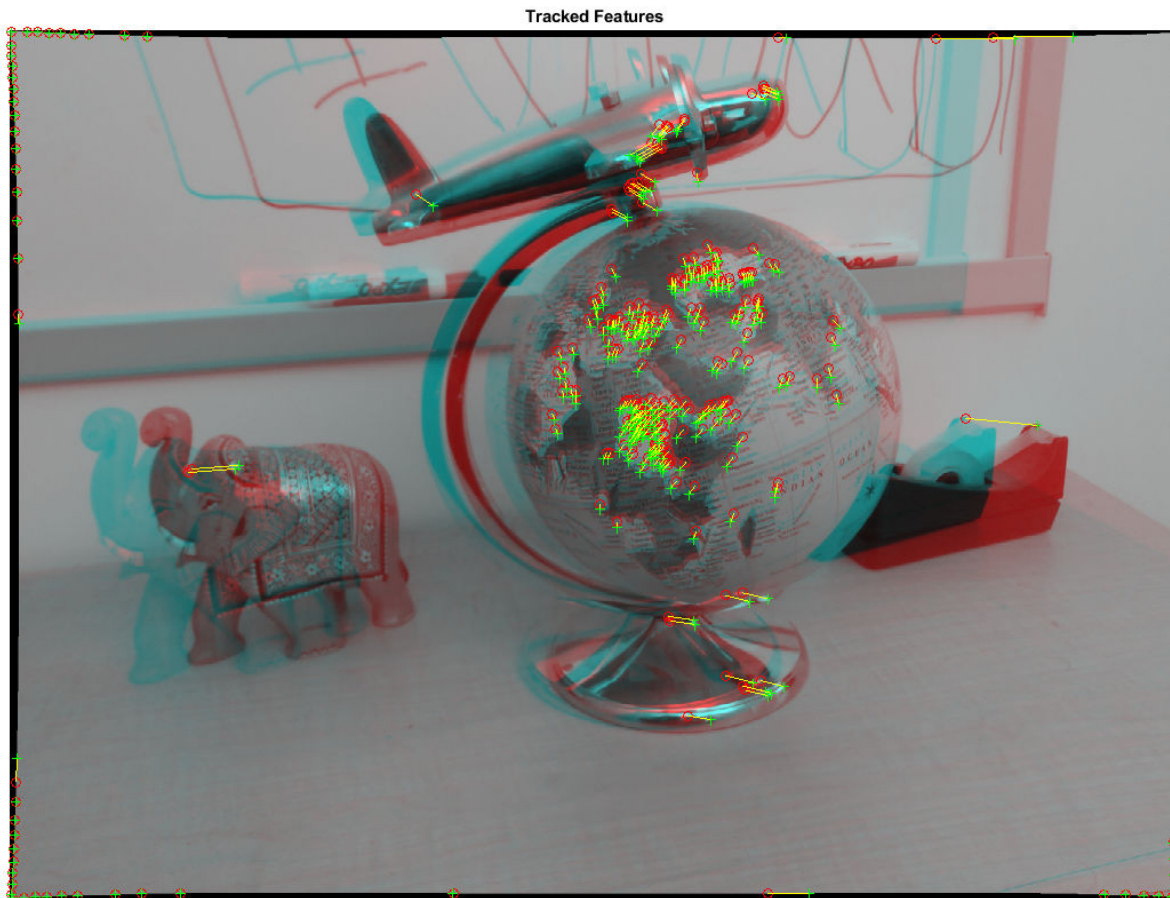


```
% Create the point tracker
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels', 5);

% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);

% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
matchedPoints1 = imagePoints1(validIdx, :);
matchedPoints2 = imagePoints2(validIdx, :);

% Visualize correspondences
figure
showMatchedFeatures(I1, I2, matchedPoints1, matchedPoints2);
title('Tracked Features');
```



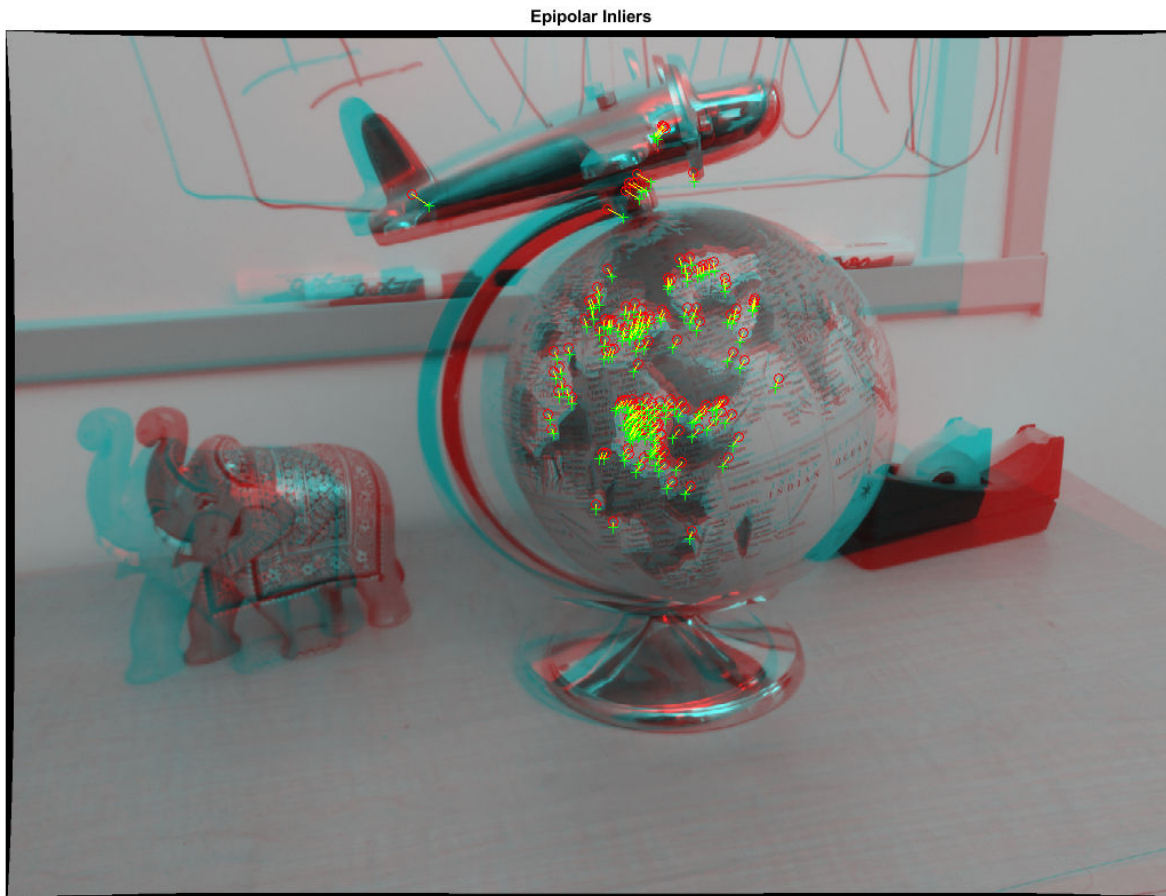
Estimate the Fundamental Matrix

Use the `estimateFundamentalMatrix` function to compute the fundamental matrix and find the inlier points that meet the epipolar constraint.

```
% Estimate the fundamental matrix
[fMatrix, epipolarInliers] = estimateFundamentalMatrix(...
    matchedPoints1, matchedPoints2, 'Method', 'MSAC', 'NumTrials', 10000);

% Find epipolar inliers
inlierPoints1 = matchedPoints1(epipolarInliers, :);
inlierPoints2 = matchedPoints2(epipolarInliers, :);

% Display inlier matches
figure
showMatchedFeatures(I1, I2, inlierPoints1, inlierPoints2);
title('Epipolar Inliers');
```



Compute the Camera Pose

Compute the rotation and translation between the camera poses corresponding to the two images. Note that \mathbf{t} is a unit vector, because translation can only be computed up to scale.

```
[R, t] = cameraPose(fMatrix, cameraParams, inlierPoints1, inlierPoints2);
```

Reconstruct the 3-D Locations of Matched Points

Re-detect points in the first image using lower 'MinQuality' to get more points. Track the new points into the second image. Estimate the 3-D locations corresponding to the matched points using the `triangulate` function, which implements the Direct Linear Transformation (DLT) algorithm [1]. Place the origin at the optical center of the camera corresponding to the first image.

```
% Detect dense feature points
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1), 'MinQuality', 0.001);

% Create the point tracker
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels', 5);

% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);
```

```

% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
matchedPoints1 = imagePoints1(validIdx, :);
matchedPoints2 = imagePoints2(validIdx, :);

% Compute the camera matrices for each position of the camera
% The first camera is at the origin looking along the X-axis. Thus, its
% rotation matrix is identity, and its translation vector is 0.
camMatrix1 = cameraMatrix(cameraParams, eye(3), [0 0 0]);
camMatrix2 = cameraMatrix(cameraParams, R', -t*R');

% Compute the 3-D points
points3D = triangulate(matchedPoints1, matchedPoints2, camMatrix1, camMatrix2);

% Get the color of each reconstructed point
numPixels = size(I1, 1) * size(I1, 2);
allColors = reshape(I1, [numPixels, 3]);
colorIdx = sub2ind([size(I1, 1), size(I1, 2)], round(matchedPoints1(:,2)), ...
    round(matchedPoints1(:, 1)));
color = allColors(colorIdx, :);

% Create the point cloud
ptCloud = pointCloud(points3D, 'Color', color);

```

Display the 3-D Point Cloud

Use the `plotCamera` function to visualize the locations and orientations of the camera, and the `pcshow` function to visualize the point cloud.

```

% Visualize the camera locations and orientations
cameraSize = 0.3;
figure
plotCamera('Size', cameraSize, 'Color', 'r', 'Label', '1', 'Opacity', 0);
hold on
grid on
plotCamera('Location', t, 'Orientation', R, 'Size', cameraSize, ...
    'Color', 'b', 'Label', '2', 'Opacity', 0);

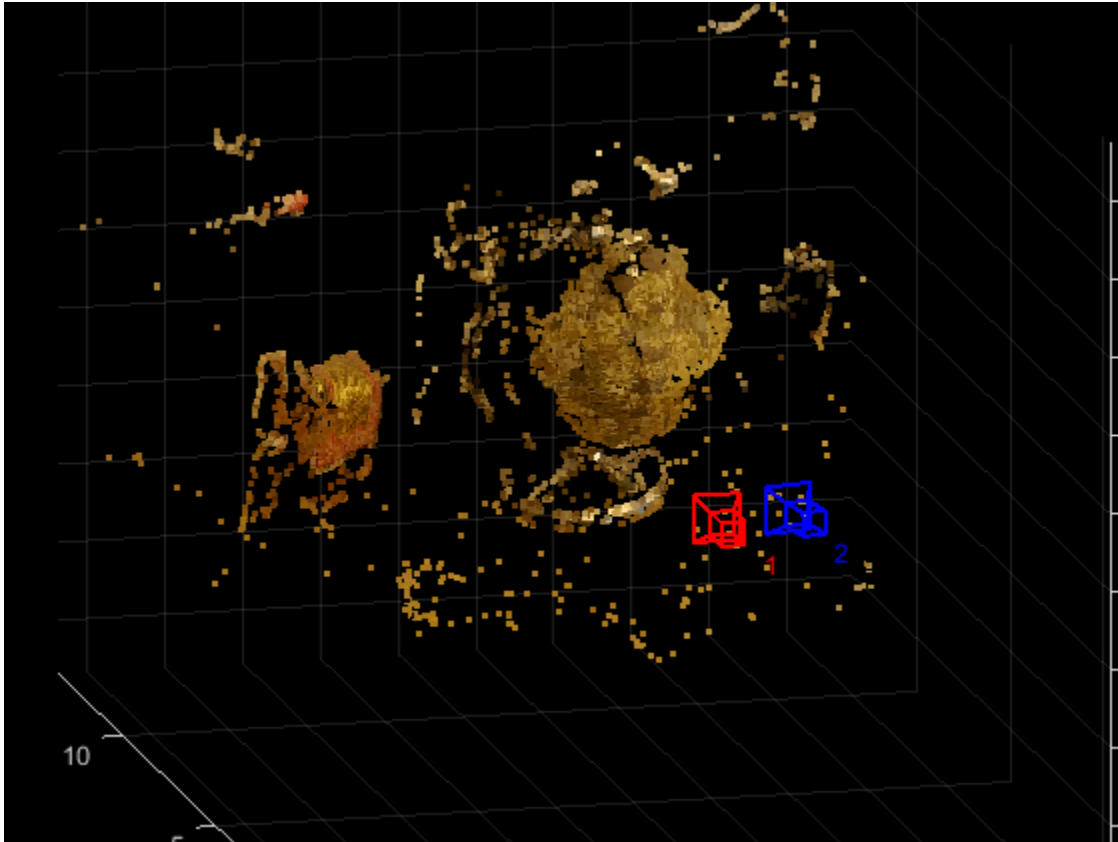
% Visualize the point cloud
pcshow(ptCloud, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
    'MarkerSize', 45);

% Rotate and zoom the plot
camorbit(0, -30);
camzoom(1.5);

% Label the axes
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis')

title('Up to Scale Reconstruction of the Scene');

```

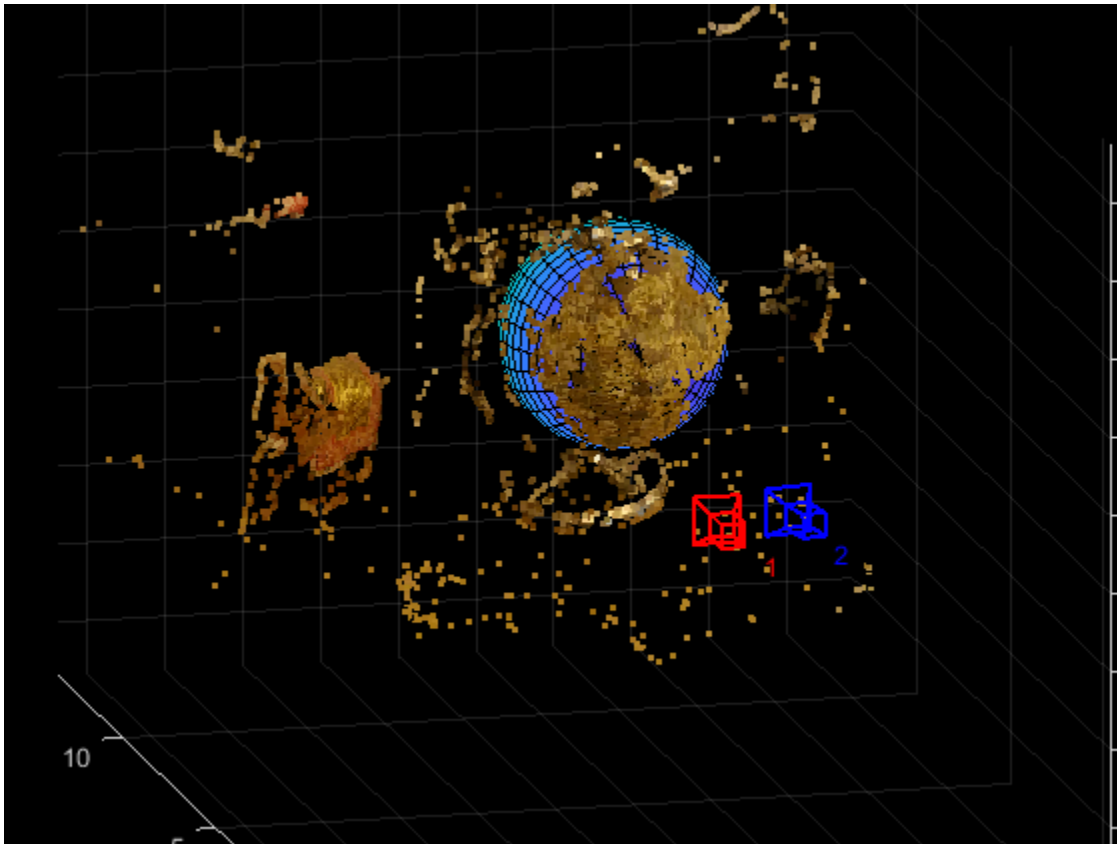


Fit a Sphere to the Point Cloud to Find the Globe

Find the globe in the point cloud by fitting a sphere to the 3-D points using the `pcfitsphere` function.

```
% Detect the globe
globe = pcfitsphere(ptCloud, 0.1);

% Display the surface of the globe
plot(globe);
title('Estimated Location and Size of the Globe');
hold off
```



Metric Reconstruction of the Scene

The actual radius of the globe is 10cm. You can now determine the coordinates of the 3-D points in centimeters.

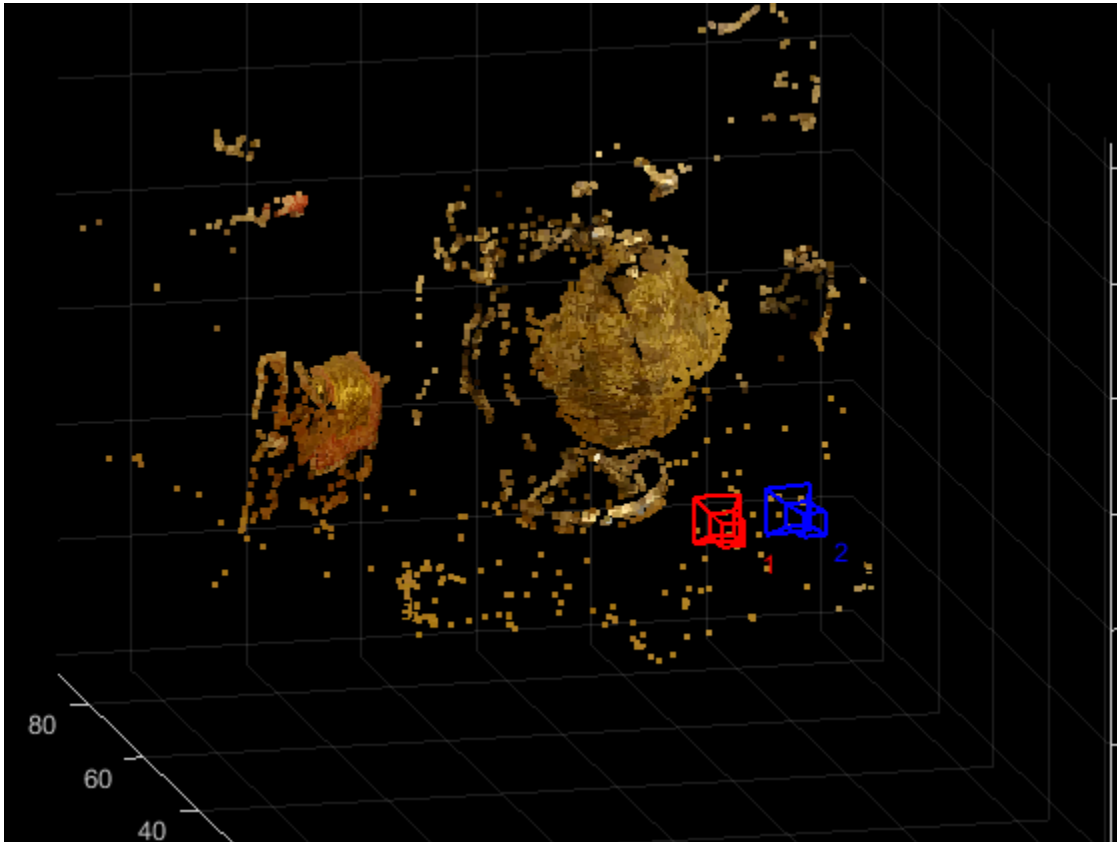
```
% Determine the scale factor
scaleFactor = 10 / globe.Radius;

% Scale the point cloud
ptCloud = pointCloud(points3D * scaleFactor, 'Color', color);
t = t * scaleFactor;

% Visualize the point cloud in centimeters
cameraSize = 2;
figure
plotCamera('Size', cameraSize, 'Color', 'r', 'Label', '1', 'Opacity', 0);
hold on
grid on
plotCamera('Location', t, 'Orientation', R, 'Size', cameraSize, ...
          'Color', 'b', 'Label', '2', 'Opacity', 0);

% Visualize the point cloud
pcshow(ptCloud, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
       'MarkerSize', 45);
camorbit(0, -30);
camzoom(1.5);
```

```
% Label the axes
xlabel('x-axis (cm)');
ylabel('y-axis (cm)');
zlabel('z-axis (cm)')
title('Metric Reconstruction of the Scene');
```



Summary

This example showed how to recover camera motion and reconstruct the 3-D structure of a scene from two images taken with a calibrated camera.

References

[1] Hartley, Richard, and Andrew Zisserman. Multiple View Geometry in Computer Vision. Second Edition. Cambridge, 2000.

Evaluating the Accuracy of Single Camera Calibration

This example shows how to evaluate the accuracy of camera parameters estimated using the `cameraCalibrator` app or the `estimateCameraParameters` function.

Overview

Camera calibration is the process of estimating parameters of the camera using images of a special calibration pattern. The parameters include camera intrinsics, distortion coefficients, and camera extrinsics. Once you calibrate a camera, there are several ways to evaluate the accuracy of the estimated parameters:

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

Calibrate the Camera

Estimate camera parameters using a set of images of a checkerboard calibration pattern.

```
% Create a set of calibration images.
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', ...
    'calibration', 'mono'));
imageFileNames = images.Files;

% Detect calibration pattern.
[imagePoints, boardSize] = detectCheckerboardPoints(imageFileNames);

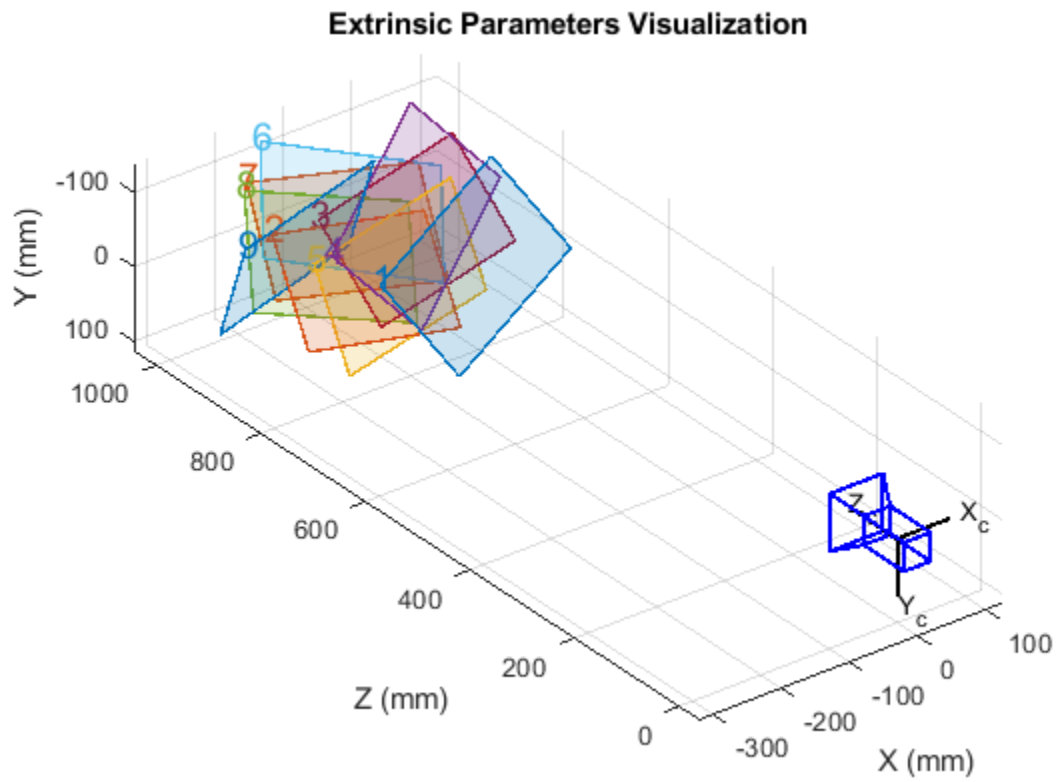
% Generate world coordinates of the corners of the squares.
squareSize = 29; % millimeters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Calibrate the camera.
I = readimage(images, 1);
imageSize = [size(I, 1), size(I, 2)];
[params, ~, estimationErrors] = estimateCameraParameters(imagePoints, worldPoints, ...
    'ImageSize', imageSize);
```

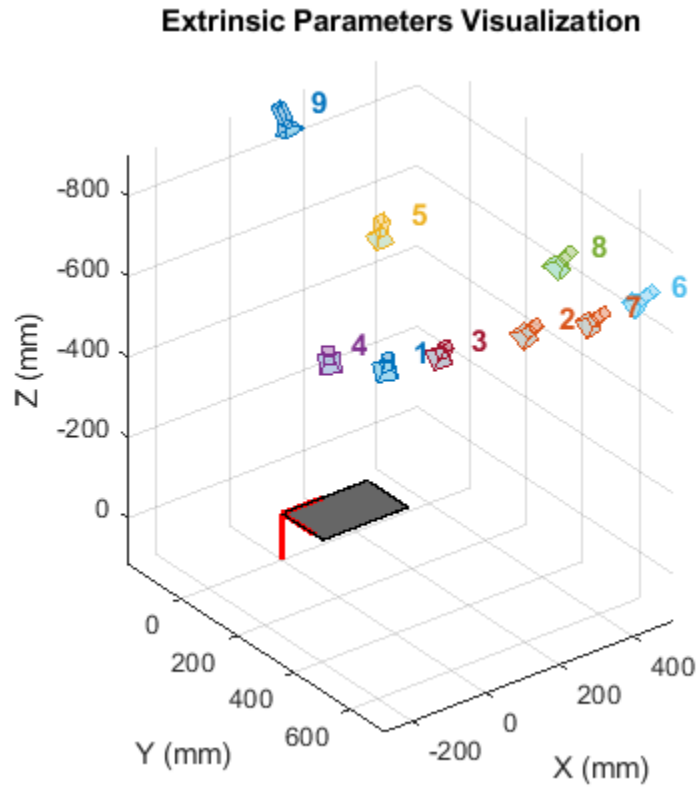
Extrinsics

You can quickly discover obvious errors in your calibration by plotting relative locations of the camera and the calibration pattern. Use the `showExtrinsics` function to either plot the locations of the calibration pattern in the camera's coordinate system, or the locations of the camera in the pattern's coordinate system. Look for obvious problems, such as the pattern being behind the camera, or the camera being behind the pattern. Also check if a pattern is too far or too close to the camera.

```
figure;
showExtrinsics(params, 'CameraCentric');
```



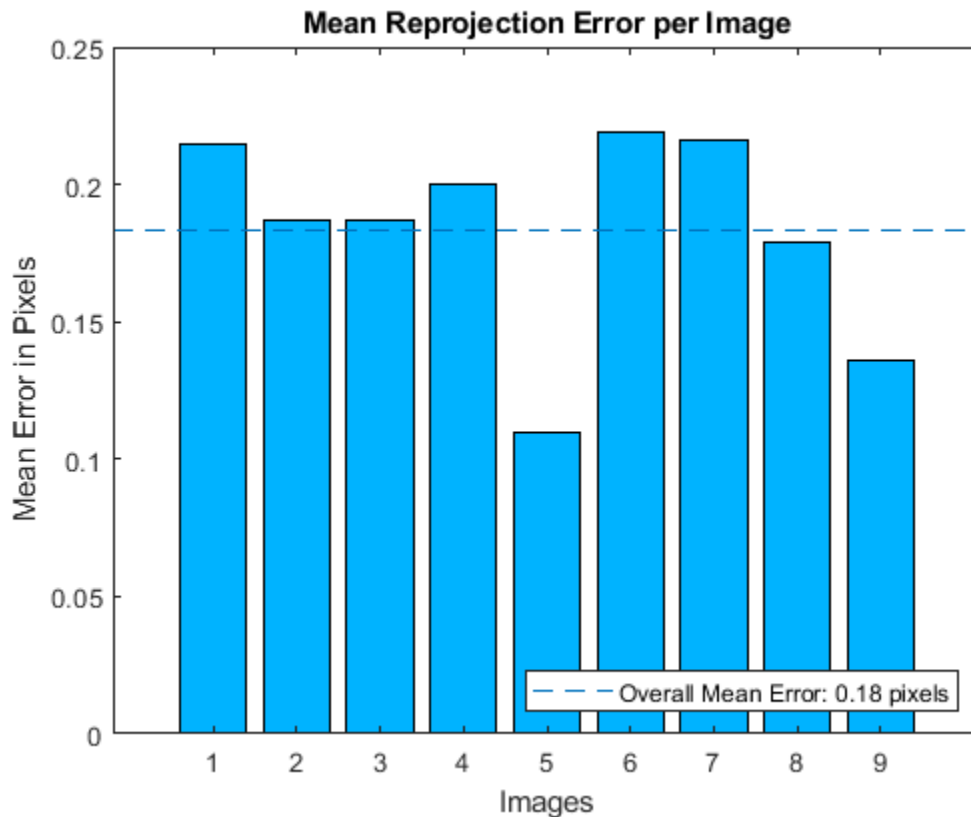
```
figure;  
showExtrinsics(params, 'PatternCentric');
```



Reprojection Errors

Reprojection errors provide a qualitative measure of accuracy. A reprojection error is the distance between a pattern keypoint detected in a calibration image, and a corresponding world point projected into the same image. The `showReprojectionErrors` function provides a useful visualization of the average reprojection error in each calibration image. If the overall mean reprojection error is too high, consider excluding the images with the highest error and recalibrating.

```
figure;  
showReprojectionErrors(params);
```



Estimation Errors

Estimation errors represent the uncertainty of each estimated parameter. The `estimateCameraParameters` function optionally returns `estimationErrors` output, containing the standard error corresponding to each estimated camera parameter. The returned standard error σ (in the same units as the corresponding parameter) can be used to calculate confidence intervals. For example $\pm 1.96\sigma$ corresponds to the 95% confidence interval. In other words, the probability that the actual value of a given parameter is within 1.96σ of its estimate is 95%.

```
displayErrors(estimationErrors, params);
```

```
Standard Errors of Estimated Camera Parameters
```

```
Intrinsics
```

```
-----
```

```
Focal length (pixels): [ 714.1886 +/- 3.3219      710.3786 +/- 4.0579 ]
Principal point (pixels): [ 563.6480 +/- 5.3967      355.7251 +/- 3.3036 ]
Radial distortion: [ -0.3536 +/- 0.0091      0.1730 +/- 0.0488 ]
```

```
Extrinsics
```

```
-----
```

```
Rotation vectors:
```

```
[ -0.6096 +/- 0.0054      -0.1789 +/- 0.0073      -0.3835 +/- 0.0025 ]
[ -0.7283 +/- 0.0050      -0.0996 +/- 0.0072      0.1964 +/- 0.0025 ]
[ -0.6722 +/- 0.0051      -0.1444 +/- 0.0074      -0.1329 +/- 0.0025 ]
[ -0.5836 +/- 0.0056      -0.2901 +/- 0.0074      -0.5622 +/- 0.0025 ]
```

[-0.3157 +/- 0.0065	-0.1441 +/- 0.0075	-0.1067 +/- 0.001
[-0.7581 +/- 0.0052	0.1947 +/- 0.0072	0.4324 +/- 0.003
[-0.7515 +/- 0.0051	0.0767 +/- 0.0072	0.2070 +/- 0.002
[-0.6223 +/- 0.0053	0.0231 +/- 0.0073	0.3663 +/- 0.002
[0.3443 +/- 0.0063	-0.2226 +/- 0.0073	-0.0437 +/- 0.001

Translation vectors (mm):

[-146.0516 +/- 6.0391	-26.8684 +/- 3.7318	797.9027 +/- 3.900
[-209.4357 +/- 6.9637	-59.4563 +/- 4.3578	921.8198 +/- 4.629
[-129.3823 +/- 7.0907	-44.1028 +/- 4.3751	937.6832 +/- 4.491
[-151.0048 +/- 6.6905	-27.3251 +/- 4.1339	884.2789 +/- 4.392
[-174.9499 +/- 6.7056	-24.3498 +/- 4.1606	886.4961 +/- 4.668
[-134.3095 +/- 7.8887	-103.4979 +/- 4.8925	1042.4554 +/- 4.818
[-173.9845 +/- 7.6891	-73.1689 +/- 4.7812	1017.2386 +/- 4.812
[-202.9446 +/- 7.4327	-87.9089 +/- 4.6482	983.6958 +/- 4.907
[-319.8860 +/- 6.3213	-119.8897 +/- 4.0922	829.4582 +/- 4.959

How to Improve Calibration Accuracy

Whether or not a particular reprojection or estimation error is acceptable depends on the precision requirements of your particular application. However, if you have determined that your calibration accuracy is unacceptable, there are several ways to improve it:

- Modify calibration settings. Try using 3 radial distortion coefficients, estimating tangential distortion, or the skew.
- Take more calibration images. The pattern in the images must be in different 3D orientations, and it should be positioned such that you have keypoints in all parts of the field of view. In particular, it is very important to have keypoints close to the edges and the corners of the image in order to get a better estimate of the distortion coefficients.
- Exclude images that have high reprojection errors and re-calibrate.

Summary

This example showed how to obtain and interpret camera calibration errors.

References

[1] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330-1334, 2000.

Measuring Planar Objects with a Calibrated Camera

This example shows how to measure the diameter of coins in world units using a single calibrated camera.

Overview

This example shows how to calibrate a camera, and then use it to measure the size of planar objects, such as coins. An example application of this approach is measuring parts on a conveyor belt for quality control.

Calibrate the Camera

Camera calibration is the process of estimating the parameters of the lens and the image sensor. These parameters are needed to measure objects captured by the camera. This example shows how to calibrate a camera programmatically. Alternatively, you can calibrate a camera using the `cameraCalibrator` app.

To calibrate the camera, we first need to take multiple images of a calibration pattern from different angles. A typical calibration pattern is an asymmetric checkerboard, where one side contains an even number of squares, both black and white, and the other contains an odd number of squares.

The pattern must be affixed to a flat surface, and it should be at approximately the same distance from the camera as the objects you want to measure. The size of a square must be measured in world units, for example millimeters, as precisely as possible. In this example we use 9 images of the pattern, but in practice it is recommended to use 10 to 20 images for accurate calibration.

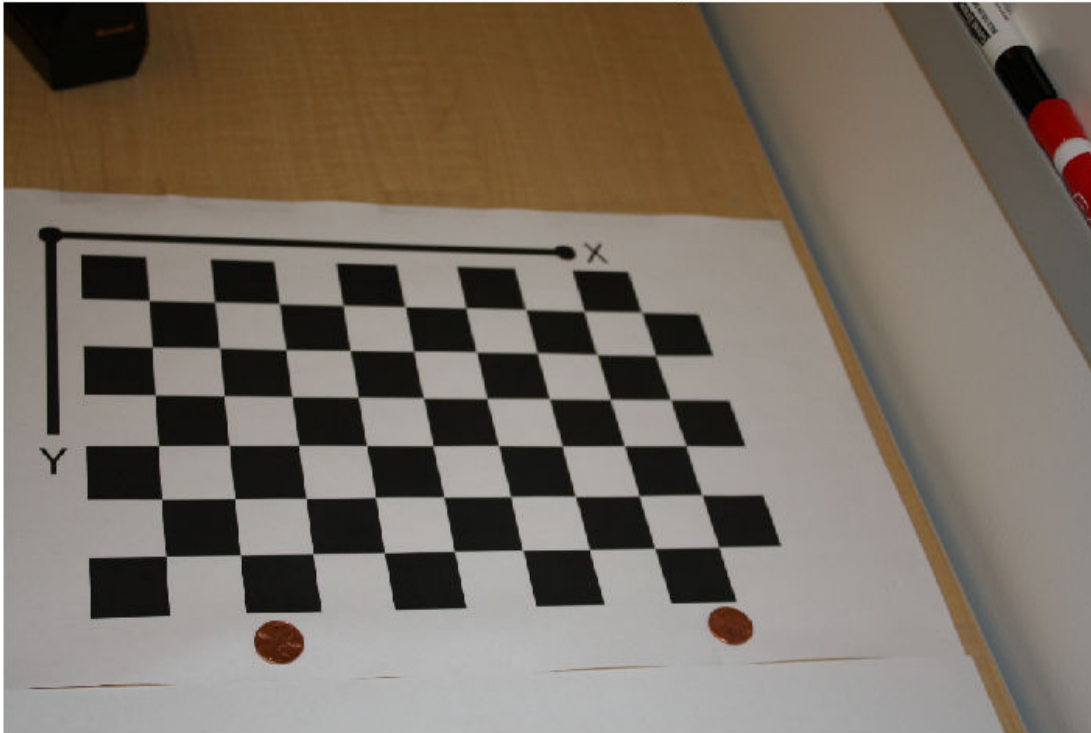
Prepare Calibration Images

Create a cell array of file names of calibration images.

```
numImages = 9;
files = cell(1, numImages);
for i = 1:numImages
    files{i} = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', ...
        'calibration', 'slr', sprintf('image%d.jpg', i));
end

% Display one of the calibration images
magnification = 25;
I = imread(files{1});
figure; imshow(I, 'InitialMagnification', magnification);
title('One of the Calibration Images');
```

One of the Calibration Images



Estimate Camera Parameters

```

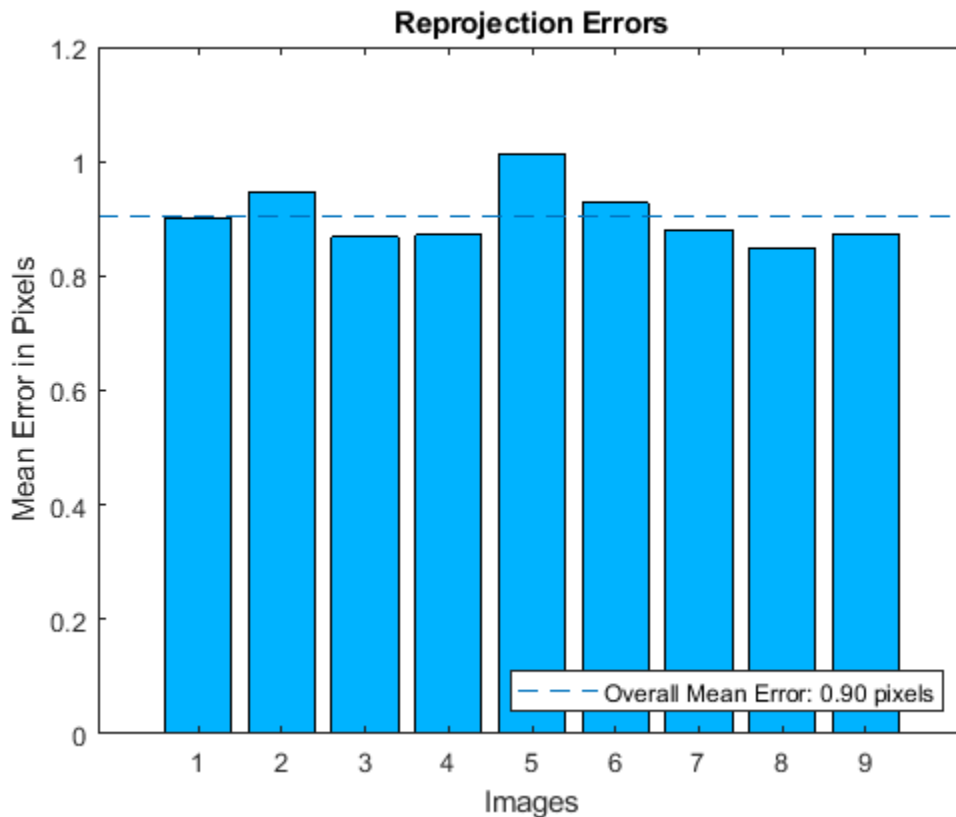
% Detect the checkerboard corners in the images.
[imagePoints, boardSize] = detectCheckerboardPoints(files);

% Generate the world coordinates of the checkerboard corners in the
% pattern-centric coordinate system, with the upper-left corner at (0,0).
squareSize = 29; % in millimeters
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Calibrate the camera.
imageSize = [size(I, 1), size(I, 2)];
cameraParams = estimateCameraParameters(imagePoints, worldPoints, ...
    'ImageSize', imageSize);

% Evaluate calibration accuracy.
figure; showReprojectionErrors(cameraParams);
title('Reprojection Errors');

```



The bar graph indicates the accuracy of the calibration. Each bar shows the mean reprojection error for the corresponding calibration image. The reprojection errors are the distances between the corner points detected in the image, and the corresponding ideal world points projected into the image.

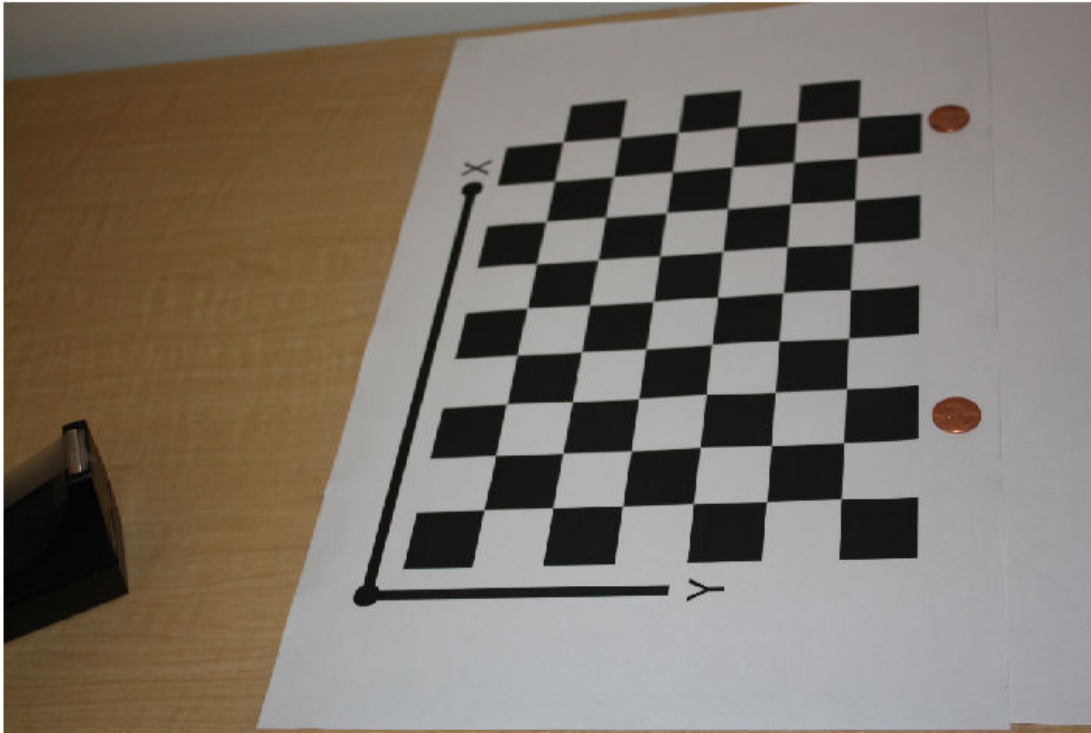
Read the Image of Objects to Be Measured

Load the image containing objects to be measured. This image includes the calibration pattern, and the pattern is in the same plane as the objects you want to measure. In this example, both the pattern and the coins are on the same table top.

Alternatively, you could use two separate images: one containing the pattern, and the other containing the objects to be measured. Again, the objects and the pattern must be in the same plane. Furthermore, images must be captured from exactly the same view point, meaning that the camera must be fixed in place.

```
imOrig = imread(fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', ...
    'calibration', 'slr', 'image9.jpg'));
figure; imshow(imOrig, 'InitialMagnification', magnification);
title('Input Image');
```


Input Image

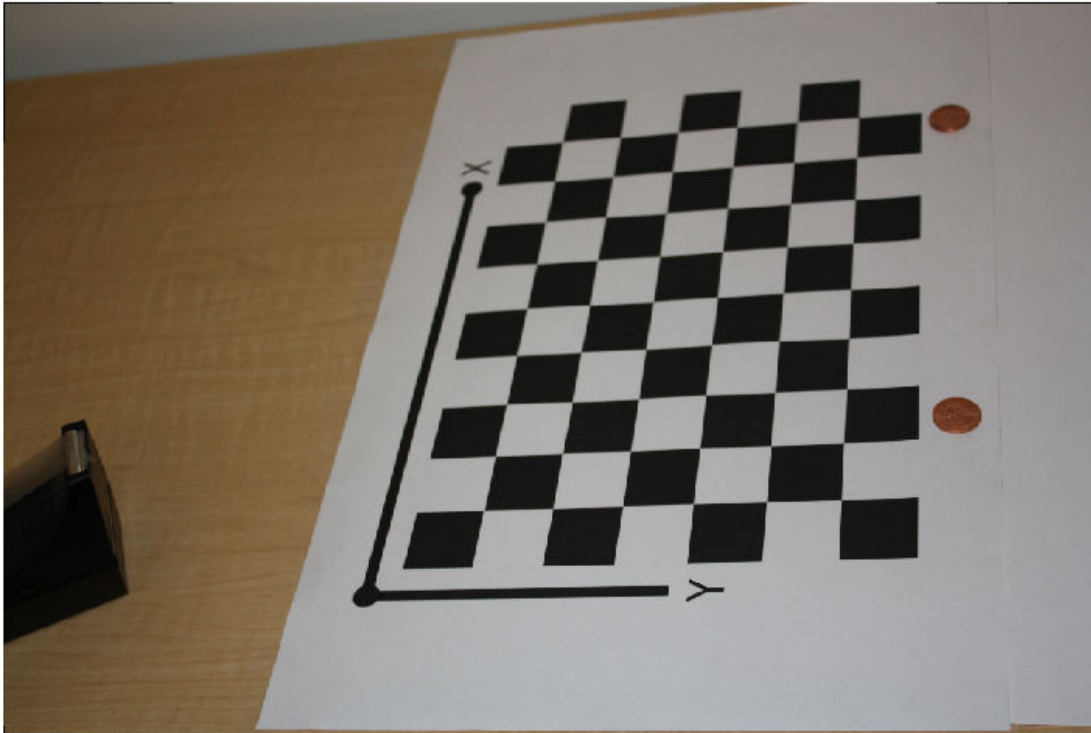


Undistort the Image

Use the `cameraParameters` object to remove lens distortion from the image. This is necessary for accurate measurement.

```
% Since the lens introduced little distortion, use 'full' output view to illustrate that
% the image was undistorted. If we used the default 'same' option, it would be difficult
% to notice any difference when compared to the original image. Notice the small black borders.
[im, newOrigin] = undistortImage(imOrig, cameraParams, 'OutputView', 'full');
figure; imshow(im, 'InitialMagnification', magnification);
title('Undistorted Image');
```

Undistorted Image



Note that this image exhibits very little lens distortion. The undistortion step is far more important if you use a wide-angle lens, or a low-end webcam.

Segment Coins

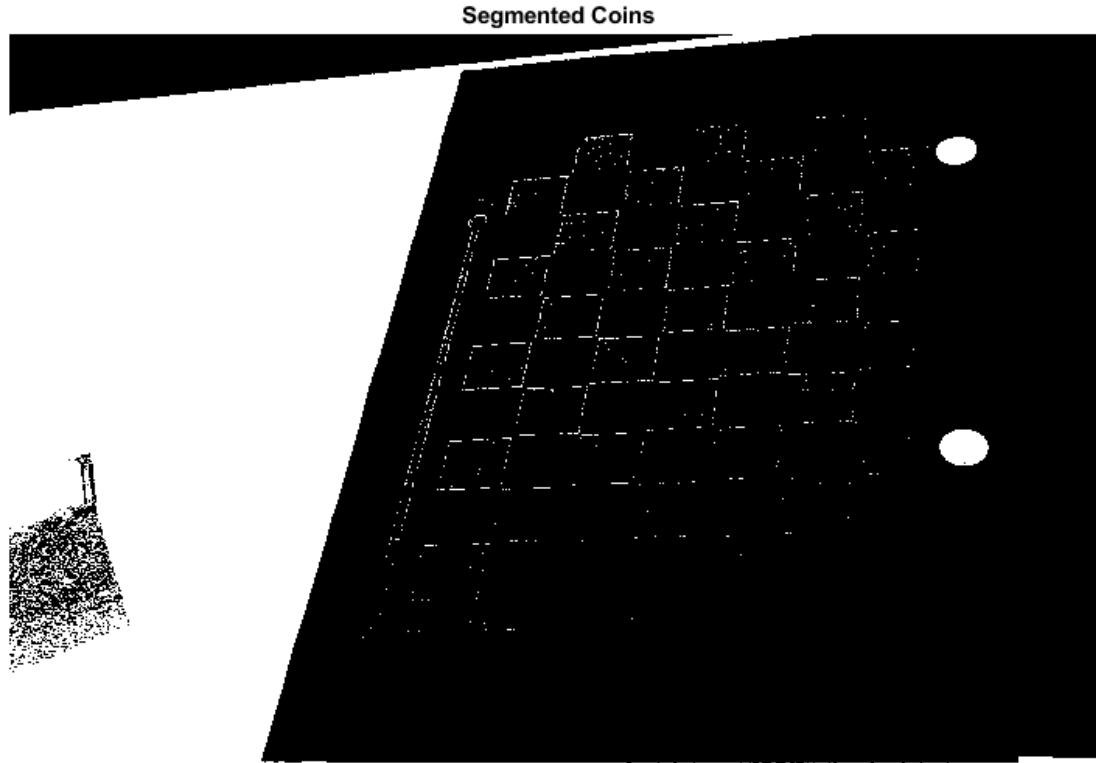
In this case, the coins are colorful on white background. Use the saturation component of the HSV representation of the image to segment them out.

```
% Convert the image to the HSV color space.
imHSV = rgb2hsv(im);

% Get the saturation channel.
saturation = imHSV(:, :, 2);

% Threshold the image
t = graythresh(saturation);
imCoin = (saturation > t);

figure; imshow(imCoin, 'InitialMagnification', magnification);
title('Segmented Coins');
```



Detect Coins

We can assume that the two largest connected components in the segmented image correspond to the coins.

```
% Find connected components.
blobAnalysis = vision.BlobAnalysis('AreaOutputPort', true,...
    'CentroidOutputPort', false,...
    'BoundingBoxOutputPort', true,...
    'MinimumBlobArea', 200, 'ExcludeBorderBlobs', true);
[areas, boxes] = step(blobAnalysis, imCoin);

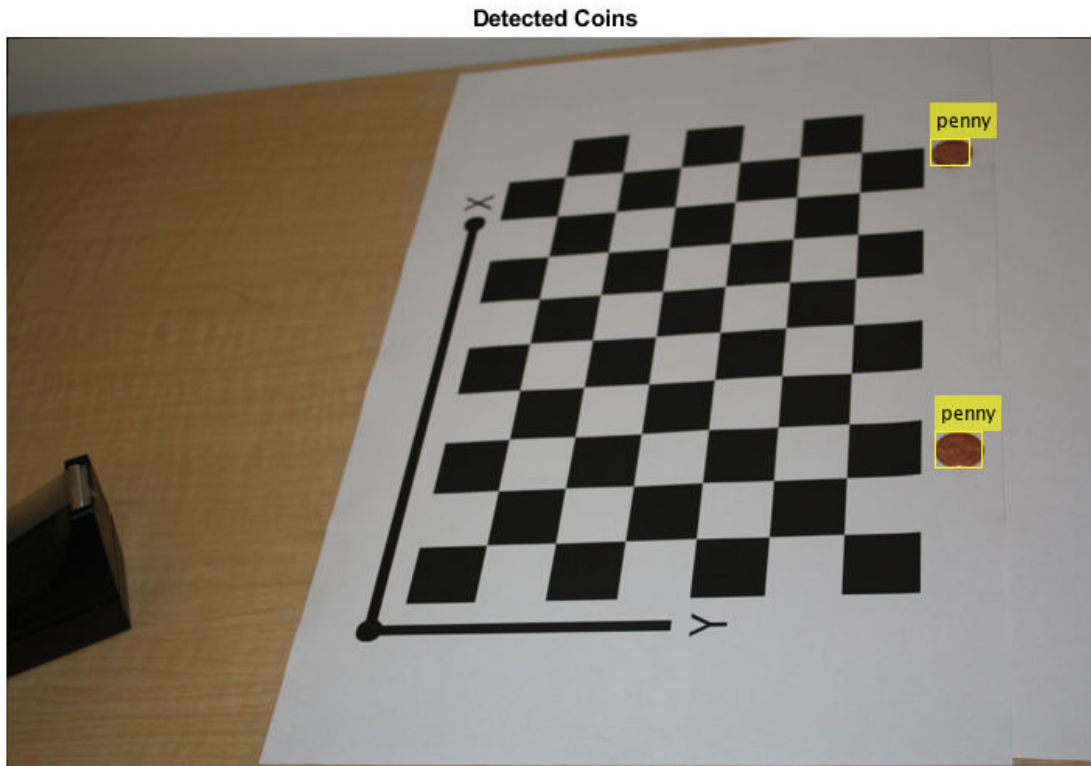
% Sort connected components in descending order by area
[~, idx] = sort(areas, 'Descend');

% Get the two largest components.
boxes = double(boxes(idx(1:2), :));

% Reduce the size of the image for display.
scale = magnification / 100;
imDetectedCoins = imresize(im, scale);

% Insert labels for the coins.
imDetectedCoins = insertObjectAnnotation(imDetectedCoins, 'rectangle', ...
    scale * boxes, 'penny');
```

```
figure; imshow(imDetectedCoins);
title('Detected Coins');
```



Compute Extrinsic

To map points in the image coordinates to points in the world coordinates we need to compute the rotation and the translation of the camera relative to the calibration pattern. Note that the `extrinsics` function assumes that there is no lens distortion. In this case `imagePoints` have been detected in an image that has already been undistorted using `undistortImage`.

```
% Detect the checkerboard.
[imagePoints, boardSize] = detectCheckerboardPoints(im);

% Adjust the imagePoints so that they are expressed in the coordinate system
% used in the original image, before it was undistorted. This adjustment
% makes it compatible with the cameraParameters object computed for the original image.
imagePoints = imagePoints + newOrigin; % adds newOrigin to every row of imagePoints

% Compute rotation and translation of the camera.
[R, t] = extrinsics(imagePoints, worldPoints, cameraParams);
```

Measure the First Coin

To measure the first coin we convert the top-left and the top-right corners of the bounding box into world coordinates. Then we compute the Euclidean distance between them in millimeters. Note that the actual diameter of a US penny is 19.05 mm.

```
% Adjust upper left corners of bounding boxes for coordinate system shift
% caused by undistortImage with output view of 'full'. This would not be
% needed if the output was 'same'. The adjustment makes the points compatible
% with the cameraParameters of the original image.
boxes = boxes + [newOrigin, 0, 0]; % zero padding is added for width and height
```

```
% Get the top-left and the top-right corners.
box1 = double(boxes(1, :));
imagePoints1 = [box1(1:2); ...
               box1(1) + box1(3), box1(2)];
```

```
% Get the world coordinates of the corners
worldPoints1 = pointsToWorld(cameraParams, R, t, imagePoints1);
```

```
% Compute the diameter of the coin in millimeters.
d = worldPoints1(2, :) - worldPoints1(1, :);
diameterInMillimeters = hypot(d(1), d(2));
fprintf('Measured diameter of one penny = %0.2f mm\n', diameterInMillimeters);
```

```
Measured diameter of one penny = 19.00 mm
```

Measure the Second Coin

Measure the second coin the same way as the first coin.

```
% Get the top-left and the top-right corners.
box2 = double(boxes(2, :));
imagePoints2 = [box2(1:2); ...
               box2(1) + box2(3), box2(2)];
```

```
% Apply the inverse transformation from image to world
worldPoints2 = pointsToWorld(cameraParams, R, t, imagePoints2);
```

```
% Compute the diameter of the coin in millimeters.
d = worldPoints2(2, :) - worldPoints2(1, :);
diameterInMillimeters = hypot(d(1), d(2));
fprintf('Measured diameter of the other penny = %0.2f mm\n', diameterInMillimeters);
```

```
Measured diameter of the other penny = 18.85 mm
```

Measure the Distance to The First Coin

In addition to measuring the size of the coin, we can also measure how far away it is from the camera.

```
% Compute the center of the first coin in the image.
center1_image = box1(1:2) + box1(3:4)/2;
```

```
% Convert to world coordinates.
center1_world = pointsToWorld(cameraParams, R, t, center1_image);
```

```
% Remember to add the 0 z-coordinate.
center1_world = [center1_world 0];
```

```
% Compute the distance to the camera.
[~, cameraLocation] = extrinsicsToCameraPose(R, t);
distanceToCamera = norm(center1_world - cameraLocation);
fprintf('Distance from the camera to the first penny = %0.2f mm\n', ...
       distanceToCamera);
```

Distance from the camera to the first penny = 719.52 mm

Summary

This example showed how to use a calibrated camera to measure planar objects. Note that the measurements were accurate to within 0.2 mm.

References

[1] Z. Zhang. A flexible new technique for camera calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(11):1330-1334, 2000.

Depth Estimation From Stereo Video

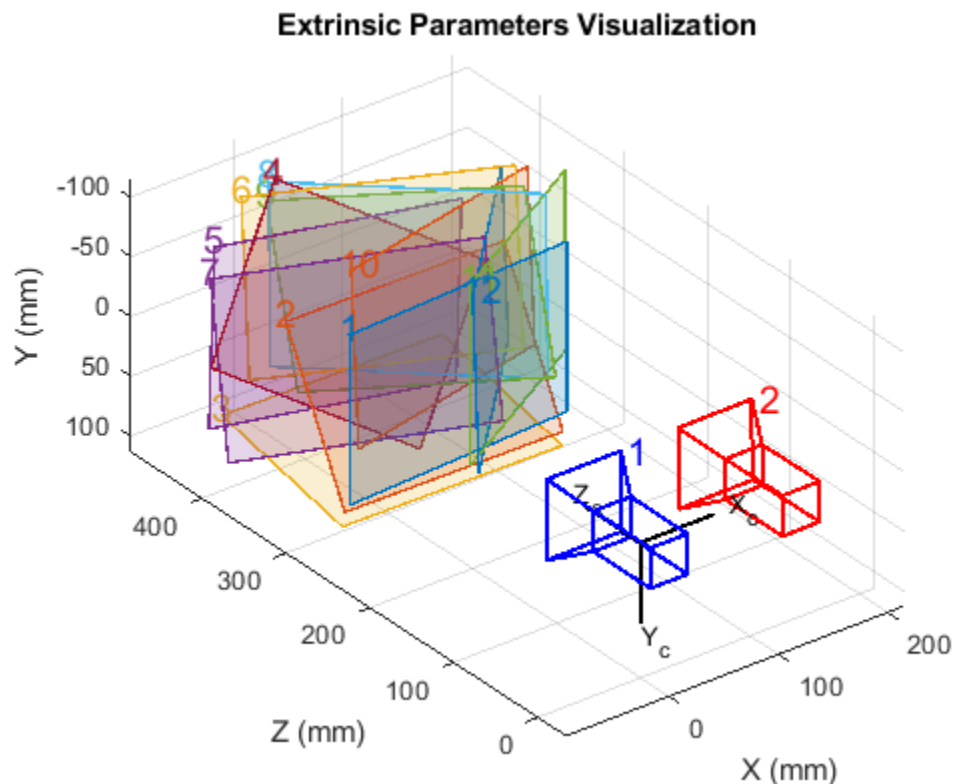
This example shows how to detect people in video taken with a calibrated stereo camera and determine their distances from the camera.

Load the Parameters of the Stereo Camera

Load the `stereoParameters` object, which is the result of calibrating the camera using either the `stereoCameraCalibrator` app or the `estimateCameraParameters` function.

```
% Load the stereoParameters object.
load('handshakeStereoParams.mat');

% Visualize camera extrinsics.
showExtrinsics(stereoParams);
```



Create Video File Readers and the Video Player

Create System Objects for reading and displaying the video.

```
videoFileLeft = 'handshake_left.avi';
videoFileRight = 'handshake_right.avi';

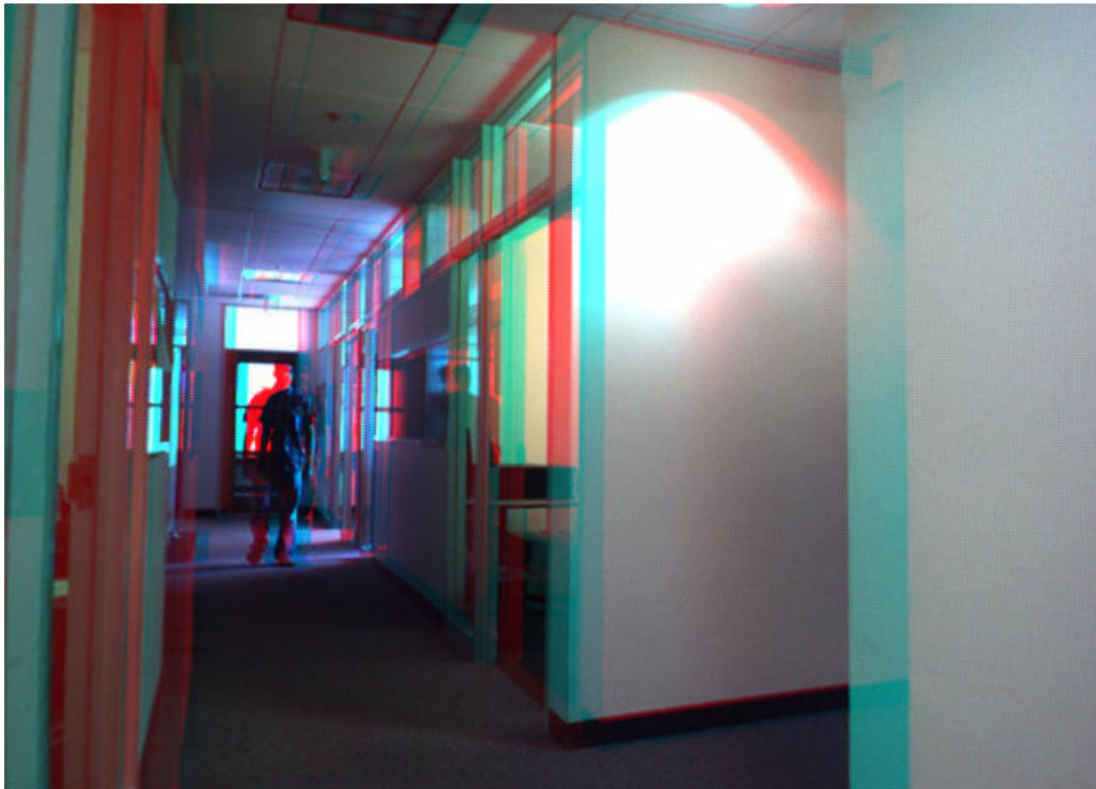
readerLeft = VideoReader(videoFileLeft);
readerRight = VideoReader(videoFileRight);
player = vision.VideoPlayer('Position', [20,200,740 560]);
```

Read and Rectify Video Frames

The frames from the left and the right cameras must be rectified in order to compute disparity and reconstruct the 3-D scene. Rectified images have horizontal epipolar lines, and are row-aligned. This simplifies the computation of disparity by reducing the search space for matching points to one dimension. Rectified images can also be combined into an anaglyph, which can be viewed using the stereo red-cyan glasses to see the 3-D effect.

```
frameLeft = readFrame(readerLeft);  
frameRight = readFrame(readerRight);  
  
[frameLeftRect, frameRightRect] = ...  
    rectifyStereoImages(frameLeft, frameRight, stereoParams);  
  
figure;  
imshow(stereoAnaglyph(frameLeftRect, frameRightRect));  
title('Rectified Video Frames');
```

Rectified Video Frames



Compute Disparity

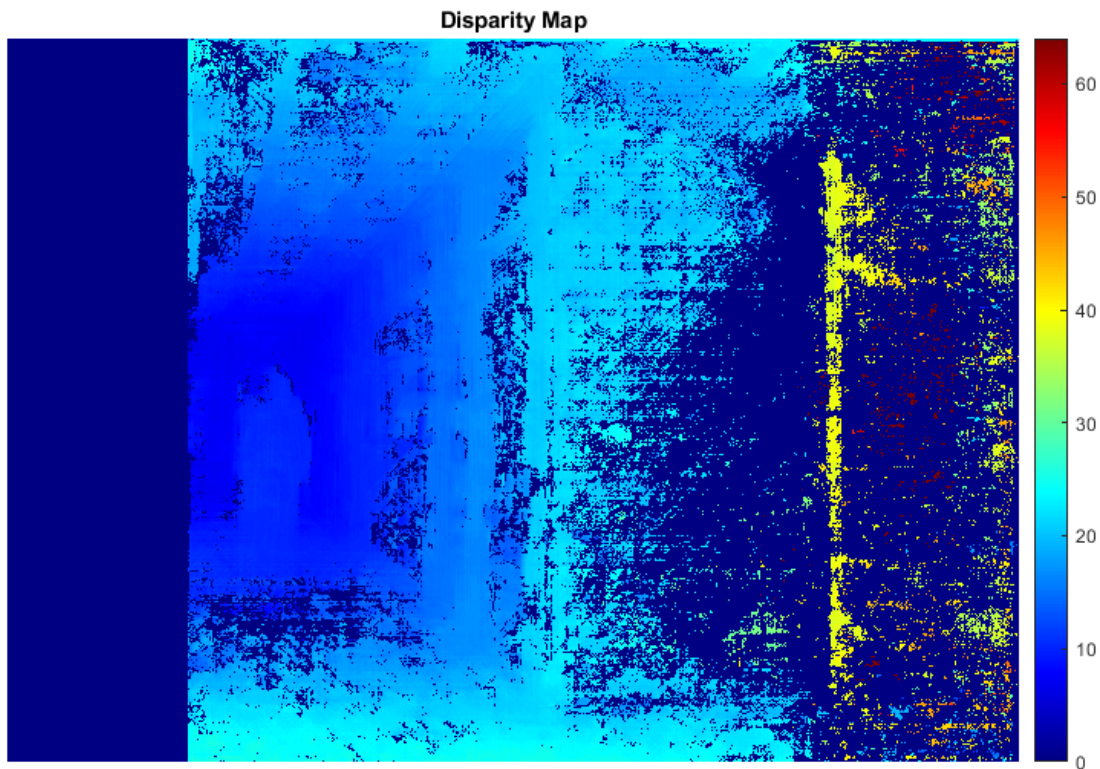
In rectified stereo images any pair of corresponding points are located on the same pixel row. For each pixel in the left image compute the distance to the corresponding pixel in the right image. This distance is called the disparity, and it is proportional to the distance of the corresponding world point from the camera.


```

frameLeftGray = rgb2gray(frameLeftRect);
frameRightGray = rgb2gray(frameRightRect);

disparityMap = disparitySGM(frameLeftGray, frameRightGray);
figure;
imshow(disparityMap, [0, 64]);
title('Disparity Map');
colormap jet
colorbar

```



Reconstruct the 3-D Scene

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

```

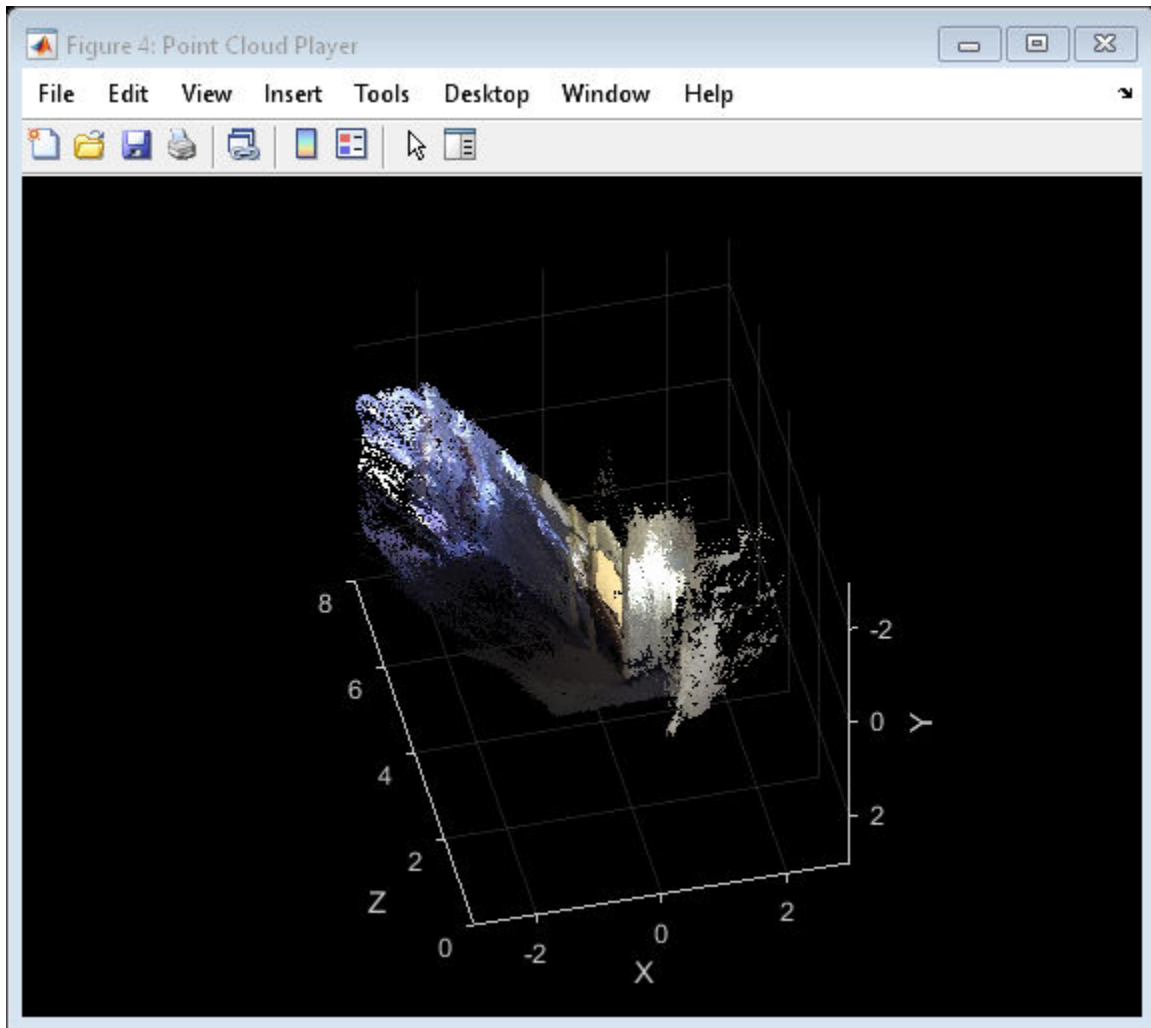
points3D = reconstructScene(disparityMap, stereoParams);

% Convert to meters and create a pointCloud object
points3D = points3D ./ 1000;
ptCloud = pointCloud(points3D, 'Color', frameLeftRect);

% Create a streaming point cloud viewer
player3D = pcplayer([-3, 3], [-3, 3], [0, 8], 'VerticalAxis', 'y', ...
    'VerticalAxisDir', 'down');

```

```
% Visualize the point cloud  
view(player3D, ptCloud);
```



Detect People in the Left Image

Use the `vision.PeopleDetector` system object to detect people.

```
% Create the people detector object. Limit the minimum object size for  
% speed.  
peopleDetector = vision.PeopleDetector('MinSize', [166 83]);
```

```
% Detect people.  
bboxes = peopleDetector.step(frameLeftGray);
```

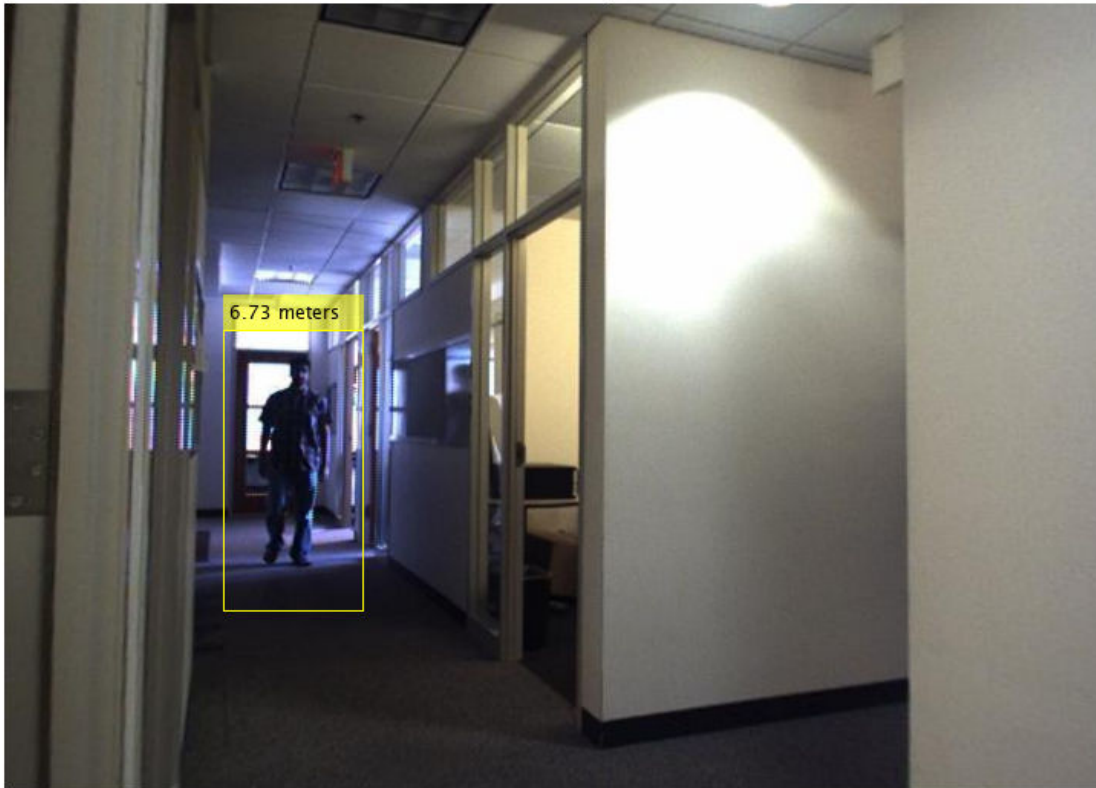
Determine The Distance of Each Person to the Camera

Find the 3-D world coordinates of the centroid of each detected person and compute the distance from the centroid to the camera in meters.

```
% Find the centroids of detected people.  
centroids = [round(bboxes(:, 1) + bboxes(:, 3) / 2), ...
```

```
round(bboxes(:, 2) + bboxes(:, 4) / 2]);  
  
% Find the 3-D world coordinates of the centroids.  
centroidsIdx = sub2ind(size(disparityMap), centroids(:, 2), centroids(:, 1));  
X = points3D(:, :, 1);  
Y = points3D(:, :, 2);  
Z = points3D(:, :, 3);  
centroids3D = [X(centroidsIdx)'; Y(centroidsIdx)'; Z(centroidsIdx)'];  
  
% Find the distances from the camera in meters.  
dists = sqrt(sum(centroids3D .^ 2));  
  
% Display the detected people and their distances.  
labels = cell(1, numel(dists));  
for i = 1:numel(dists)  
    labels{i} = sprintf('%0.2f meters', dists(i));  
end  
figure;  
imshow(insertObjectAnnotation(frameLeftRect, 'rectangle', bboxes, labels));  
title('Detected People');
```

Detected People



Process the Rest of the Video

Apply the steps described above to detect people and measure their distances to the camera in every frame of the video.

```
while hasFrame(readerLeft) && hasFrame(readerRight)
    % Read the frames.
    frameLeft = readFrame(readerLeft);
    frameRight = readFrame(readerRight);

    % Rectify the frames.
    [frameLeftRect, frameRightRect] = ...
        rectifyStereoImages(frameLeft, frameRight, stereoParams);

    % Convert to grayscale.
    frameLeftGray = rgb2gray(frameLeftRect);
    frameRightGray = rgb2gray(frameRightRect);

    % Compute disparity.
    disparityMap = disparitySGM(frameLeftGray, frameRightGray);

    % Reconstruct 3-D scene.
    points3D = reconstructScene(disparityMap, stereoParams);
    points3D = points3D ./ 1000;
    ptCloud = pointCloud(points3D, 'Color', frameLeftRect);
    view(player3D, ptCloud);

    % Detect people.
    bboxes = peopleDetector.step(frameLeftGray);

    if ~isempty(bboxes)
        % Find the centroids of detected people.
        centroids = [round(bboxes(:, 1) + bboxes(:, 3) / 2), ...
            round(bboxes(:, 2) + bboxes(:, 4) / 2)];

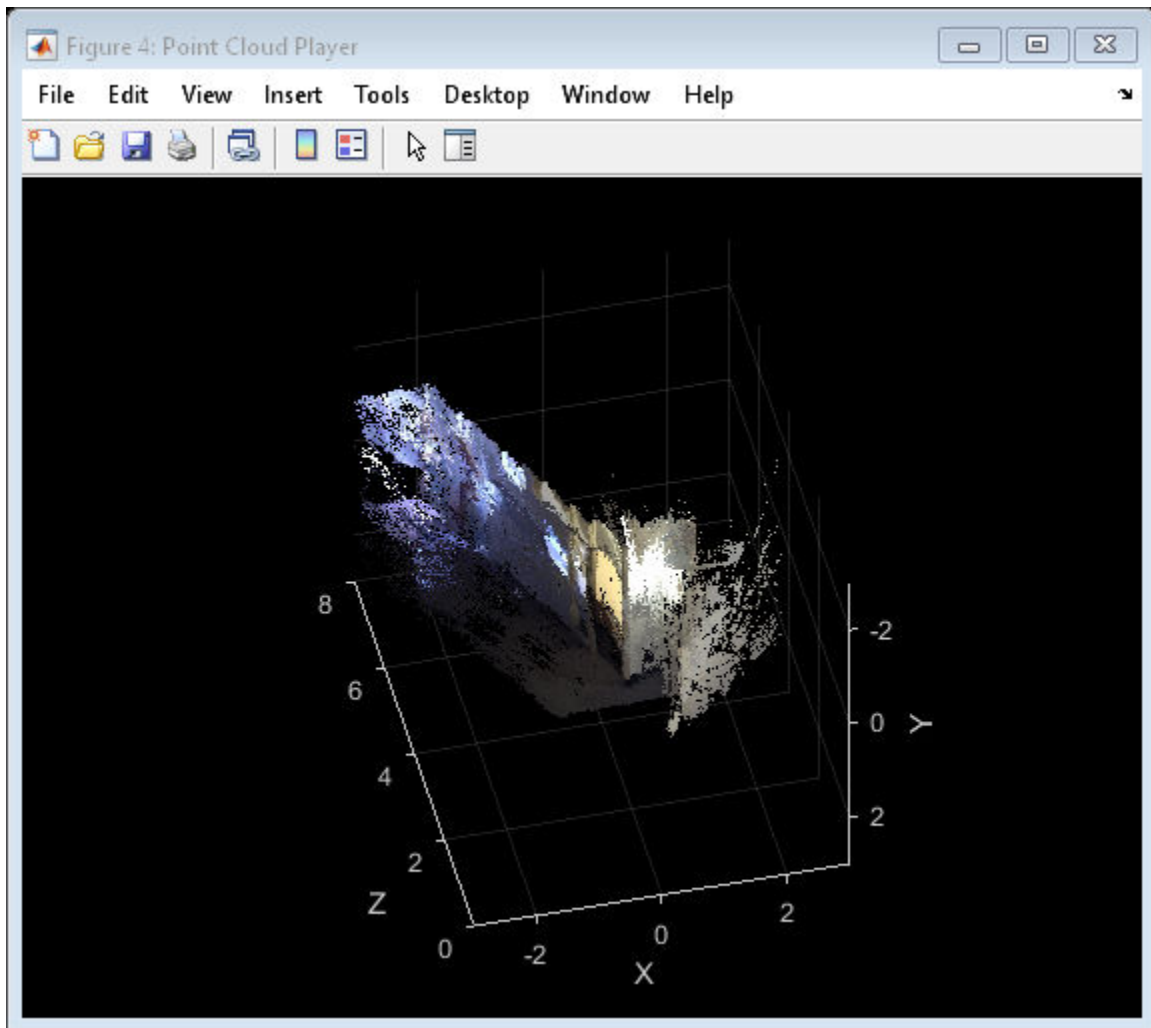
        % Find the 3-D world coordinates of the centroids.
        centroidsIdx = sub2ind(size(disparityMap), centroids(:, 2), centroids(:, 1));
        X = points3D(:, :, 1);
        Y = points3D(:, :, 2);
        Z = points3D(:, :, 3);
        centroids3D = [X(centroidsIdx), Y(centroidsIdx), Z(centroidsIdx)];

        % Find the distances from the camera in meters.
        dists = sqrt(sum(centroids3D.^2, 2));

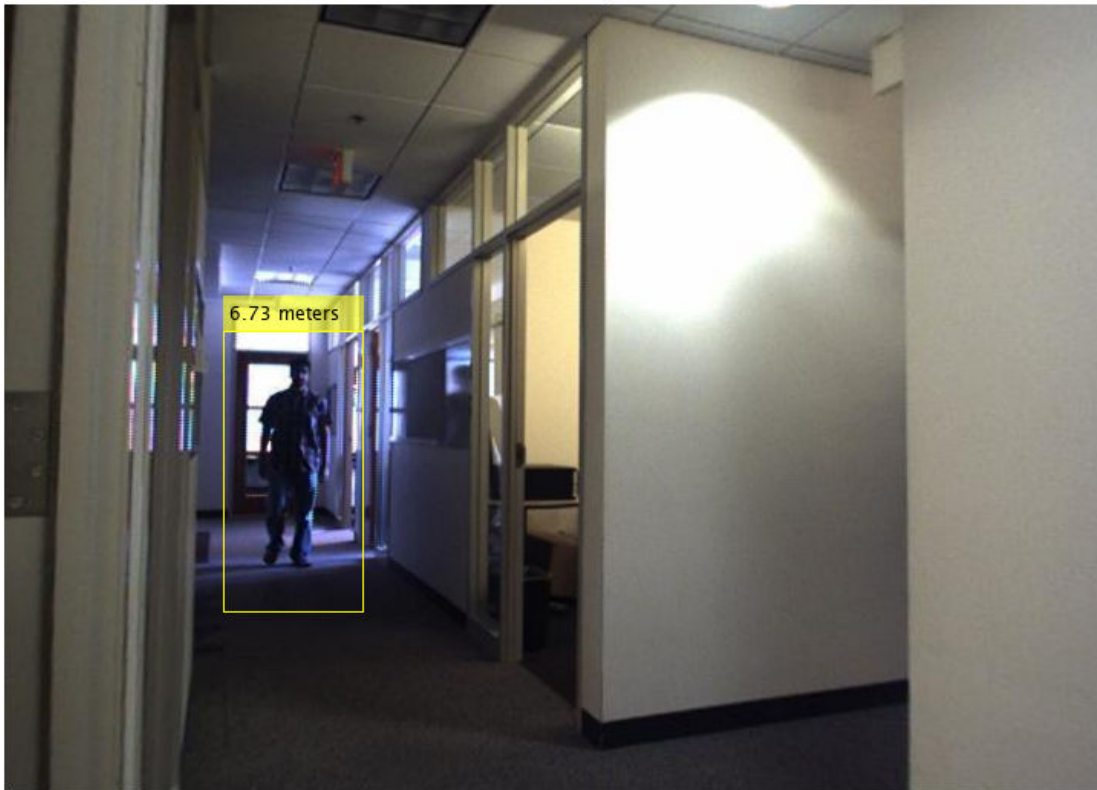
        % Display the detect people and their distances.
        labels = cell(1, numel(dists));
        for i = 1:numel(dists)
            labels{i} = sprintf('%0.2f meters', dists(i));
        end
        dispFrame = insertObjectAnnotation(frameLeftRect, 'rectangle', bboxes,...
            labels);
    else
        dispFrame = frameLeftRect;
    end

    % Display the frame.
```

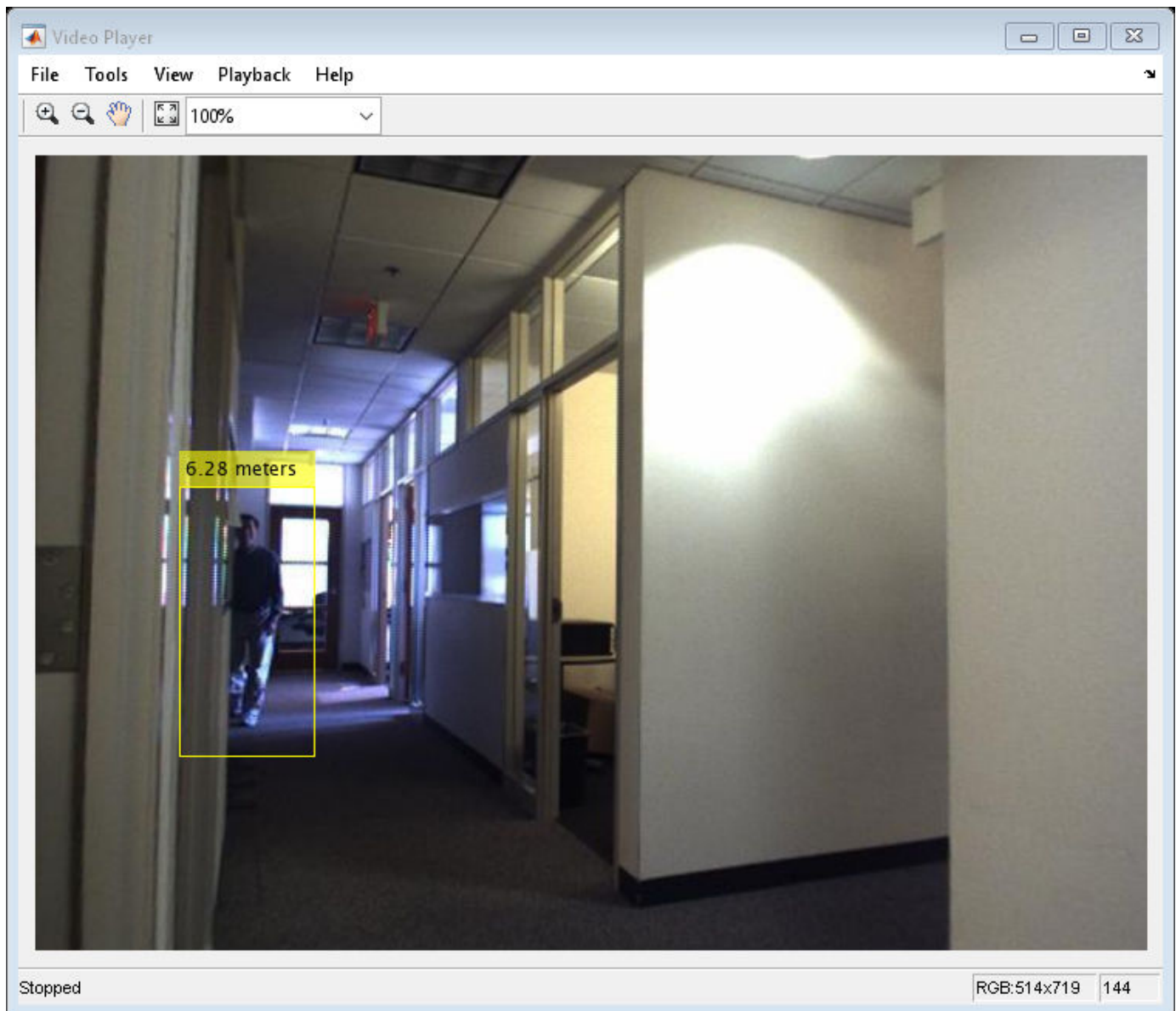
```
step(player, dispFrame);  
end
```



Detected People



```
% Clean up  
release(player);
```



Summary

This example showed how to localize pedestrians in 3-D using a calibrated stereo camera.

References

- [1] G. Bradski and A. Kaehler, "Learning OpenCV : Computer Vision with the OpenCV Library," O'Reilly, Sebastopol, CA, 2008.
- [2] Dalal, N. and Triggs, B., Histograms of Oriented Gradients for Human Detection. CVPR 2005.

Structure From Motion From Multiple Views

Structure from motion (SfM) is the process of estimating the 3-D structure of a scene from a set of 2-D views. It is used in many applications, such as robot navigation, autonomous driving, and augmented reality. This example shows you how to estimate the poses of a calibrated camera from a sequence of views, and reconstruct the 3-D structure of the scene up to an unknown scale factor.

Overview

This example shows how to reconstruct a 3-D scene from a sequence of 2-D views taken with a camera calibrated using the Camera Calibrator. The example uses an `imageviewset` object to store and manage the data associated with each view, such as the camera pose and the image points, as well as matches between points from pairs of views.

The example uses the pairwise point matches to estimate the camera pose of the current view relative to the previous view. It then links the pairwise matches into longer point tracks spanning multiple views using the `findTracks` method of the `imageviewset` object. These tracks then serve as inputs to multiview triangulation using the `triangulateMultiview` function and the refinement of camera poses and the 3-D scene points using the `bundleAdjustment` function.

The example consists of two main parts: camera motion estimation and dense scene reconstruction. In the first part, the example estimates the camera pose for each view using a sparse set of points matched across the views. In the second part, the example iterates over the sequence of views again, using `vision.PointTracker` to track a dense set of points across the views, to compute a dense 3-D reconstruction of the scene.

The camera motion estimation algorithm consists of the following steps:

- 1 For each pair of consecutive images, find a set of point correspondences. This example detects the interest points using the `detectSURFFeatures` function, extracts the feature descriptors using the `extractFeatures` functions, and finds the matches using the `matchFeatures` function. Alternatively, you can track the points across the views using `vision.PointTracker`.
- 2 Estimate the relative pose of the current view, which is the camera orientation and location relative to the previous view. The example uses a helper function `helperEstimateRelativePose`, which calls `estimateEssentialMatrix` and `relativeCameraPose`.
- 3 Transform the relative pose of the current view into the coordinate system of the first view of the sequence.
- 4 Store the current view attributes: the camera pose and the image points.
- 5 Store the inlier matches between the previous and the current view.
- 6 Find point tracks across all the views processed so far.
- 7 Use the `triangulateMultiview` function to compute the initial 3-D locations corresponding to the tracks.
- 8 Use the `bundleAdjustment` function to refine the camera poses and the 3-D points. Store the refined camera poses in the `imageviewset` object.

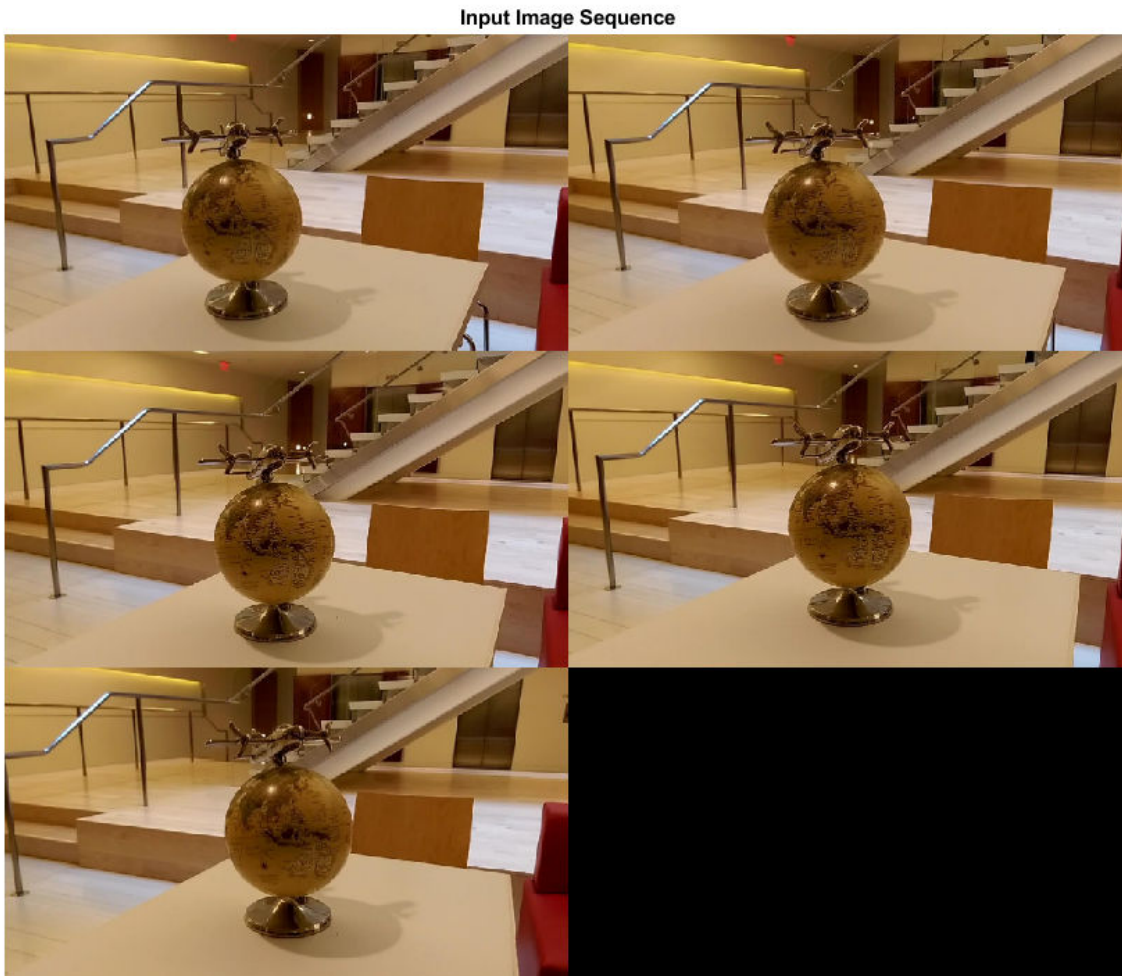
Read the Input Image Sequence

Read and display the image sequence.

```
% Use |imageDatastore| to get a list of all image file names in a  
% directory.
```



```
imageDir = fullfile(toolboxdir('vision'), 'visiondata', ...  
    'structureFromMotion');  
imds = imageDatastore(imageDir);  
  
% Display the images.  
figure  
montage(imds.Files, 'Size', [3, 2]);  
  
% Convert the images to grayscale.  
images = cell(1, numel(imds.Files));  
for i = 1:numel(imds.Files)  
    I = readimage(imds, i);  
    images{i} = rgb2gray(I);  
end  
  
title('Input Image Sequence');
```



Load Camera Parameters

Load the `cameraParameters` object created using the Camera Calibrator.

```
data = load(fullfile(imageDir, 'cameraParams.mat'));
cameraParams = data.cameraParams;
```

Create a View Set Containing the First View

Use an `imageviewset` object to store and manage the image points and the camera pose associated with each view, as well as point matches between pairs of views. Once you populate an `imageviewset` object, you can use it to find point tracks across multiple views and retrieve the camera poses to be used by `triangulateMultiview` and `bundleAdjustment` functions.

```
% Get intrinsic parameters of the camera
intrinsics = cameraParams.Intrinsics;

% Undistort the first image.
I = undistortImage(images{1}, intrinsics);

% Detect features. Increasing 'NumOctaves' helps detect large-scale
% features in high-resolution images. Use an ROI to eliminate spurious
% features around the edges of the image.
border = 50;
roi = [border, border, size(I, 2)- 2*border, size(I, 1)- 2*border];
prevPoints = detectSURFFeatures(I, 'NumOctaves', 8, 'ROI', roi);

% Extract features. Using 'Upright' features improves matching, as long as
% the camera motion involves little or no in-plane rotation.
prevFeatures = extractFeatures(I, prevPoints, 'Upright', true);

% Create an empty imageviewset object to manage the data associated with each
% view.
vSet = imageviewset;

% Add the first view. Place the camera associated with the first view
% and the origin, oriented along the Z-axis.
viewId = 1;
vSet = addView(vSet, viewId, rigid3d, 'Points', prevPoints);
```

Add the Rest of the Views

Go through the rest of the images. For each image

- 1 Match points between the previous and the current image.
- 2 Estimate the camera pose of the current view relative to the previous view.
- 3 Compute the camera pose of the current view in the global coordinate system relative to the first view.
- 4 Triangulate the initial 3-D world points.
- 5 Use bundle adjustment to refine all camera poses and the 3-D world points.

```
for i = 2:numel(images)
    % Undistort the current image.
    I = undistortImage(images{i}, intrinsics);

    % Detect, extract and match features.
```

```

currPoints = detectSURFFeatures(I, 'NumOctaves', 8, 'ROI', roi);
currFeatures = extractFeatures(I, currPoints, 'Upright', true);
indexPairs = matchFeatures(prevFeatures, currFeatures, ...
    'MaxRatio', .7, 'Unique', true);

% Select matched points.
matchedPoints1 = prevPoints(indexPairs(:, 1));
matchedPoints2 = currPoints(indexPairs(:, 2));

% Estimate the camera pose of current view relative to the previous view.
% The pose is computed up to scale, meaning that the distance between
% the cameras in the previous view and the current view is set to 1.
% This will be corrected by the bundle adjustment.
[relativeOrient, relativeLoc, inlierIdx] = helperEstimateRelativePose(...
    matchedPoints1, matchedPoints2, intrinsics);

% Get the table containing the previous camera pose.
prevPose = poses(vSet, i-1).AbsolutePose;
relPose = rigid3d(relativeOrient, relativeLoc);

% Compute the current camera pose in the global coordinate system
% relative to the first view.
currPose = rigid3d(relPose.T * prevPose.T);

% Add the current view to the view set.
vSet = addView(vSet, i, currPose, 'Points', currPoints);

% Store the point matches between the previous and the current views.
vSet = addConnection(vSet, i-1, i, relPose, 'Matches', indexPairs(inlierIdx,:));

% Find point tracks across all views.
tracks = findTracks(vSet);

% Get the table containing camera poses for all views.
camPoses = poses(vSet);

% Triangulate initial locations for the 3-D world points.
xyzPoints = triangulateMultiview(tracks, camPoses, intrinsics);

% Refine the 3-D world points and camera poses.
[xyzPoints, camPoses, reprojectionErrors] = bundleAdjustment(xyzPoints, ...
    tracks, camPoses, intrinsics, 'FixedViewId', 1, ...
    'PointsUndistorted', true);

% Store the refined camera poses.
vSet = updateView(vSet, camPoses);

prevFeatures = currFeatures;
prevPoints = currPoints;
end

```

Display Camera Poses

Display the refined camera poses and 3-D world points.

```

% Display camera poses.
camPoses = poses(vSet);
figure;

```

```

plotCamera(camPoses, 'Size', 0.2);
hold on

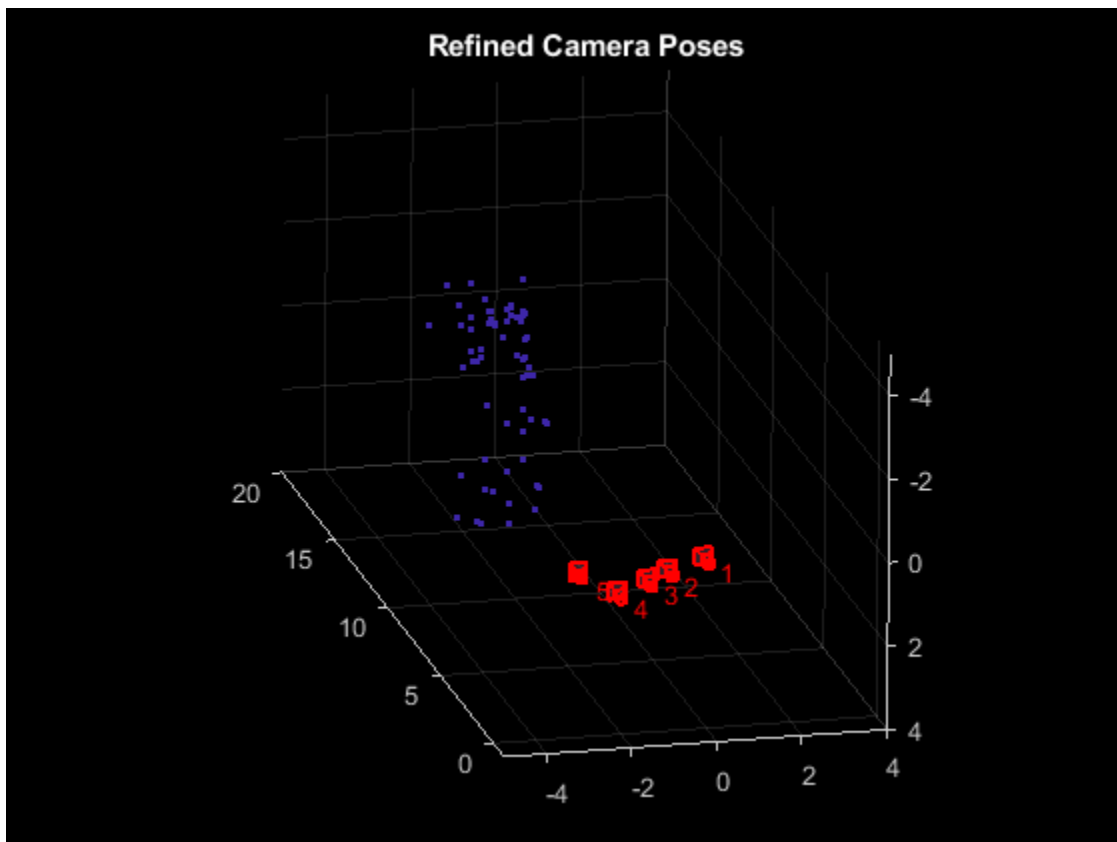
% Exclude noisy 3-D points.
goodIdx = (reprojectionErrors < 5);
xyzPoints = xyzPoints(goodIdx, :);

% Display the 3-D points.
pcshow(xyzPoints, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
       'MarkerSize', 45);
grid on
hold off

% Specify the viewing volume.
loc1 = camPoses.AbsolutePose(1).Translation;
xlim([loc1(1)-5, loc1(1)+4]);
ylim([loc1(2)-5, loc1(2)+4]);
zlim([loc1(3)-1, loc1(3)+20]);
camorbit(0, -30);

title('Refined Camera Poses');

```



Compute Dense Reconstruction

Go through the images again. This time detect a dense set of corners, and track them across all views using `vision.PointTracker`.

```

% Read and undistort the first image
I = undistortImage(images{1}, intrinsics);

% Detect corners in the first image.
prevPoints = detectMinEigenFeatures(I, 'MinQuality', 0.001);

% Create the point tracker object to track the points across views.
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels', 6);

% Initialize the point tracker.
prevPoints = prevPoints.Location;
initialize(tracker, prevPoints, I);

% Store the dense points in the view set.

vSet = updateConnection(vSet, 1, 2, 'Matches', zeros(0, 2));
vSet = updateView(vSet, 1, 'Points', prevPoints);

% Track the points across all views.
for i = 2:numel(images)
    % Read and undistort the current image.
    I = undistortImage(images{i}, intrinsics);

    % Track the points.
    [currPoints, validIdx] = step(tracker, I);

    % Clear the old matches between the points.
    if i < numel(images)
        vSet = updateConnection(vSet, i, i+1, 'Matches', zeros(0, 2));
    end
    vSet = updateView(vSet, i, 'Points', currPoints);

    % Store the point matches in the view set.
    matches = repmat((1:size(prevPoints, 1))', [1, 2]);
    matches = matches(validIdx, :);
    vSet = updateConnection(vSet, i-1, i, 'Matches', matches);
end

% Find point tracks across all views.
tracks = findTracks(vSet);

% Find point tracks across all views.
camPoses = poses(vSet);

% Triangulate initial locations for the 3-D world points.
xyzPoints = triangulateMultiview(tracks, camPoses,...
    intrinsics);

% Refine the 3-D world points and camera poses.
[xyzPoints, camPoses, reprojectionErrors] = bundleAdjustment(...
    xyzPoints, tracks, camPoses, intrinsics, 'FixedViewId', 1, ...
    'PointsUndistorted', true);

```

Display Dense Reconstruction

Display the camera poses and the dense point cloud.

```

% Display the refined camera poses.
figure;
plotCamera(camPoses, 'Size', 0.2);
hold on

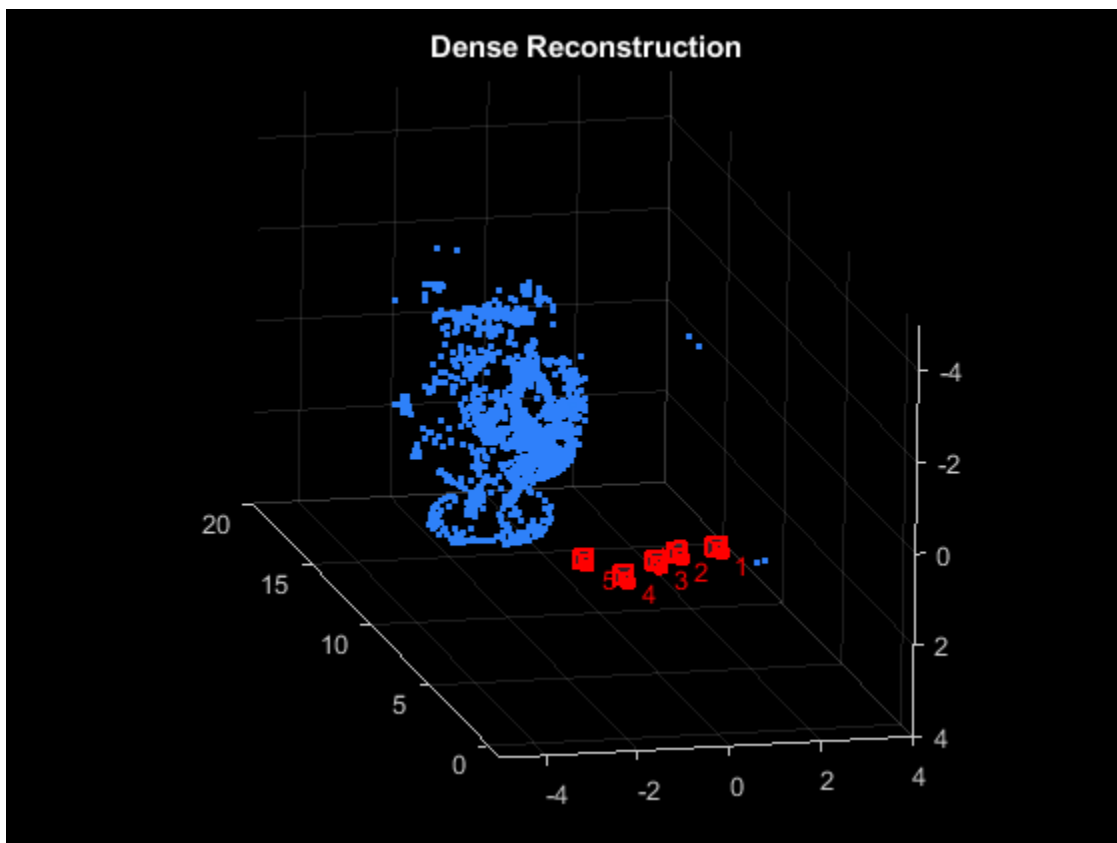
% Exclude noisy 3-D world points.
goodIdx = (reprojectionErrors < 5);

% Display the dense 3-D world points.
pcshow(xyzPoints(goodIdx, :), 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
       'MarkerSize', 45);
grid on
hold off

% Specify the viewing volume.
loc1 = camPoses.AbsolutePose(1).Translation;
xlim([loc1(1)-5, loc1(1)+4]);
ylim([loc1(2)-5, loc1(2)+4]);
zlim([loc1(3)-1, loc1(3)+20]);
camorbit(0, -30);

title('Dense Reconstruction');

```



References

- [1] M.I.A. Lourakis and A.A. Argyros (2009). "SBA: A Software Package for Generic Sparse Bundle Adjustment". ACM Transactions on Mathematical Software (ACM) 36 (1): 1-30.

[2] R. Hartley, A. Zisserman, "Multiple View Geometry in Computer Vision," Cambridge University Press, 2003.

[3] B. Triggs; P. McLauchlan; R. Hartley; A. Fitzgibbon (1999). "Bundle Adjustment: A Modern Synthesis". Proceedings of the International Workshop on Vision Algorithms. Springer-Verlag. pp. 298-372.

Uncalibrated Stereo Image Rectification

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

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

Step 1: Read Stereo Image Pair

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

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

```
% Convert to grayscale.  
I1gray = rgb2gray(I1);  
I2gray = rgb2gray(I2);
```

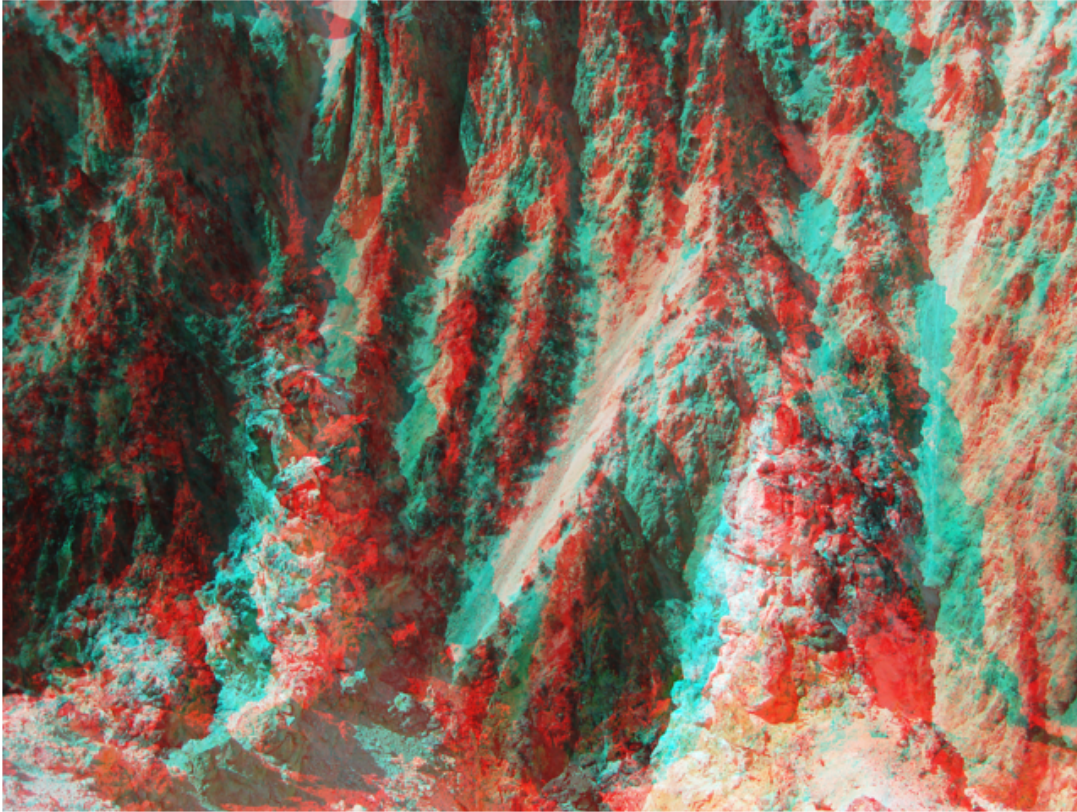
Display both images side by side. Then, display a color composite demonstrating the pixel-wise differences between the images.

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

I1 (left); I2 (right)



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



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

Step 2: Collect Interest Points from Each Image

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

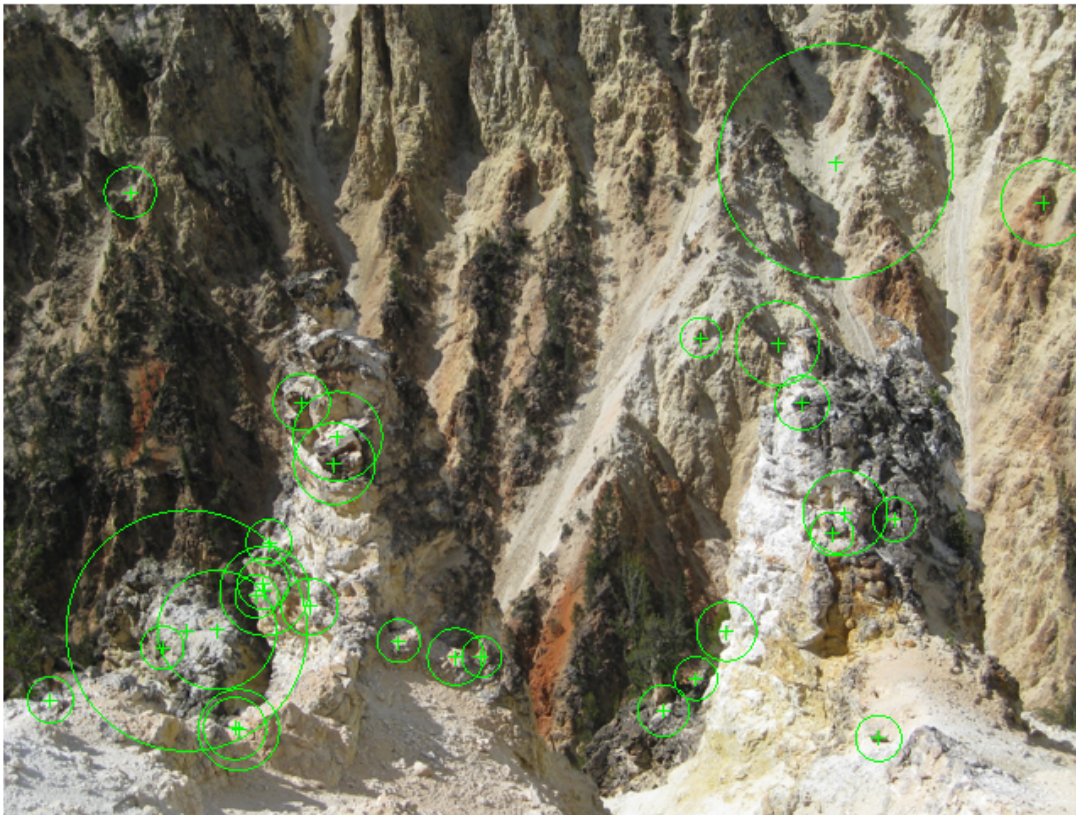
```
blobs1 = detectSURFFeatures(I1gray, 'MetricThreshold', 2000);
blobs2 = detectSURFFeatures(I2gray, 'MetricThreshold', 2000);
```

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

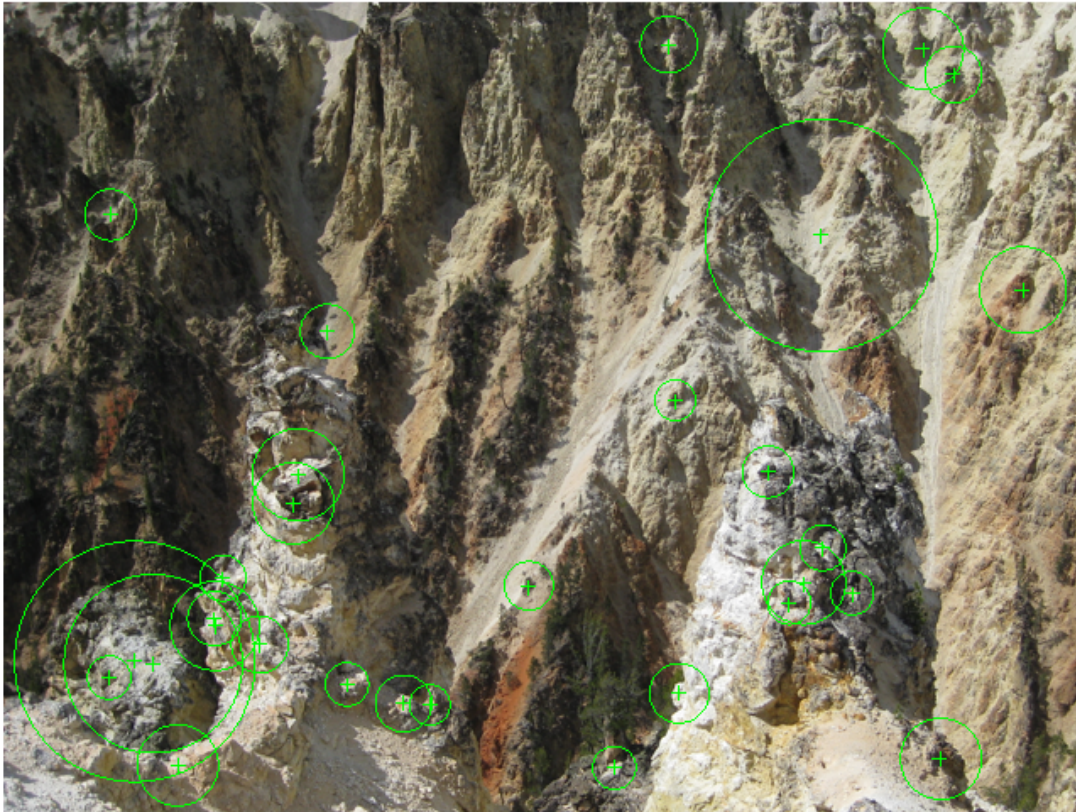
```
figure;
imshow(I1);
hold on;
plot(selectStrongest(blobs1, 30));
```

```
title('Thirty strongest SURF features in I1');  
  
figure;  
imshow(I2);  
hold on;  
plot(selectStrongest(blobs2, 30));  
title('Thirty strongest SURF features in I2');
```

Thirty strongest SURF features in I1



Thirty strongest SURF features in I2



Step 3: Find Putative Point Correspondences

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

```
[features1, validBlobs1] = extractFeatures(I1gray, blobs1);
[features2, validBlobs2] = extractFeatures(I2gray, blobs2);
```

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

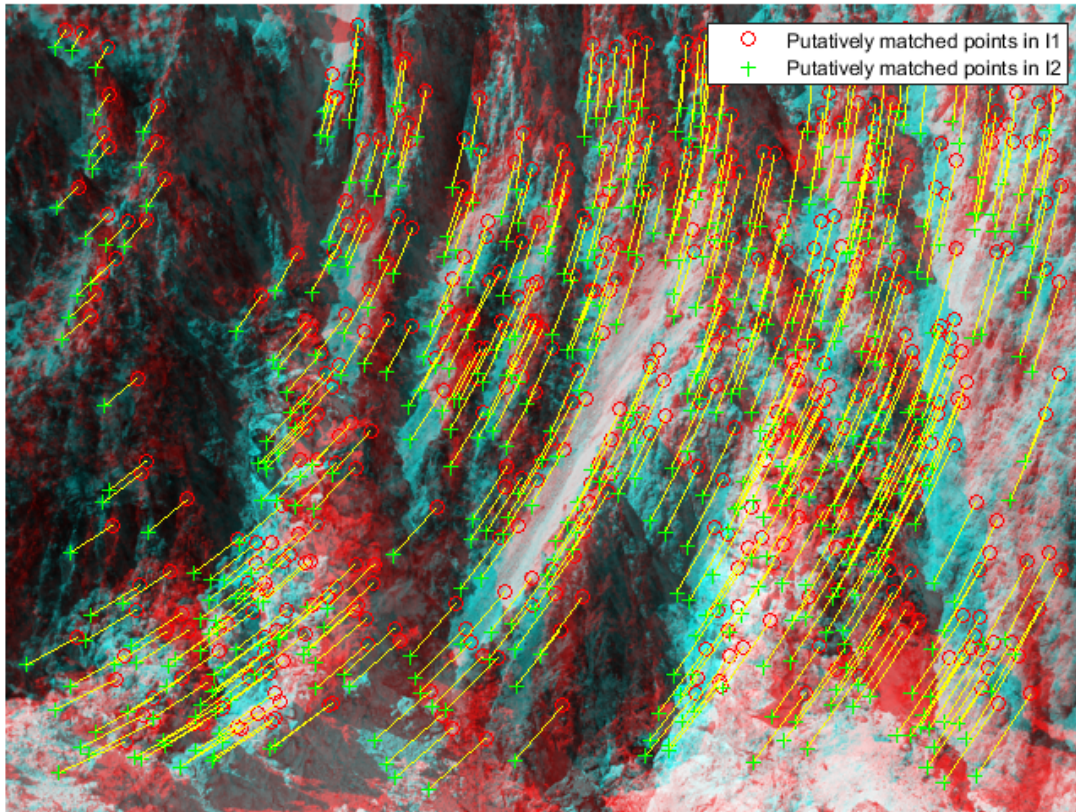
```
indexPairs = matchFeatures(features1, features2, 'Metric', 'SAD', ...
    'MatchThreshold', 5);
```

Retrieve locations of matched points for each image.

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

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

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



Step 4: Remove Outliers Using Epipolar Constraint

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

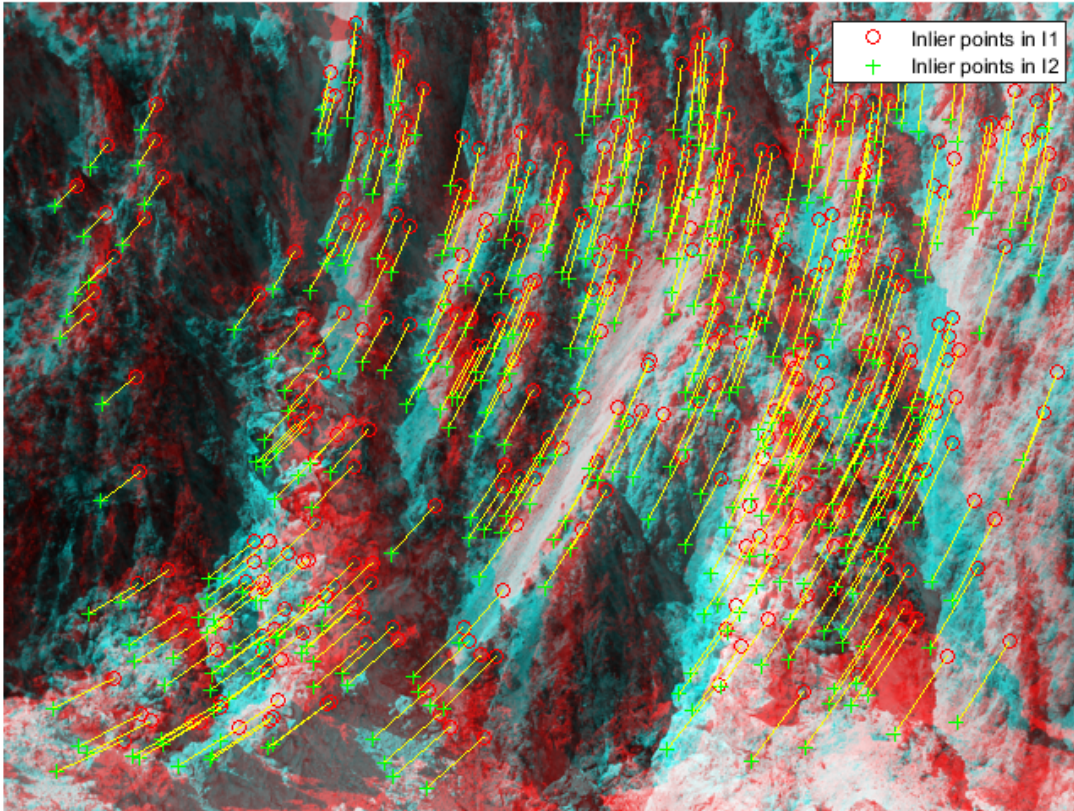
```
[fMatrix, epipolarInliers, status] = estimateFundamentalMatrix(...
    matchedPoints1, matchedPoints2, 'Method', 'RANSAC', ...
    'NumTrials', 10000, 'DistanceThreshold', 0.1, 'Confidence', 99.99);

if status ~= 0 || isEpipoleInImage(fMatrix, size(I1)) ...
    || isEpipoleInImage(fMatrix, size(I2))
    error(['Either not enough matching points were found or '...
        'the epipoles are inside the images. You may need to '...
        'inspect and improve the quality of detected features ',...
        'and/or improve the quality of your images.']);
end

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

figure;
```

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



Step 5: Rectify Images

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

```
[t1, t2] = estimateUncalibratedRectification(fMatrix, ...
    inlierPoints1.Location, inlierPoints2.Location, size(I2));
tform1 = projective2d(t1);
tform2 = projective2d(t2);
```

Rectify the stereo images, and display them as a stereo anaglyph. You can use red-cyan stereo glasses to see the 3D effect.

```
[I1Rect, I2Rect] = rectifyStereoImages(I1, I2, tform1, tform2);
figure;
imshow(stereoAnaglyph(I1Rect, I2Rect));
title('Rectified Stereo Images (Red - Left Image, Cyan - Right Image)');
```

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



Step 6: Generalize The Rectification Process

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

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

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

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

Code Generation and Third-Party Examples

- “Code Generation for Object Detection by Using Single Shot Multibox Detector” on page 2-2
- “Code Generation for Object Detection by Using YOLO v2” on page 2-5
- “Introduction to Code Generation with Feature Matching and Registration” on page 2-8
- “Code Generation for Face Tracking with PackNGo” on page 2-14
- “Code Generation for Depth Estimation From Stereo Video” on page 2-22
- “Detect Face (Raspberry Pi2)” on page 2-27
- “Track Face (Raspberry Pi2)” on page 2-33
- “Video Display in a Custom User Interface” on page 2-39

Code Generation for Object Detection by Using Single Shot Multibox Detector

This example shows how to generate CUDA® code for an SSD network (`ssdObjectDetector` object) and take advantage of the NVIDIA® cuDNN and TensorRT libraries. An SSD network is based on a feed-forward convolutional neural network that detect multiple objects within the image in a single shot. SSD network can be thought of as having two sub-networks. A feature extraction network, followed by a detection network.

This example generates code for the network trained in the *Object Detection Using SSD Deep Learning* example from Computer Vision Toolbox™. For more information, see “Object Detection Using SSD Deep Learning” on page 3-23. The *Object Detection Using SSD Deep Learning* example uses ResNet-50 for feature extraction. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to SSD.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Get Pretrained DAGNetwork

```
net = getSSDNW();
```

```
Downloading pretrained detector (44 MB)...
```

The DAG network contains 180 layers including convolution, ReLU, and batch normalization layers, anchor box, SSD merge, focal loss, and other layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

The `ssdObj_detect` Entry-Point Function

The `ssdObj_detect.m` entry-point function takes an image input and runs the detector on the image using the deep learning network saved in the `ssdResNet50VehicleExample_20a.mat` file. The function loads the network object from the `ssdResNet50VehicleExample_20a.mat` file into a persistent variable `ssdObj` and reuses the persistent object on subsequent detection calls.

```
type('ssdObj_detect.m')

function outImg = ssdObj_detect(in)

% Copyright 2019-2020 The MathWorks, Inc.

persistent ssdObj;

if isempty(ssdObj)
    ssdObj = coder.loadDeepLearningNetwork('ssdResNet50VehicleExample_20a.mat');
end

% Pass in input
[bboxes,~,labels] = ssdObj.detect(in,'Threshold',0.7);

% Convert categorical labels to cell array of character vectors for
% execution
labels = cellstr(labels);

% Annotate detections in the image.
if ~isempty(labels)
    outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
else
    outImg = in;
end
```

Run MEX Code Generation

To generate CUDA code for the `ssdObj_detect.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of [300,300,3]. This value corresponds to the input layer size of SSD Network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg ssdObj_detect -args {ones(300,300,3,'uint8')} -report
```

Code generation successful: To view the report, open('codegen/mex/ssdObj_detect/html/report.mldat

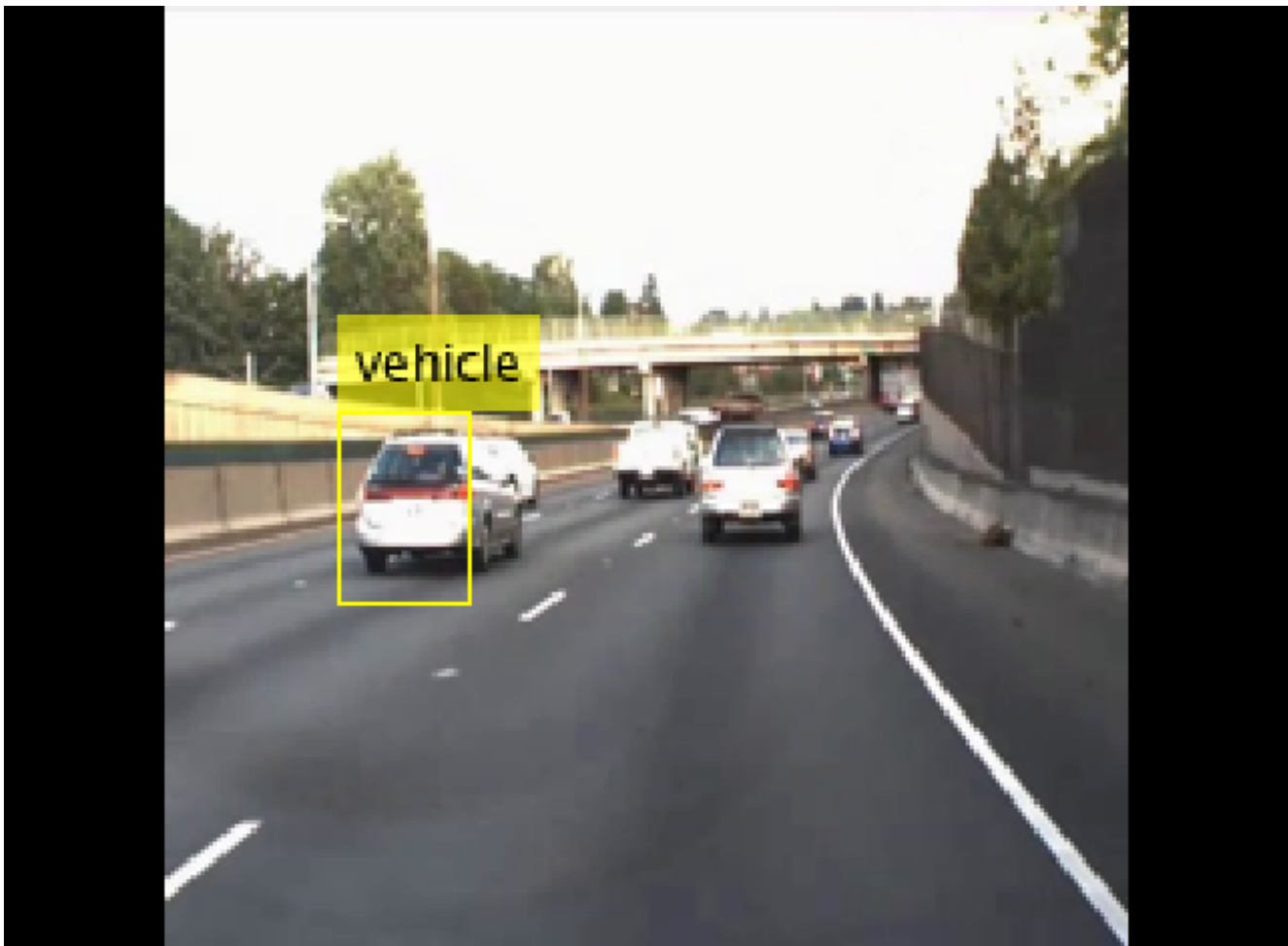
Run Generated MEX

Set up the video file reader and read the input video. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';  
videoFreader = vision.VideoFileReader(videoFile, 'VideoOutputDataType', 'uint8');  
depVideoPlayer = vision.DeployableVideoPlayer('Size', 'Custom', 'CustomSize', [640 480]);
```

Read the video input frame-by-frame and detect the vehicles in the video using the detector.

```
cont = ~isDone(videoFreader);  
while cont  
    I = step(videoFreader);  
    in = imresize(I, [300, 300]);  
    out = ssdObj_detect_mex(in);  
    step(depVideoPlayer, out);  
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer); % Exit the loop if the video player  
end
```



References

[1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single shot multibox detector." In 14th European Conference on Computer Vision, ECCV 2016. Springer Verlag, 2016.

Code Generation for Object Detection by Using YOLO v2

This example shows how to generate CUDA® MEX for a you only look once (YOLO) v2 object detector. A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. This example generates code for the network trained in the *Object Detection Using YOLO v2 Deep Learning* example from Computer Vision Toolbox™. For more information, see “Object Detection Using YOLO v2 Deep Learning” on page 3-123. You can modify this example to generate CUDA® MEX for the network imported in the *Import Pretrained ONNX YOLO v2 Object Detector* example from Computer Vision Toolbox™. For more information, see “Import Pretrained ONNX YOLO v2 Object Detector” on page 3-133.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Get Pretrained DAGNetwork

```
net = getYOLOv2();
```

The DAG network contains 150 layers including convolution, ReLU, and batch normalization layers and the YOLO v2 transform and YOLO v2 output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

The `yolov2_detect` Entry-Point Function

The `yolov2_detect.m` entry-point function takes an image input and runs the detector on the image using the deep learning network saved in the `yolov2ResNet50VehicleExample.mat` file. The function loads the network object from the `yolov2ResNet50VehicleExample.mat` file into a persistent variable `yolov2Obj` and reuses the persistent object on subsequent detection calls.

```
type('yolov2_detect.m')

function outImg = yolov2_detect(in)

% Copyright 2018-2019 The MathWorks, Inc.

persistent yolov2obj;

if isempty(yolov2obj)
    yolov2obj = coder.loadDeepLearningNetwork('yolov2ResNet50VehicleExample.mat');
end

% pass in input
[bboxes,~,labels] = yolov2obj.detect(in,'Threshold',0.5);

% convert categorical labels to cell array of character vectors for MATLAB
% execution
if coder.target('MATLAB')
    labels = cellstr(labels);
end

% Annotate detections in the image.
outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
```

Run MEX Code Generation

To generate CUDA code for the `yolov2_detect.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of `[224,224,3]`. This value corresponds to the input layer size of YOLOv2.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg yolov2_detect -args {ones(224,224,3,'uint8')} -report
```

Code generation successful: To view the report, open('codegen/mex/yolov2_detect/html/report.mldat')

Run Generated MEX

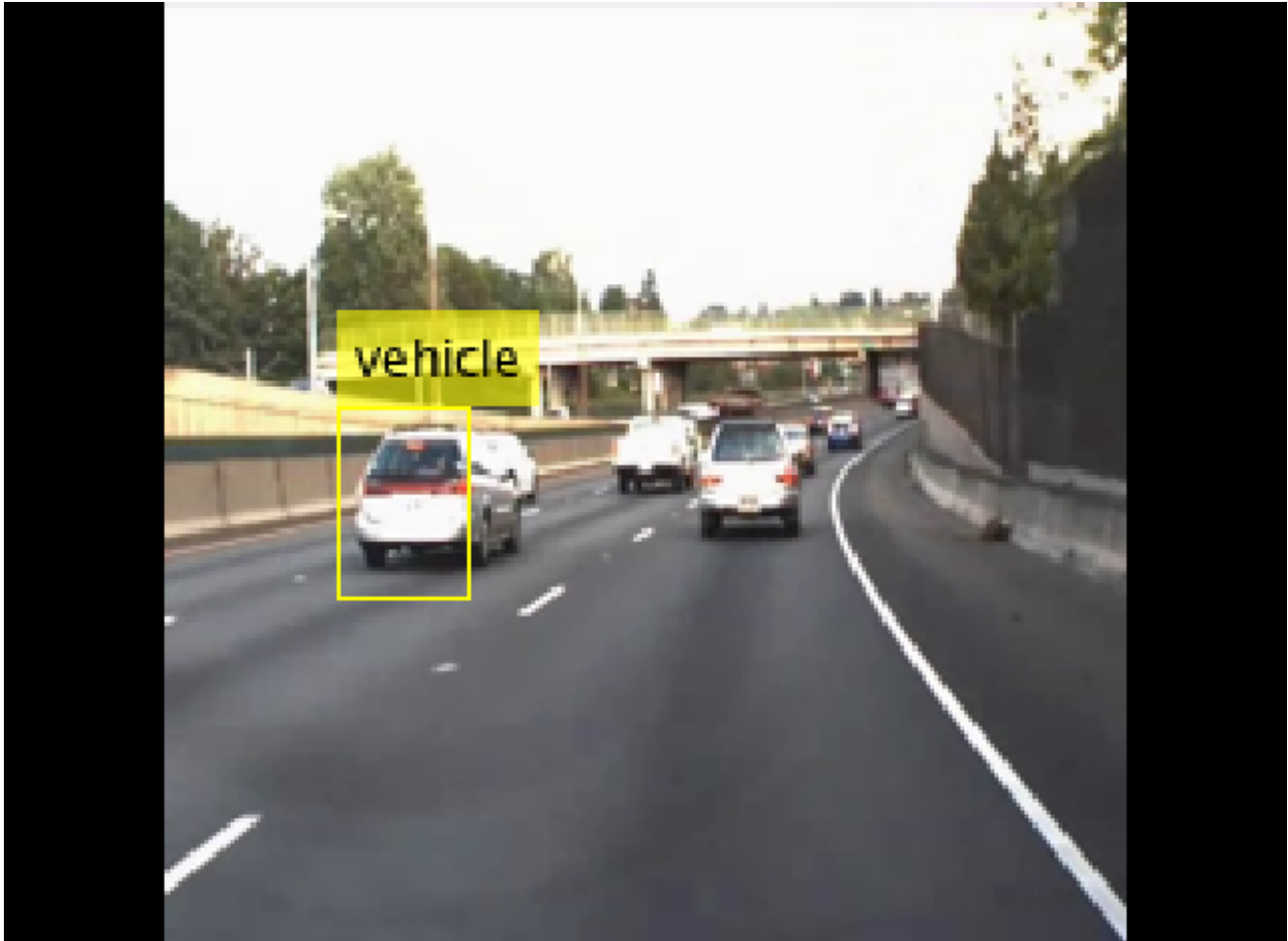
Set up the video file reader and read the input video. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';
videoFreader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
depVideoPlayer = vision.DeployableVideoPlayer('Size','Custom','CustomSize',[640 480]);
```

Read the video input frame-by-frame and detect the vehicles in the video using the detector.

```
cont = ~isDone(videoFreader);
while cont
    I = step(videoFreader);
    in = imresize(I,[224,224]);
    out = yolov2_detect_mex(in);
```

```
step(depVideoPlayer, out);  
cont = ~isDone(videoFreader) && isOpen(depVideoPlayer); % Exit the loop if the video player  
end
```



References

- [1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

Introduction to Code Generation with Feature Matching and Registration

This example shows how to use the MATLAB® Coder™ to generate C code for a MATLAB file. The example explains how to modify the MATLAB code used by the “Find Image Rotation and Scale Using Automated Feature Matching” on page 4-22 example so that it is supported for code generation. The example highlights some of the general requirements for code generation, as well as some of the specific actions you must take to prepare MATLAB code. Once the MATLAB code is ready for code generation, you use the `codegen` (MATLAB Coder) command to generate a C-MEX function. Finally, to verify results, the example shows you how to run the generated C-MEX function in MATLAB and compare its output with the output of the MATLAB code.

This example requires a MATLAB Coder license.

Set Up Your C Compiler

To run this example, you must have access to a C compiler and you must configure it using 'mex -setup' command. For more information, see “Get Started with MATLAB Coder” (MATLAB Coder).

Decide Whether to Run Under MATLAB or as a Standalone Application

Generated code can run inside the MATLAB environment as a C-MEX file, or outside the MATLAB environment as a standalone executable or shared utility to be linked with another standalone executable. For more details about setting code generation options, see the `-config` option of the `codegen` (MATLAB Coder) command.

MEX Executables

This example generates a MEX executable to be run inside the MATLAB environment.

Generating a C-MEX executable to run inside of MATLAB can also be a great first step in a workflow that ultimately leads to standalone code. The inputs and the outputs of the MEX-file are available for inspection in the MATLAB environment, where visualization and other kinds of tools for verification and analysis are readily available. You also have the choice of running individual commands either as generated C code, or via the MATLAB engine. To run via MATLAB, declare relevant commands as `coder.extrinsic` (MATLAB Coder), which means that the generated code will re-enter the MATLAB environment when it needs to run that particular command. This is useful in cases where either an isolated command does not yet have code generation support, or if you wish to embed certain commands that do not generate code (such as plot command).

Standalone Executables

If deployment of code to another application is the goal, then a standalone executable will be required. The first step is to configure MATLAB Coder appropriately. For example, one way to tell it you want a standalone executable is to create a MATLAB Coder project using the MATLAB Coder IDE and configure that project to generate a module or an executable. You can do so using the C/C++ static library or C/C++ executable options from the Build type widget on the Generate page. This IDE is available by navigating as follows:

- Click APPS tab - Scroll down to MATLAB Coder - In MATLAB Coder Project dialog box, click OK

You can also define a config object using

```
a=coder.config('exe')
```


and pass that object to the coder command on the MATLAB command line. When you create a standalone executable, you have to write your own main.c (or main.cpp). Note that when you create a standalone executable, there are no ready-made utilities for importing or exporting data between the executable and the MATLAB environment. One of the options is to use printf/fprintf to a file (in your handwritten main.c) and then import data into MATLAB using 'load -ascii' with your file.

Break Out the Computational Part of the Algorithm into a Separate MATLAB Function

MATLAB Coder requires MATLAB code to be in the form of a function in order to generate C code. Note that it is generally not necessary to generate C code for all of the MATLAB code in question. It is often desirable to separate the code into the primary computational portion, from which C code generation is desired, and a harness or driver, which does not need to generate C code - that code will run in MATLAB. The harness may contain visualization and other verification aids that are not actually part of the system under test. The code for the main algorithm of this example resides in a function called `visionRecoverFromCodeGeneration_kernel`

Once the code has been re-architected as described above, you must check that the rest of the code uses capabilities that are supported by MATLAB coder. For a list of supported commands, see MATLAB Coder “Functions and Objects Supported for C/C++ Code Generation” (MATLAB Coder). For a list of supported language constructs, see “MATLAB Language Features Supported for C/C++ Code Generation” (MATLAB Coder).

It may be convenient to have limited visualization or some other capability that is not supported by the MATLAB Coder present in the function containing the main algorithm, which we hope to compile. In these cases, you can declare these items 'extrinsic' (using `coder.extrinsic`). Such capability is only possible when you generate the C code into a MATLAB MEX-file, and those functions will actually run in interpreted MATLAB mode. If generating code for standalone use, extrinsic functions are either ignored or they generate an error, depending on whether the code generation engine determines that they affect the results. Thus the code must be properly architected so that the extrinsic functions do not materially affect the code in question if a standalone executable is ultimately desired.

The original example uses `showMatchedFeatures` and `imshowpair` routines for visualization of the results. These routines are extracted to a new function `featureMatchingVisualization_extrinsic`. This function is declared extrinsic.

Run the Simulation

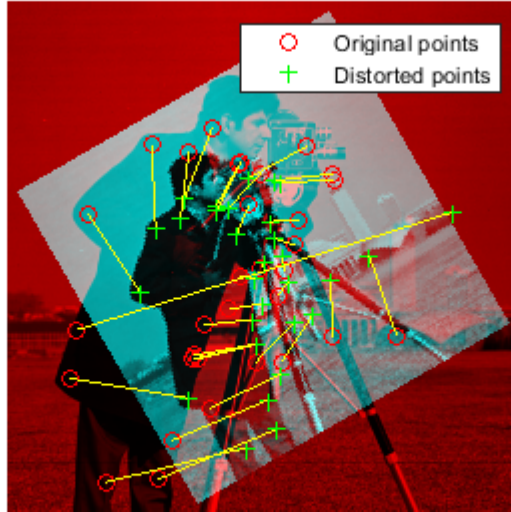
The kernel file `visionRecoverFromCodeGeneration_kernel.m` has two input parameters. The first input is the original image and the second input is the image distorted by rotation and scale.

```
% define original image
original = imread('cameraman.tif');
% define distorted image by resizing and then rotating original image
scale = 0.7;
J = imresize(original, scale);
theta = 30;
distorted = imrotate(J, theta);
% call the generated mex file
[matchedOriginalLoc, matchedDistortedLoc, ...
  thetaRecovered, ...
  scaleRecovered, recovered] = ...
  visionRecoverFromCodeGeneration_kernel(original, distorted);

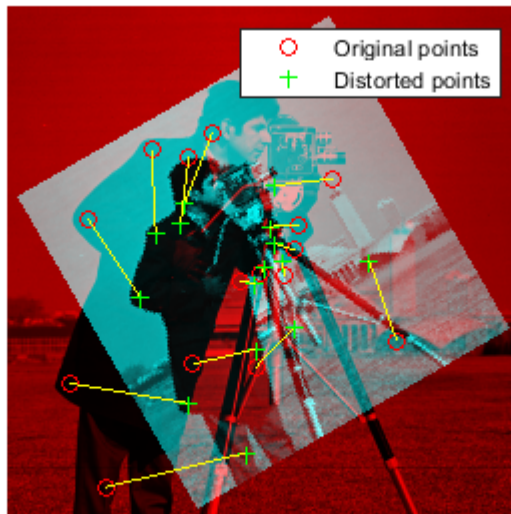
scaleRecovered = 0.701003
```

thetaRecovered = 30.235065

Putatively matched points (including outliers)



Matching points (inliers only)





Compile the MATLAB Function Into a MEX File

Now use the `codegen` (MATLAB Coder) function to compile the `visionRecoverformCodeGeneration_kernel` function into a MEX-file. You can specify the `'-report'` option to generate a compilation report that shows the original MATLAB code and the associated files that were created during C code generation. You may want to create a temporary directory where MATLAB Coder can create new files. Note that the generated MEX-file has the same name as the original MATLAB file with `_mex` appended, unless you use the `-o` option to specify the name of the executable.

MATLAB Coder requires that you specify the properties of all the input parameters. One easy way to do this is to define the input properties by example at the command-line using the `-args` option. For more information see “Define Input Properties by Example at the Command Line” (MATLAB Coder). Since the inputs to `% visionRecoverformCodeGeneration_kernel` are a pair of images, we define both the inputs with the following properties:

- variable-sized at run-time with upper-bound [1000 1000]
- data type `uint8`

```
% Define the properties of input images
```

```
imageTypeAndSize = coder.typeof(uint8(0), [1000 1000],[true true]);
compileTimeInputs = {imageTypeAndSize, imageTypeAndSize};
```

```
codegen visionRecoverformCodeGeneration_kernel.m -report -args compileTimeInputs;
```

```
Code generation successful: To view the report, open('codegen\mex\visionRecoverformCodeGenerati
```

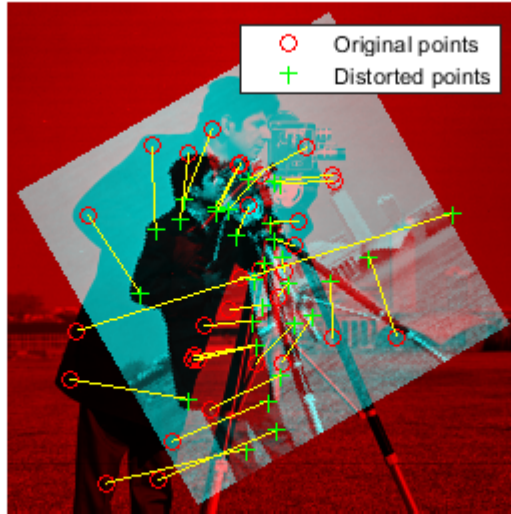
Run the Generated Code

```
[matchedOriginalLocCG, matchedDistortedLocCG, ...
 thetaRecoveredCG, scaleRecoveredCG, recoveredCG] = ...
 visionRecoverformCodeGeneration_kernel_mex(original, distorted);
```

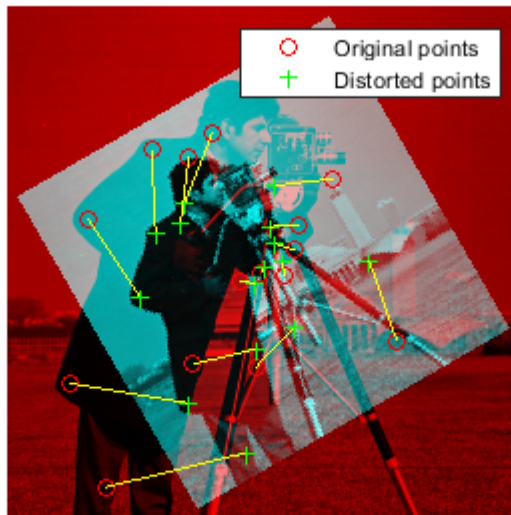
```
scaleRecovered = 0.701003
```

```
thetaRecovered = 30.235065
```

Putatively matched points (including outliers)



Matching points (inliers only)





Clean Up

```
clear visionRecoverFromCodeGeneration_kernel_mex;
```

Compare Codegen with MATLAB Code

Recovered scale and theta for both MATLAB and CODEGEN, as shown above, are within reasonable tolerance. Furthermore, the matched points are identical, as shown below:

```
isequal(matchedOriginalLocCG, matchedOriginalLoc)
isequal(matchedDistortedLocCG, matchedDistortedLoc)
```

```
ans =
    logical
     1
```

```
ans =
    logical
     1
```

Appendix

The following helper functions are used in this example.

- `featureMatchingVisualization_extrinsic`

Code Generation for Face Tracking with PackNGo

This example shows how to generate code from Face Detection and Tracking Using the KLT Algorithm example with packNGo function. The packNGo function packages all relevant files in a compressed zip file so you can relocate, unpack, and rebuild your project in another development environment without MATLAB present. This example also shows how to create a makefile for the packNGo content, rebuild the source files and finally run the standalone executable outside MATLAB environment.

This example requires a MATLAB® Coder™ license.

This example is a function with the main body at the top and helper routines in the form of nested functions below.

```
function FaceTrackingKLTpackNGoExample()
```

Set Up Your C++ Compiler

To run this example, you must have access to a C++ compiler and you must configure it using 'mex - setup c++' command. For more information, see Choose a C++ Compiler. If you deploy the application on MATLAB host, use a C++ compiler that is compatible with the compiler used to build OpenCV libraries. For more information, see Compiler used to build OpenCV libraries.

Break Out the Computational Part of the Algorithm into a Separate MATLAB Function

MATLAB Coder requires MATLAB code to be in the form of a function in order to generate C code. The code for the main algorithm of this example resides in a function called FaceTrackingKLTpackNGo_kernel.m. This file is derived from Face Detection and Tracking Using the KLT Algorithm. To learn how to modify the MATLAB code to make it compatible for code generation, you can look at example Introduction to Code Generation with Feature Matching and Registration

```
fileName = 'FaceTrackingKLTpackNGo_kernel.m';
visiondemo_dir = pwd;
currentDir = pwd; % Store the current directory
fileName = fullfile(visiondemo_dir, fileName);
```

Configure Code Generation Arguments for packNGo

Create a code generation configuration object for EXE output with packNGo function call in post code generation stage.

```
codegenArgs = createCodegenArgs(visiondemo_dir);
```

Setup Code Generation Environment

Change output directory name.

```
codegenOutDir = fullfile(visiondemo_dir, 'codegen');
mkdir(codegenOutDir);
```

Add path to the existing directory to have access to necessary files.

```
currentPath = addpath(visiondemo_dir);
pathCleanup = onCleanup(@()path(currentPath));
cd(codegenOutDir);
dirChange = onCleanup(@()cd(currentDir));
```

Create the Packaged Zip-file

Invoke codegen command with packNGo function call.

```
fprintf('-> Generating Code (it may take a few minutes) ....\n');
codegen(codegenArgs{:}, fileName);

-> Generating Code (it may take a few minutes) ....
```

Note that, instead of using codegen command, you can open a dialog and launch a code generation project using coder. Use the post code generation command with packNGo function to create a zip file.

Build Standalone Executable

Unzip the zip file into a new folder. Note that the zip file contains source files, header files, libraries, MAT-file containing the build information object, data files. unzipPackageContents and other helper functions are included in the appendix.

```
zipFileLocation = codegenOutDir;
fprintf('-> Unzipping files ....\n');
unzipFolderLocation = unzipPackageContents(zipFileLocation);

-> Unzipping files ....
```

Create platform dependent makefile from a template makefile.

```
fprintf('-> Creating makefile ....\n');
[~, fname, ~] = fileparts(fileName);
makefileName = createMakeFile(visiondemo_dir, unzipFolderLocation, fname);

-> Creating makefile ....
```

Create the commands required to build the project and to run it.

```
fprintf('-> Creating 'Build Command' and 'Run command' ....\n');
[buildCommand, runCommand] = createBuildAndRunCommands(zipFileLocation,...
    unzipFolderLocation,makefileName,fname);

-> Creating 'Build Command' and 'Run command' ....
```

Build the project using build command.

```
fprintf('-> Building executable....\n');
buildExecutable(unzipFolderLocation, buildCommand);

-> Building executable....
```

Run the Executable and Deploy

Run the executable and verify that it works.

```
cd(unzipFolderLocation);
system(runCommand);
```

The application can be deployed in another machine by copying the executable and the library files.

```
isPublishing = ~isempty(snapnow('get'));
if ~isPublishing % skip printing out directory to html page
```

```

fprintf('Executable and library files are located in the following folder:\n%s\n', unzipFolder);
fprintf('To re-execute run the following commands:\n');
fprintf('1. cd('%s')\n', unzipFolderLocation);
fprintf('2. system('%s')\n', runCommand);
end

```

Appendix - Helper Functions

```

% Configure coder to create executable. Use packNGo at post code
% generation stage.
function codegenArgs = createCodegenArgs(folderForMainC)
    % Create arguments required for code generation.

    % For standalone executable a main C function is required. The main.c
    % created for this example is compatible with the content of the file
    % visionFaceTrackingKLTpackNGo_kernel.m
    mainCFile = fullfile(folderForMainC, 'main.c');

    % Handle path with space
    if contains(mainCFile, ' ')
        mainCFile = ['\" mainCFile '\"'];
    end

    cfg                                = coder.config('exe');
    cfg.PostCodeGenCommand              = 'packNGo(buildInfo, 'packType', 'hierarchical');';
    cfg.CustomSource                    = mainCFile;
    cfg.CustomInclude                   = folderForMainC;
    cfg.EnableOpenMP                    = false;

    codegenArgs = {'-config', cfg};
end

% Create a folder and unzip the packNGo content into it.
function unzipFolderLocation = unzipPackageContents(zipFileLocation)
    % Unzip the packaged zip file.

    unzipFolderLocationName = 'unzipPackNGo';
    mkdir(unzipFolderLocationName);

    % Get the name of the zip file generated by packNGo.
    zipFile = dir('*.zip');

    assert(numel(zipFile)==1);

    unzip(zipFile.name, unzipFolderLocationName);

    % Unzip internal zip files created in hierarchical packNGo.
    zipFileInternal = dir(fullfile(unzipFolderLocationName, '*.zip'));
    assert(numel(zipFileInternal)==3);

    for i=1:numel(zipFileInternal)
        unzip(fullfile(unzipFolderLocationName, zipFileInternal(i).name), ...
            unzipFolderLocationName);
    end

    unzipFolderLocation = fullfile(zipFileLocation, unzipFolderLocationName);
end

```



```

% Create platform dependent makefile from template makefile. Use
% buildInfo to get info about toolchain.
function makefileName = createMakeFile(visiondemo_dir, unzipFolderLocation, fname)
    % Create Makefile from buildInfo.

    binfo = load(fullfile(pwd, 'codegen', 'exe', fname, 'buildInfo.mat'));

    lastDir    = cd(unzipFolderLocation);
    dirCleanup = onCleanup(@()cd(lastDir));

    % Get the root directory that contains toolbox/vision sub-directories
    matlabDirName = getRootDirName(unzipFolderLocation);

    % Get defines
    horzcat_with_space = @(cellval)sprintf('%s ',cellval{:});
    defs    = horzcat_with_space(getDefines(binInfo.buildInfo));

    % Get source file list
    if ispc
        [~, cFiles] = system(['dir /s/b ' '*.c']);
        [~, cppFiles] = system(['dir /s/b ' '*.cpp']);
    else
        [~, cFiles] = system(['find ./ ' '-name ' '*.c']);
        [~, cppFiles] = system(['find ./ ' '-name ' '*.cpp']);
    end

    cIndx = strfind(cFiles, '.c');
    cppIndx = strfind(cppFiles, '.cpp');
    srcFilesC = [];
    srcFilesCPP = [];

    for i = 1:length(cIndx)
        if i == 1
            startIdx = 1;
            endIdx = cIndx(i);
        else
            startIdx = cIndx(i-1)+1;
            endIdx = cIndx(i);
        end

        [~, b, ~] = fileparts(cFiles(startIdx:endIdx));
        srcFilesC = [srcFilesC ' ' b '.c']; %#ok<AGROW>
    end

    for i = 1:length(cppIndx)
        if i == 1
            startIdx = 1;
            endIdx = cppIndx(i);
        else
            startIdx = cppIndx(i-1)+1;
            endIdx = cppIndx(i);
        end

        [~, b, ~] = fileparts(cppFiles(startIdx:endIdx));
        srcFilesCPP = [srcFilesCPP ' ' b '.cpp']; %#ok<AGROW>
    end

```

```

end

srcFiles = [srcFilesC ' ' srcFilesCPP];

% Get platform dependent names
if isunix % both mac and linux
    tmf = 'TemplateMakefilePackNGo_unix';
    if ismac
        archDir = 'maci64';
        dllExt = 'dylib';
    else
        archDir = 'glnxa64';
        dllExt = 'so';
    end
end
else
    tmf = 'TemplateMakefilePackNGo_win';
    archDir = 'win64';
    dllExt = 'dll';
end

% Now that we have defines, lets create a platform dependent makefile
% from template.
fid = fopen(fullfile(visiondemo_dir,tmf));

filecontent = char(fread(fid));
fclose(fid);

newfilecontent = regexprep(filecontent,...
    {'PASTE_ARCH','PASTE_EXT','PASTE_DEFINES','PASTE_SRCFILES','PASTE_MATLAB'},...
    { archDir,      dllExt,      defs,          srcFiles,      matlabDirName});

makefileName = 'Makefile';
mk_name = fullfile(unzipFolderLocation,makefileName);

if isunix
    if( ismac )
        [status,sysHeaderPath] = system( 'xcode-select -print-path' );
        assert(status==0, ['Could not obtain a path to the system ' ...
            'header files using 'xcode-select -print-path' ' ']);

        [status,sdkPaths] = system( [ 'find ' deblank( sysHeaderPath ) ...
            ' -name 'MacOSX*.sdk' ' ' ] );
        assert(status==0, 'Could not find MacOSX sdk' );

        % There might be multiple SDK's
        sdkPathCell = strsplit(sdkPaths,'\n');
        for idx = 1:numel(sdkPathCell)
            if ~isempty(sdkPathCell{idx})
                % Pick the first one that's not empty.
                sdkPath = sdkPathCell{idx};
                fprintf('Choosing SDK in %s\n',sdkPath);
                break;
            end
        end
        assert(~isempty(sdkPath), ...
            sprintf('There is no sdk available in %s. Please check system environment.\n',
                ccCMD = [ 'xcrun clang -isysroot ' deblank( sdkPath ) ];

```

```

        cppCMD = [ 'xcrun clang++ -isysroot ' deblank( sdkPath ) ];
    else
        ccCMD = 'gcc';
        cppCMD = 'g++';
    end

    newfilecontent = regexprep(newfilecontent, 'PASTE_CC', ccCMD);
    newfilecontent = regexprep(newfilecontent, 'PASTE_CPP', cppCMD);
end

fid = fopen(mk_name, 'w+');
fprintf(fid, '%s', newfilecontent);
fclose(fid);

end

% Create platform specific commands needed to build the executable and
% to run it.
function [buildCommand, runCommand] = createBuildAndRunCommands( ...
    packageLocation, unzipFolderLocation, makefileName, fileName)
% Create the build and run command.

if ismac
    buildCommand = [ ' xcrun make -f ' makefileName];
    runCommand = ['./' fileName ' "' fileName '"];
elseif isunix
    buildCommand = [ ' make -f ' makefileName];
    runCommand = ['./' fileName ' "' fileName '"];
else
    % On PC we use the generated BAT files (there should be 2) to help
    % build the generated code. These files are copied to the
    % unzipFolderLocation where we can use them to build.
    batFilename = [fileName '_rtw.bat'];
    batFilelocation = fullfile(packageLocation, 'codegen', ...
        filesep, 'exe', filesep, fileName);
    batFileDestination = unzipFolderLocation;

    % For MSVC, also copy 'setup_msvc.bat'
    fid = fopen(fullfile(batFilelocation, batFilename));
    batFileContent = fread(fid, '*char');
    fclose(fid);
    if ~isempty(regexprep(convertCharsToStrings(batFileContent), 'setup_msvc.bat', 'once'))
        setup_msvc_batFile = fullfile(batFilelocation, 'setup_msvc.bat');
        copyfile(setup_msvc_batFile, batFileDestination);
    end

    % Copy it to packNGo output directory.
    copyfile(fullfile(batFilelocation, batFilename), batFileDestination);

    % The Makefile we created is named 'Makefile', whereas the Batch
    % file refers to <filename>_rtw.mk. Hence we rename the file.
    newMakefileName = [fileName '_rtw.mk'];
    oldMakefilename = makefileName;
    copyfile(fullfile(batFileDestination, oldMakefilename), ...
        fullfile(batFileDestination, newMakefileName));

    buildCommand = batFilename;
    runCommand = [fileName '.exe' ' "' fileName '"];
end

```

```
    end

end

% Build the executable with the build command.
function buildExecutable(unzipFolderLocation, buildCommand)
    % Call system command to build the executable.

    lastDir    = cd(unzipFolderLocation);
    dirCleanup = onCleanup(@()cd(lastDir));

    [hadError, sysResults] = system(buildCommand);

    if hadError
        error (sysResults);
    end

end

% Get the root directory that contains toolbox/vision sub-directories
function matlabDirName = getRootDirName(unzipFolderName)
    dirLists = dir(unzipFolderName);
    dirLists = dirLists(~ismember({dirLists.name},{'.','..'}));

    matlabDirName='';
    for ij=1:length(dirLists)
        thisDirName = dirLists(ij).name;
        if (isfolder(thisDirName))
            % subdirectory will have toolbox/vision
            [subDir1, hasSubDir1] = hasSubdirectory(thisDirName, 'toolbox');
            if hasSubDir1
                [~, hasSubDir2] = hasSubdirectory(subDir1, 'vision');
                if hasSubDir2
                    matlabDirName = thisDirName;
                    break;
                end
            end
        end
    end
end

% Find the directory that contains the specified sub-directory
function [subDir, hasSubDir] = hasSubdirectory(dirName, subDirName)
    dirLists = dir(dirName);
    dirLists = dirLists(~ismember({dirLists.name},{'.','..'}));

    subDir = '';
    hasSubDir = false;

    for ij=1:length(dirLists)
        thisDirName = dirLists(ij).name;
        thisDir = fullfile(dirName,thisDirName);

        if (isfolder(thisDir) && strcmp(thisDirName, subDirName))
            hasSubDir = true;
            subDir = thisDir;
            break;
        end
    end
end
```

```
    end  
  end  
end
```

Code Generation for Depth Estimation From Stereo Video

This example shows how to use the MATLAB® Coder™ to generate C code for a MATLAB function, which uses the `stereoParameters` object produced by Stereo Camera Calibrator app or the `estimateCameraParameters` function. The example explains how to modify the MATLAB code in the “Depth Estimation From Stereo Video” on page 1-61 example to support code generation.

This example requires a MATLAB Coder license.

Code Generation

You can learn about the basics of code generation using the MATLAB® Coder™ from the “Introduction to Code Generation with Feature Matching and Registration” on page 2-8 example.

Restructuring the MATLAB Code for C Code Generation

MATLAB Coder requires MATLAB code to be in the form of a function in order to generate C code. Furthermore, the arguments of the function cannot be MATLAB objects.

This presents a problem for generating code from MATLAB code, which uses `cameraParameters` or `stereoParameters` objects, which are typically created in advance during camera calibration. To solve this problem, use the `toStruct()` method to convert the `cameraParameters` or the `stereoParameters` object into a struct. The struct can then be passed into the generated code.

The restructured code for the main algorithm of “Depth Estimation From Stereo Video” on page 1-61 example resides in a function called `depthEstimationFromStereoVideo_kernel.m`. Note that `depthEstimationFromStereoVideo_kernel` is a function that takes a struct created from a `stereoParameters` object. Note also that it does not display the reconstructed 3-D point cloud, because the `showPointCloudFunction` does not support code generation.

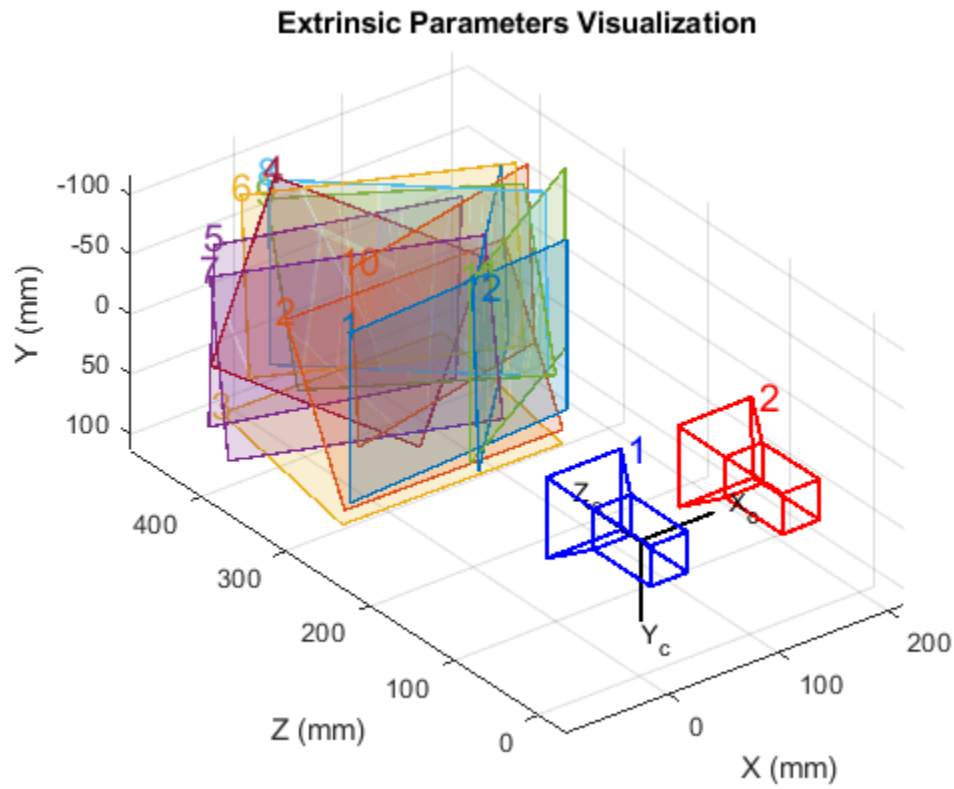
Load the Parameters of the Stereo Camera

Load the `stereoParameters` object, which is the result of calibrating the camera using either the `stereoCameraCalibrator` app or the `estimateCameraParameters` function.

```
% Load the stereoParameters object.
load('handshakeStereoParams.mat');

% Visualize camera extrinsics.
showExtrinsics(stereoParams);

% Convert the object into a struct, which can be passed into generated
% code.
stereoParamsStruct = toStruct(stereoParams);
```



Uncompress Video Files

On Macintosh, VideoReader does not support code generation for reading compressed video. Uncompress the video files, and store them in the temporary directory.

```

if strcmp(computer(), 'MACI64')
    % Uncompress the left video.
    videoFileLeft = 'handshake_left.avi';
    reader = VideoReader(videoFileLeft);
    writer = vision.VideoFileWriter(videoFileLeft);
    while hasFrame(reader)
        frame = readFrame(reader);
        step(writer, frame);
    end
    release(reader);
    release(writer);

    % Uncompress the right video.
    videoFileRight = 'handshake_right.avi';
    reader = VideoReader(videoFileRight);
    writer = vision.VideoFileWriter(videoFileRight);
    while hasFrame(reader)
        frame = readFrame(reader);
        step(writer, frame);
    end
    release(reader);

```

```
        release(writer);  
end
```

Compile the MATLAB Function Into a MEX File

Use the `codegen` function to compile the `depthEstimationFromStereoVideo_kernel` function into a MEX-file. You can specify the `'-report'` option to generate a compilation report that shows the original MATLAB code and the associated files that were created during C code generation. You may want to create a temporary directory where MATLAB Coder can store generated files. Note that the generated MEX-file has the same name as the original MATLAB file with `_mex` appended, unless you use the `-o` option to specify the name of the executable.

MATLAB Coder requires that you specify the properties of all the input parameters. One easy way to do this is to define the input properties by example at the command-line using the `-args` option. For more information see “Define Input Properties by Example at the Command Line” (MATLAB Coder).

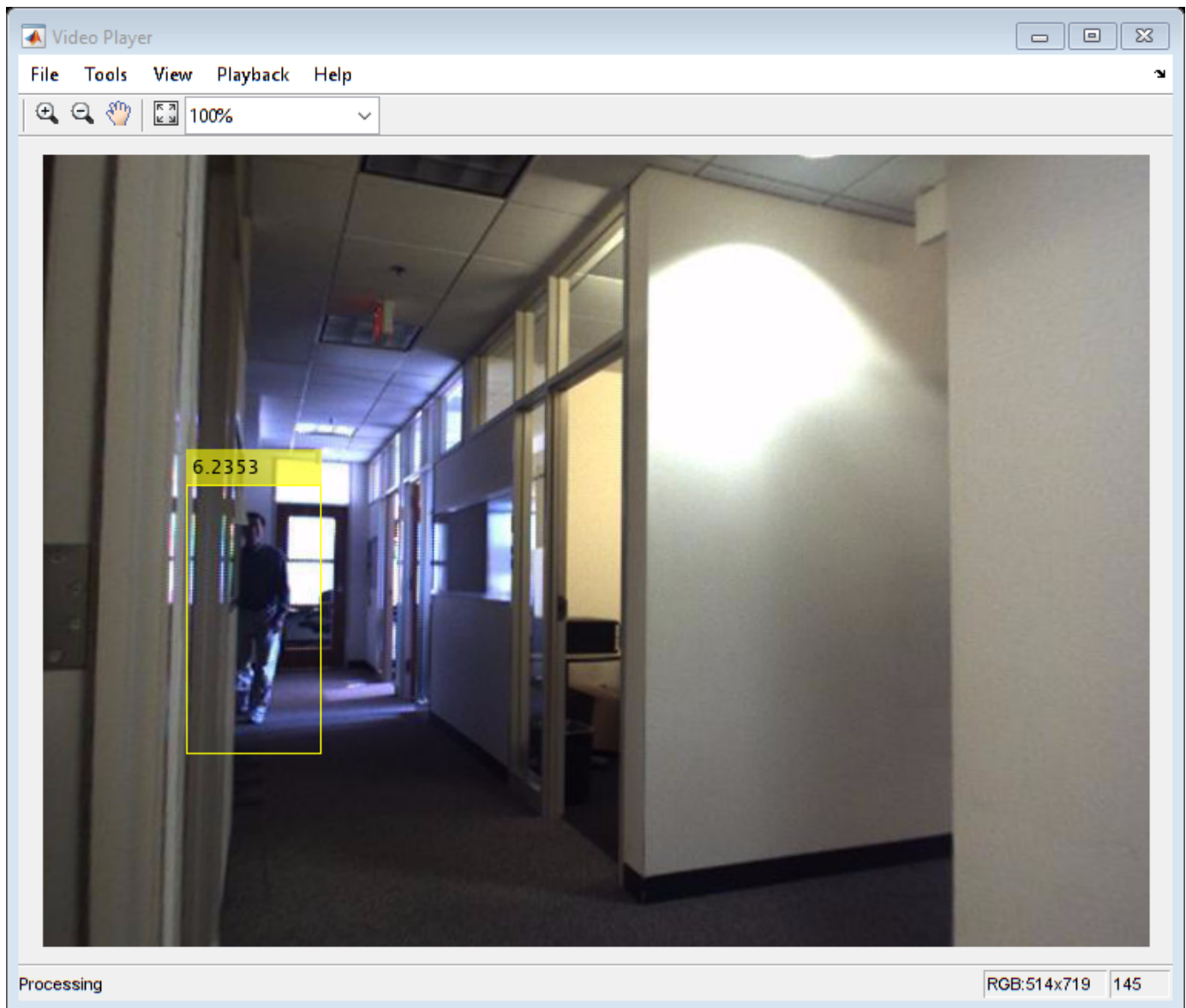
```
compileTimeInputs = {coder.typeof(stereoParamsStruct)};
```

```
% Generate code.
```

```
codegen depthEstimationFromStereoVideo_kernel -args compileTimeInputs;
```

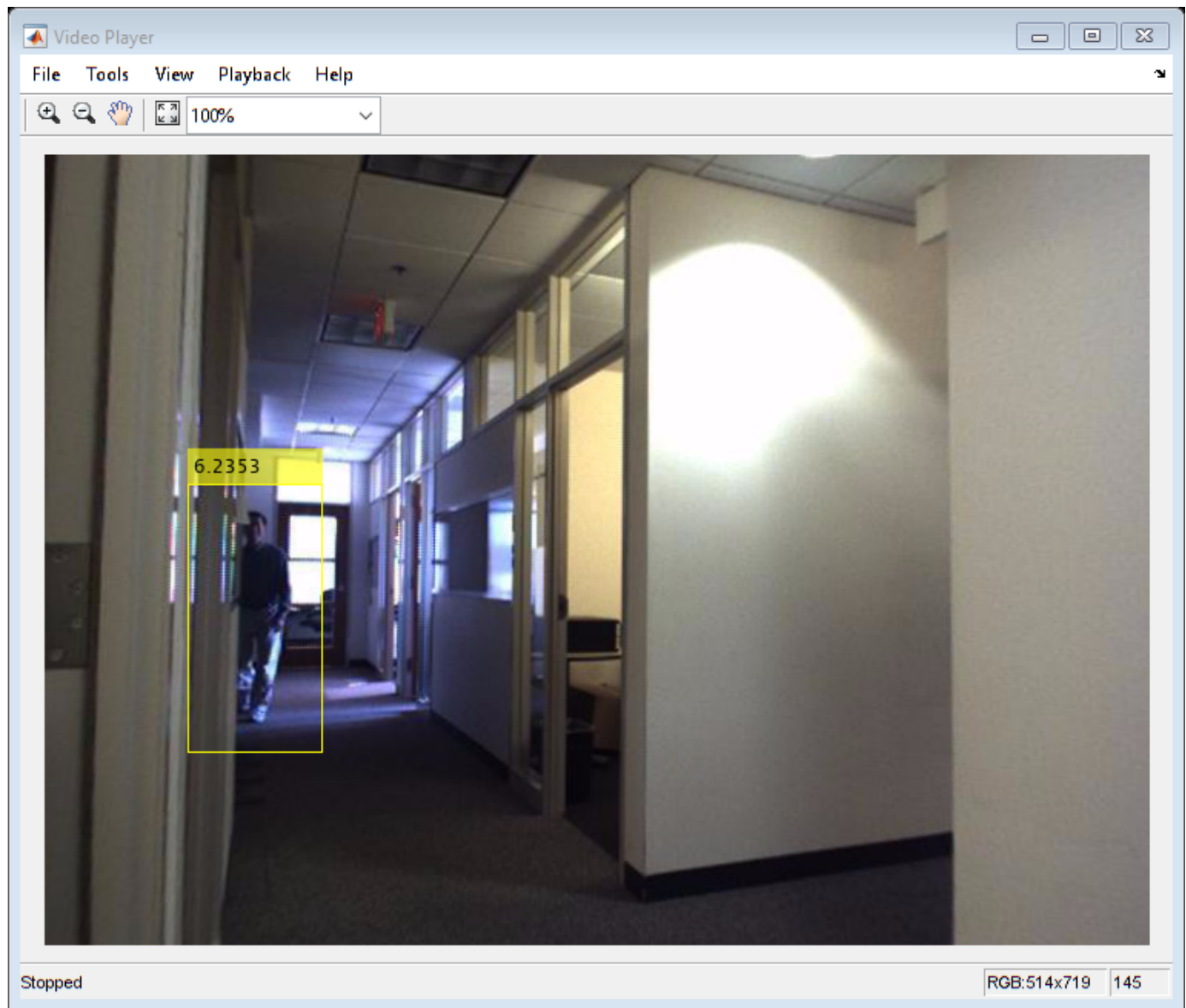
Run the Generated Code

```
player = vision.VideoPlayer('Position', [100 200 750 560]);  
eofReached = false;  
while ~eofReached  
    [eofReached, dispFrame] = depthEstimationFromStereoVideo_kernel_mex(stereoParamsStruct);  
  
    % Hold the last frame.  
    if ~eofReached  
        step(player, dispFrame);  
    end  
end
```

Clean Up

```
clear depthEstimationFromStereoVideo_kernel_mex;  
release(player);
```



Summary

This example showed how to generate C code from MATLAB code that takes a `cameraParameters` or a `stereoParameters` object as input.

Detect Face (Raspberry Pi2)

This example shows how to use the MATLAB® Coder™ to generate C code from a MATLAB file and deploy the application on an ARM target.

The example reads video frames from a webcam and detects faces in each of the frames using the Viola-Jones face detection algorithm. The detected faces are displayed with bounding boxes. The webcam function, from 'MATLAB Support Package for USB Webcams', and the VideoPlayer object, from the Computer Vision System toolbox™, are used for the simulation on the MATLAB host. The two functions do not support the ARM target, so OpenCV-based webcam reader and video viewer functions are used for deployment.

The target must have OpenCV version 3.4.0 libraries (built with GTK) and a standard C++ compiler. A Raspberry Pi 2 with Raspbian Stretch operating system was used for deployment. The example should work on any ARM target.

This example requires a MATLAB Coder license.

This example is a function with the main body at the top and helper routines in the form of nested functions below.

```
function FaceDetectionARMCodeGenerationExample()
```

Set Up Your C++ Compiler

To run this example, you must have access to a C++ compiler and you must configure it using 'mex -setup c++' command. For more information, see [Choose a C++ Compiler](#).

Break Out the Computational Part of the Algorithm into a Separate MATLAB Function

MATLAB Coder requires MATLAB code to be in the form of a function in order to generate C code. The code for the main algorithm of this example resides in a function called `faceDetectionARMKernel.m`. The function takes an image from a webcam, as the input. The function outputs the image with a bounding box around the detected faces. The output image will be displayed on video viewer window. To learn how to modify the MATLAB code to make it compatible for code generation, you can look at example [Introduction to Code Generation with Feature Matching and Registration](#)

```
fileName = 'faceDetectionARMKernel.m';
```

Create Main Function with I/O Functionality

For a standalone executable target, MATLAB Coder requires that you create a C file containing a function named "main". This example uses `faceDetectionARMMain.c` file. This main function in this file performs the following tasks:

- Reads video frames from the webcam
- Sends video frames to the face detection algorithm
- Displays output frames containing bounding boxes around detected faces

For simulation on MATLAB host, the tasks performed in `faceDetectionARMMain.c` file is implemented in `faceDetectionARMMain.m`

Webcam Reader and Video Viewer

For deployment on ARM, this example implements webcam reader functionality using OpenCV functions. It also implements a video viewer using OpenCV functions. These OpenCV based utility functions are implemented in the following files:

- helperOpenCVWebcam.hpp
- helperOpenCVWebcam.cpp
- helperOpenCVVideoViewer.cpp
- helperOpenCVVideoViewer.hpp

For simulation on MATLAB host, the example uses the webcam function from the 'MATLAB Support Package for USB Webcams' and the VideoPlayer object from the Computer Vision System toolbox. Run the simulation on the MATLAB host by typing faceDetectionARMMain at the MATLAB® command line.

OpenCV for ARM Target

This example requires that you install OpenCV 3.4.0 libraries on your ARM target. The video viewer requires that you build the highgui library in OpenCV with GTK for the ARM target.

Follow the steps to download and build OpenCV 3.4.0 on Raspberry Pi 2 with preinstalled Raspbian Stretch. You must update your system firmware or install other developer tools and packages as needed for your system configuration before you start building OpenCV.

Turn off `INSTALL_C_EXAMPLES` due to: <https://github.com/opencv/opencv/issues/5851>

Turn off `ENABLE_PRECOMPILED_HEADERS` due to: <https://github.com/opencv/opencv/issues/9942>

- `$ wget -O opencv-3.4.0.zip https://github.com/opencv/opencv/archive/3.4.0.zip`
- `$ unzip opencv-3.4.0.zip`
- `$ cd opencv-3.4.0`
- `$ mkdir build`
- `$ cd build`
- `$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D INSTALL_C_EXAMPLES=OFF -D BUILD_EXAMPLES=ON -D WITH_GTK=ON -D WITH_FFMPEG=OFF -D ENABLE_PRECOMPILED_HEADERS=OFF ..`

These steps are followed to compile and install OpenCV:

- `$ make`
- `$ sudo make install`

For official deployment of the example, OpenCV libraries were installed in the following directory on Raspberry Pi 2:

`/usr/local/lib`

and the associated headers were placed in

`/usr/local/include`

Configure Code Generation Arguments

Create a code generation configuration object for EXE output.

```
codegenArgs = createCodegenArgs();
```

Generate Code

Invoke codegen command.

```
fprintf('-> Generating Code (it may take a few minutes) ....\n');
codegen(codegenArgs{:}, fileName);
% During code generation, all dependent file information is stored in a mat
% file named buildInfo.mat.
```

```
-> Generating Code (it may take a few minutes) ....
```

Create the Packaged Zip-file

Use build information stored in buildInfo.mat to create a zip folder using packNGo.

```
fprintf('-> Creating zip folder (it may take a few minutes) ....\n');
bInfo = load(fullfile('codegen','exe','faceDetectionARMKernel','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'packType', 'hierarchical', ...
    'fileName', 'faceDetectionARMKernel'});
% The generated zip folder is faceDetectionARMKernel.zip
```

```
-> Creating zip folder (it may take a few minutes) ....
```

Create Project Folder

Unzip faceDetectionARMKernel.zip into a folder named FaceDetectionARM. Unzip all files and remove the .zip files.

```
packngoDir = hUnzipPackageContents();
```

Update Makefile and Copy to Project Folder

The Makefile, faceDetectionARMMakefile.mk, provided in this example is written for Raspberry PI 2 with specific optimization flags. The Makefile was written to work with GCC in a Linux environment and with your OpenCV libraries located in /usr/local/lib. You can update the Makefile based on your target configuration. Copy the Makefile to the project folder.

```
copyfile('faceDetectionARMMakefile.mk', packngoDir);
% Also move the file containing the main function in the top level folder.
copyfile('faceDetectionARMMain.c', packngoDir);
% For simplicity, make sure the root directory name is matlab.
setRootDirectory(packngoDir);
```

Deployment on ARM

Deploy your project on ARM:

```
disp('Follow these steps to deploy your project on ARM');
```

Follow these steps to deploy your project on ARM

Transfer Code to ARM Target

Transfer your project folder named FaceDetectionARM to your ARM target using your preferred file transfer tool. Since the Raspberry Pi 2 (with Raspbian Stretch) already has an SSH server, you can use SFTP to transfer files from host to target.

For official deployment of this example, the FileZilla SFTP Client was installed on the host machine and the project folder was transferred from the host to the `/home/pi/FaceDetectionARM` folder on Raspberry Pi.

```
disp('Step-1: Transfer the folder 'FaceDetectionARM' to your ARM target');
```

```
Step-1: Transfer the folder 'FaceDetectionARM' to your ARM target
```

Build the Executable on ARM

Run the makefile to build the executable on ARM. For Raspberry Pi 2, (with Raspbian Stretch), open a linux shell and cd to `/home/pi/FaceDetectionARM`. Build the executable using the following command:

```
make -f faceDetectionARMMakefile
```

The command creates an executable, `faceDetectionARMKernel`.

```
disp('Step-2: Build the executable on ARM using the shell command: make -f faceDetectionARMMakefile');
```

```
Step-2: Build the executable on ARM using the shell command: make -f faceDetectionARMMakefile.mk
```

Run the Executable on ARM

Run the executable generated in the above step. For Raspberry Pi 2, (with Raspbian Stretch), use the following command in the shell window:

```
./faceDetectionARMKernel
```

Make sure that you are connected to the Raspberry Pi with a window manager, and not just through a command line terminal to avoid errors related to GTK. This is necessary for the tracking window to show up.

To close the video viewer while the executable is running on Raspberry Pi2, click on the video viewer and press the escape key.

```
disp('Step-3: Run the executable on ARM using the shell command: ./faceDetectionARMKernel');
```

```
Step-3: Run the executable on ARM using the shell command: ./faceDetectionARMKernel
```

Appendix - Helper Functions

```
% Configure coder to create executable. Use packNGo at post code
% generation stage.
function codegenArgs = createCodegenArgs()
    % Create arguments required for code generation.

    % First - create configuration object
    %
    % For standalone executable a main C function is required. The
    % faceDetectionARMMain.c created for this example is compatible
    % with the content of the file faceDetectionARMKernel.m
    mainCFile = 'faceDetectionARMMain.c';
```

```

% Include helper functions
camCPPFile = 'helperOpenCVWebcam.cpp';
viewerCPPFile = 'helperOpenCVVideoViewer.cpp';

% Handle path with space
if contains(mainCFile, ' ')
    mainCFile    = ['\" mainCFile '\"'];
    camCPPFile   = ['\" camCPPFile '\"'];
    viewerCPPFile = ['\" viewerCPPFile '\"'];
end

% Create configuration object
cfg = coder.config('exe');
cfg.CustomSource      = sprintf('%s\n%s\n%s',mainCFile,camCPPFile,viewerCPPFile);
cfg.CustomInclude     = pwd;
% Set production hardware to ARM to generate ARM compatible portable code
cfg.HardwareImplementation.ProdHWDeviceType = 'ARM Compatible->ARM Cortex';
cfg.EnableOpenMP      = false;

% Create input arguments
inRGB_type = coder.typeof(uint8(0),[480 640 3]);
% Use '-c' option to generate C code without calling C++ compiler.
codegenArgs = {'-config', cfg, '-c', '-args', {inRGB_type}};

end

% Unzip the packaged zip file
function packngoDir = hUnzipPackageContents()

    packngoDirName = 'FaceDetectionARM';

    % create packngo directory
    mkdir(packngoDirName);

    % get the name of the single zip file generated by packngo
    zipFile = dir('*.zip');
    assert(numel(zipFile)==1);

    unzip(zipFile.name,packngoDirName);

    % unzip internal zip files created in hierarchical packNGo
    zipFileInternal = dir(fullfile(packngoDirName,'*.zip'));

    for i=1:numel(zipFileInternal)
        unzip(fullfile(packngoDirName,zipFileInternal(i).name), ...
            packngoDirName);
    end
    % delete internal zip files
    delete(fullfile(packngoDirName,'*.zip'));
    packngoDir = packngoDirName;
end

% Set root directory as matlab
function setRootDirectory(packngoDir)
    dirList = dir(packngoDir);
    if isempty(find(ismember({dirList.name},'matlab'), 1))
        % root directory is not matlab. Change it to matlab
    end
end

```

```
    for i=1:length(dirList)
        thisDir = fullfile(packngoDir,dirList(i).name, 'toolbox', 'vision');
        if isfolder(thisDir)
            % rename the dir
            movefile(fullfile(packngoDir,dirList(i).name), ...
                fullfile(packngoDir,'matlab'));
            break;
        end
    end
end
end
end
end
```


Track Face (Raspberry Pi2)

This example shows how to use the MATLAB® Coder™ to generate C code from a MATLAB file and deploy the application on ARM target.

The example reads video frames from a webcam. It detects a face using Viola-Jones face detection algorithm and tracks the face in a live video stream using the KLT algorithm. It finally displays the frame with a bounding box and a set of markers around the face being tracked. The webcam function, from 'MATLAB Support Package for USB Webcams', and the VideoPlayer object, from the Computer Vision System toolbox™, are used for the simulation on the MATLAB host. The two functions do not support the ARM target, so OpenCV-based webcam reader and video viewer functions are used for deployment.

The target must have OpenCV version 3.4.0 libraries (built with GTK) and a standard C++ compiler. A Raspberry Pi 2 with Raspbian Stretch operating system was used for deployment. The example should work on any ARM target.

This example requires a MATLAB Coder license.

This example is a function with the main body at the top and helper routines in the form of nested functions below.

```
function FaceTrackingARMCodeGenerationExample()
```

Set Up Your C++ Compiler

To run this example, you must have access to a C++ compiler and you must configure it using 'mex -setup c++' command. For more information, see Choose a C++ Compiler.

Break Out the Computational Part of the Algorithm into a Separate MATLAB Function

MATLAB Coder requires MATLAB code to be in the form of a function in order to generate C code. The code for the main algorithm of this example resides in a function called `faceTrackingARMKernel.m`. The function takes an image from a webcam, as the input. The function outputs the image with a bounding box and a set of markers around the face. The output image will be displayed on video viewer window. To learn how to modify the MATLAB code to make it compatible for code generation, you can look at example Introduction to Code Generation with Feature Matching and Registration

```
fileName = 'faceTrackingARMKernel.m';
```

Create Main Function with I/O Functionality

For a standalone executable target, MATLAB Coder requires that you create a C file containing a function named "main". This example uses `faceTrackingARMMain.c` file. This main function in this file performs the following tasks:

- Reads video frames from the webcam
- Sends video frames to the face tracking algorithm
- Displays output frames containing bounding box and markers around the face

For simulation on MATLAB host, the tasks performed in `faceTrackingARMMain.c` file is implemented in `faceTrackingARMMain.m`

Webcam Reader and Video Viewer

For deployment on ARM, this example implements webcam reader functionality using OpenCV functions. It also implements a video viewer using OpenCV functions. These OpenCV based utility functions are implemented in the following files:

- helperOpenCVWebcam.hpp
- helperOpenCVWebcam.cpp
- helperOpenCVVideoViewer.cpp
- helperOpenCVVideoViewer.hpp

For simulation on MATLAB host, the example uses the webcam function from the 'MATLAB Support Package for USB Webcams' and the VideoPlayer object from the Computer Vision System toolbox. Run the simulation on the MATLAB host by typing faceTrackingARMMain at the MATLAB® command line.

OpenCV for ARM Target

This example requires that you install OpenCV 3.4.0 libraries on your ARM target. The video viewer requires that you build the highgui library in OpenCV with GTK for the ARM target.

Follow the steps to download and build OpenCV 3.4.0 on Raspberry Pi 2 with preinstalled Raspbian Stretch. You must update your system firmware or install other developer tools and packages as needed for your system configuration before you start building OpenCV.

Turn off `INSTALL_C_EXAMPLES` due to: <https://github.com/opencv/opencv/issues/5851>

Turn off `ENABLE_PRECOMPILED_HEADERS` due to: <https://github.com/opencv/opencv/issues/9942>

- `$ wget -O opencv-3.4.0.zip https://github.com/opencv/opencv/archive/3.4.0.zip`
- `$ unzip opencv-3.4.0.zip`
- `$ cd opencv-3.4.0`
- `$ mkdir build`
- `$ cd build`
- `$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D INSTALL_C_EXAMPLES=OFF -D BUILD_EXAMPLES=ON -D WITH_GTK=ON -D WITH_FFMPEG=OFF -D ENABLE_PRECOMPILED_HEADERS=OFF ..`

These steps are followed to compile and install OpenCV:

- `$ make`
- `$ sudo make install`

For official deployment of the example, OpenCV libraries were installed in the following directory on Raspberry Pi 2:

```
/usr/local/lib
```

and the associated headers were placed in

```
/usr/local/include
```

Configure Code Generation Arguments

Create a code generation configuration object for EXE output.

```
codegenArgs = createCodegenArgs();
```

Generate Code

Invoke codegen command.

```
fprintf('-> Generating Code (it may take a few minutes) ....\n');
codegen(codegenArgs{:}, fileName);
% During code generation, all dependent file information is stored in a mat
% file named buildInfo.mat.
```

```
-> Generating Code (it may take a few minutes) ....
```

Create the Packaged Zip-file

Use build information stored in buildInfo.mat to create a zip folder using packNGo.

```
fprintf('-> Creating zip folder (it may take a few minutes) ....\n');
bInfo = load(fullfile('codegen','exe','faceTrackingARMKernel','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'packType', 'hierarchical', ...
                        'fileName', 'faceTrackingARMKernel'});
% The generated zip folder is faceTrackingARMKernel.zip
```

```
-> Creating zip folder (it may take a few minutes) ....
```

Create Project Folder

Unzip faceTrackingARMKernel.zip into a folder named FaceTrackingARM. Unzip all files and remove the .zip files.

```
packngoDir = hUnzipPackageContents();
```

Update Makefile and Copy to Project Folder

The Makefile, faceTrackingARMMakefile.mk, provided in this example is written for Raspberry PI 2 with specific optimization flags. The Makefile was written to work with GCC in a Linux environment and with your OpenCV libraries located in /usr/local/lib. You can update the Makefile based on your target configuration. Copy the Makefile to the project folder.

```
copyfile('faceTrackingARMMakefile.mk', packngoDir);
% Also move the file containing the main function in the top level folder.
copyfile('faceTrackingARMMain.c', packngoDir);
% For simplicity, make sure the root directory name is matlab.
setRootDirectory(packngoDir);
```

Deployment on ARM

Deploy your project on ARM:

```
disp('Follow these steps to deploy your project on ARM');
```

Follow these steps to deploy your project on ARM

Transfer Code to ARM Target

Transfer your project folder named FaceTrackingARM to your ARM target using your preferred file transfer tool. Since the Raspberry Pi 2 (with Raspbian Stretch) already has an SSH server, you can use SFTP to transfer files from host to target.

For official deployment of this example, the FileZilla SFTP Client was installed on the host machine and the project folder was transferred from the host to the `/home/pi/FaceTrackingARM` folder on Raspberry Pi.

```
disp('Step-1: Transfer the folder 'FaceTrackingARM' to your ARM target');
```

```
Step-1: Transfer the folder 'FaceTrackingARM' to your ARM target
```

Build the Executable on ARM

Run the makefile to build the executable on ARM. For Raspberry Pi 2, (with Raspbian Stretch), open a command line terminal and 'cd' to `/home/pi/FaceTrackingARM`. Build the executable using the following command:

```
make -f faceTrackingARMMakefile.mk
```

The command creates an executable, `faceTrackingARMKernel`.

```
disp('Step-2: Build the executable on ARM using the shell command: make -f faceTrackingARMMakefile.mk');
```

```
Step-2: Build the executable on ARM using the shell command: make -f faceTrackingARMMakefile.mk
```

Run the Executable on ARM

Run the executable generated in the above step. For Raspberry Pi 2, (with Raspbian Stretch), use the following command in the shell window:

```
./faceTrackingARMKernel
```

Make sure that you are connected to the Raspberry Pi with a window manager, and not just through a command line terminal to avoid errors related to GTK. This is necessary for the tracking window to show up.

To close the video viewer while the executable is running on Raspberry Pi2, click on the video viewer and press the escape key.

```
disp('Step-3: Run the executable on ARM using the shell command: ./faceTrackingARMKernel');
```

```
Step-3: Run the executable on ARM using the shell command: ./faceTrackingARMKernel
```

Appendix - Helper Functions

```
% Configure coder to create executable. Use packNGo at post code
% generation stage.
function codegenArgs = createCodegenArgs()
    % Create arguments required for code generation.

    % First - create configuration object
    %
    % For standalone executable a main C function is required. The
    % faceTrackingARMMain.c created for this example is compatible
    % with the content of the file faceTrackingARMKernel.m
```

```

mainCFile = 'faceTrackingARMMain.c';

% Include helper functions
camCPPFile = 'helperOpenCVWebcam.cpp';
viewerCPPFile = 'helperOpenCVVideoViewer.cpp';

% Handle path with space
if contains(mainCFile, ' ')
    mainCFile = ['\" mainCFile '\"'];
    camCPPFile = ['\" camCPPFile '\"'];
    viewerCPPFile = ['\" viewerCPPFile '\"'];
end

% Create configuration object
cfg = coder.config('exe');
cfg.CustomSource = sprintf('%s\n%s\n%s',mainCFile,camCPPFile,viewerCPPFile);
cfg.CustomInclude = pwd;
% Set production hardware to ARM to generate ARM compatible portable code
cfg.HardwareImplementation.ProdHWDeviceType = 'ARM Compatible->ARM Cortex';
cfg.EnableOpenMP = false;

% Create input arguments
inRGB_type = coder.typeof(uint8(0),[480 640 3]);
% Use '-c' option to generate C code without calling C++ compiler.
codegenArgs = {'-config', cfg, '-c', '-args', {inRGB_type}};

end

% Unzip the packaged zip file
function packngoDir = hUnzipPackageContents()

    packngoDirName = 'FaceTrackingARM';

    % create packngo directory
    mkdir(packngoDirName);

    % get the name of the single zip file generated by packngo
    zipFile = dir('*.zip');
    assert(numel(zipFile)==1);

    unzip(zipFile.name,packngoDirName);

    % unzip internal zip files created in hierarchical packNGO
    zipFileInternal = dir(fullfile(packngoDirName,'*.zip'));

    for i=1:numel(zipFileInternal)
        unzip(fullfile(packngoDirName,zipFileInternal(i).name), ...
            packngoDirName);
    end
    % delete internal zip files
    delete(fullfile(packngoDirName,'*.zip'));
    packngoDir = fullfile(packngoDirName);
end

% Set root directory as matlab
function setRootDirectory(packngoDir)
    dirList = dir(packngoDir);
    if isempty(find(ismember({dirList.name},'matlab'), 1))

```

```
% root directory is not matlab. Change it to matlab
for i=1:length(dirList)
    thisDir = fullfile(packngoDir,dirList(i).name, 'toolbox', 'vision');
    if isfolder(thisDir)
        % rename the dir
        movefile(fullfile(packngoDir,dirList(i).name), ...
            fullfile(packngoDir,'matlab'));
        break;
    end
end
end
end
end
end
```

Video Display in a Custom User Interface

This example shows how to display multiple video streams in a custom graphical user interface (GUI).

Overview

When working on a project involving video processing, we are often faced with creating a custom user interface. It may be needed for the purpose of visualizing and/or demonstrating the effects of our algorithms on the input video stream. This example illustrates how to create a figure window with two axes to display two video streams. It also shows how to set up buttons and their corresponding callbacks.

The example is written as a function with the main body at the top and helper routines in the form of nested functions.

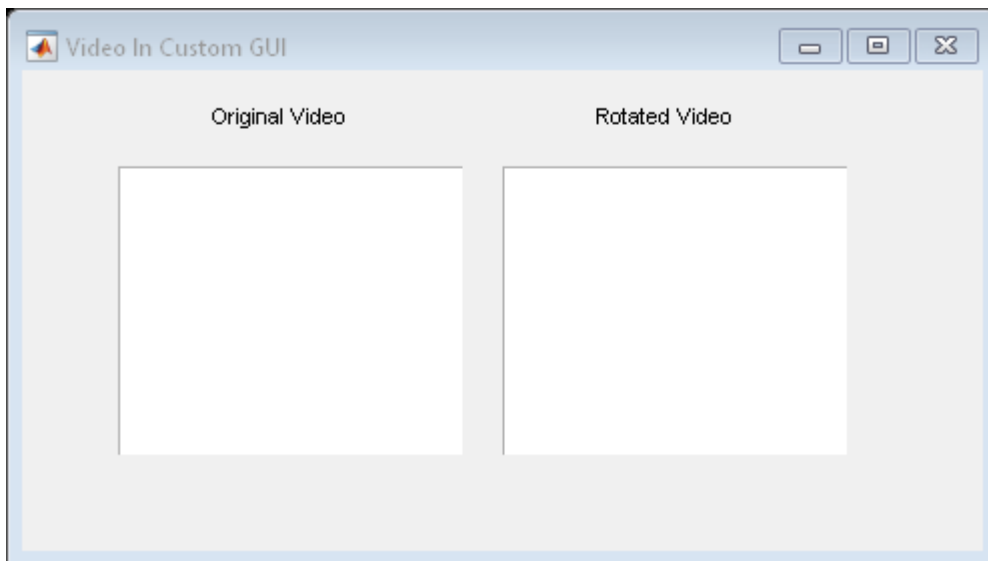
```
function VideoInCustomGUIExample()
```

Initialize the video reader.

```
videoSrc = vision.VideoFileReader('vipmen.avi', 'ImageColorSpace', 'Intensity');
```

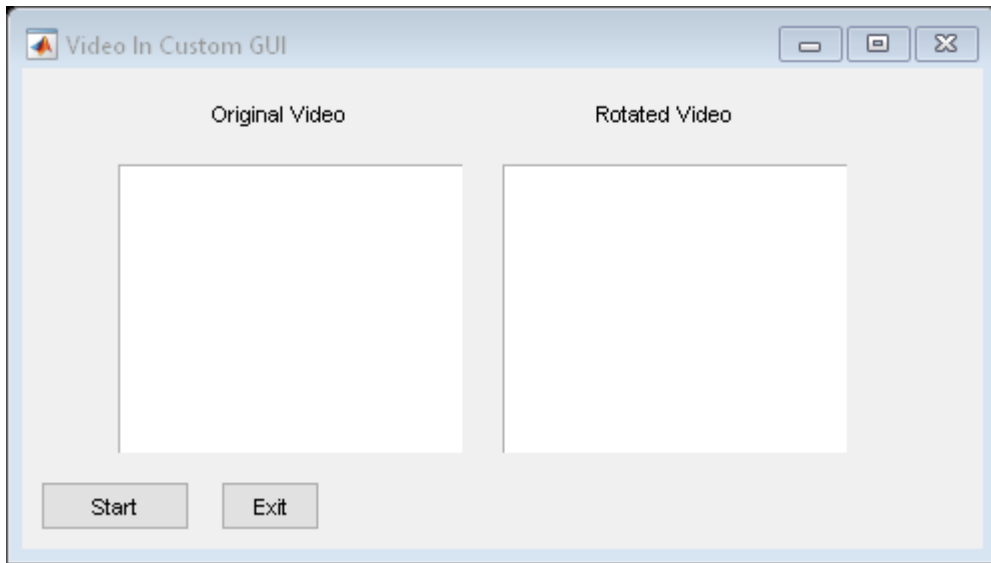
Create a figure window and two axes to display the input video and the processed video.

```
[hFig, hAxes] = createFigureAndAxes();
```



Add buttons to control video playback.

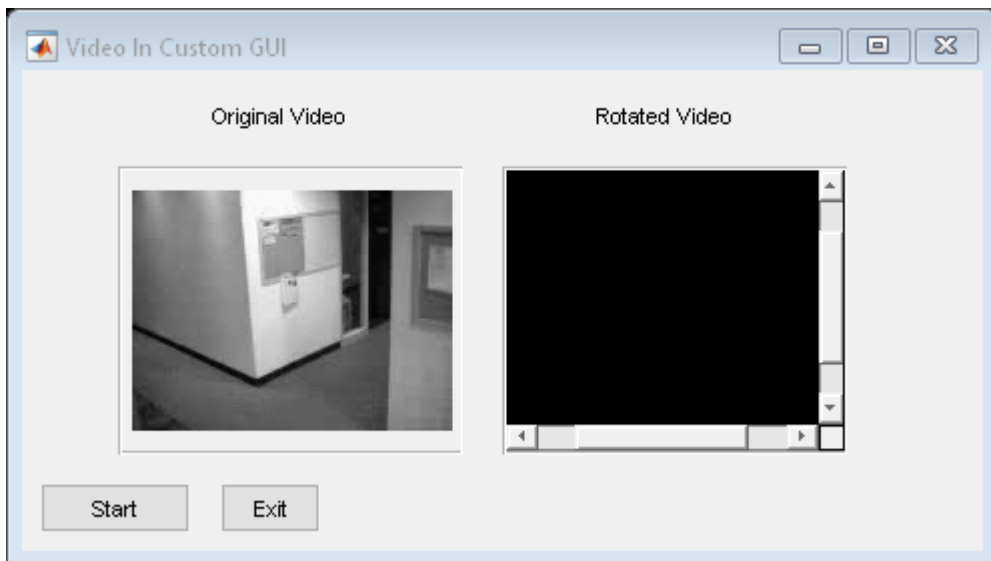
```
insertButtons(hFig, hAxes, videoSrc);
```



Interact with the New User Interface

Now that the GUI is constructed, we can press the play button to trigger the main video processing loop defined in the `getAndProcessFrame` function listed below.

```
% Initialize the display with the first frame of the video
frame = getAndProcessFrame(videoSrc, 0);
% Display input video frame on axis
showFrameOnAxis(hAxes.axis1, frame);
showFrameOnAxis(hAxes.axis2, zeros(size(frame)+60, 'uint8'));
```



Note that each video frame is centered in the axis box. If the axis size is bigger than the frame size, video frame borders are padded with background color. If axis size is smaller than the frame size scroll bars are added.

Create Figure, Axes, Titles

Create a figure window and two axes with titles to display two videos.

```
function [hFig, hAxes] = createFigureAndAxes()

% Close figure opened by last run
figTag = 'CVST_VideoOnAxis_9804532';
close(findobj('tag',figTag));

% Create new figure
hFig = figure('numbertitle', 'off', ...
    'name', 'Video In Custom GUI', ...
    'menubar','none', ...
    'toolbar','none', ...
    'resize', 'on', ...
    'tag',figTag, ...
    'renderer','painters', ...
    'position',[680 678 480 240],...
    'HandleVisibility','callback'); % hide the handle to prevent unintended modificat

% Create axes and titles
hAxes.axis1 = createPanelAxisTitle(hFig,[0.1 0.2 0.36 0.6],'Original Video'); % [X Y W H]
hAxes.axis2 = createPanelAxisTitle(hFig,[0.5 0.2 0.36 0.6],'Rotated Video');

end
```

Create Axis and Title

Axis is created on uipanel container object. This allows more control over the layout of the GUI. Video title is created using uicontrol.

```
function hAxis = createPanelAxisTitle(hFig, pos, axisTitle)

% Create panel
hPanel = uipanel('parent',hFig,'Position',pos,'Units','Normalized');

% Create axis
hAxis = axes('position',[0 0 1 1],'Parent',hPanel);
hAxis.XTick = [];
hAxis.YTick = [];
hAxis.XColor = [1 1 1];
hAxis.YColor = [1 1 1];
% Set video title using uicontrol. uicontrol is used so that text
% can be positioned in the context of the figure, not the axis.
titlePos = [pos(1)+0.02 pos(2)+pos(3)+0.3 0.3 0.07];
uicontrol('style','text',...
    'String', axisTitle,...
    'Units','Normalized',...
    'Parent',hFig,'Position', titlePos,...
    'BackgroundColor',hFig.Color);

end
```

Insert Buttons

Insert buttons to play, pause the videos.

```
function insertButtons(hFig,hAxes,videoSrc)
```

```

% Play button with text Start/Pause/Continue
uicontrol(hFig,'unit','pixel','style','pushbutton','string','Start',...
    'position',[10 10 75 25], 'tag','PButton123','callback',...
    {@playCallback,videoSrc,hAxes});

% Exit button with text Exit
uicontrol(hFig,'unit','pixel','style','pushbutton','string','Exit',...
    'position',[100 10 50 25],'callback', ...
    {@exitCallback,videoSrc,hFig});

end

```

Play Button Callback

This callback function rotates input video frame and displays original input video frame and rotated frame on axes. The function `showFrameOnAxis` is responsible for displaying a frame of the video on user-defined axis. This function is defined in the file `showFrameOnAxis.m`

```

function playCallback(hObject,~,videoSrc,hAxes)
    try
        % Check the status of play button
        isTextStart = strcmp(hObject.String,'Start');
        isTextCont = strcmp(hObject.String,'Continue');
        if isTextStart
            % Two cases: (1) starting first time, or (2) restarting
            % Start from first frame
            if isDone(videoSrc)
                reset(videoSrc);
            end
        end
        if (isTextStart || isTextCont)
            hObject.String = 'Pause';
        else
            hObject.String = 'Continue';
        end

        % Rotate input video frame and display original and rotated
        % frames on figure
        angle = 0;
        while strcmp(hObject.String, 'Pause') && ~isDone(videoSrc)
            % Get input video frame and rotated frame
            [frame,rotatedImg,angle] = getAndProcessFrame(videoSrc,angle);
            % Display input video frame on axis
            showFrameOnAxis(hAxes.axis1, frame);
            % Display rotated video frame on axis
            showFrameOnAxis(hAxes.axis2, rotatedImg);
        end

        % When video reaches the end of file, display "Start" on the
        % play button.
        if isDone(videoSrc)
            hObject.String = 'Start';
        end
    catch ME
        % Re-throw error message if it is not related to invalid handle
        if ~strcmp(ME.identifier, 'MATLAB:class:InvalidHandle')
            rethrow(ME);
        end
    end
end

```

```
end  
end
```

Video Processing Algorithm

This function defines the main algorithm that is invoked when play button is activated.

```
function [frame,rotatedImg,angle] = getAndProcessFrame(videoSrc,angle)  
  
    % Read input video frame  
    frame = step(videoSrc);  
  
    % Pad and rotate input video frame  
    paddedFrame = padarray(frame, [30 30], 0, 'both');  
    rotatedImg = imrotate(paddedFrame, angle, 'bilinear', 'crop');  
    angle      = angle + 1;  
end
```

Exit Button Callback

This callback function releases system objects and closes figure window.

```
function exitCallback(~,~,videoSrc,hFig)  
  
    % Close the video file  
    release(videoSrc);  
    % Close the figure window  
    close(hFig);  
end  
end
```


Deep Learning, Semantic Segmentation, and Detection Examples

- “Point Cloud Classification Using PointNet Deep Learning” on page 3-2
- “Object Detection Using SSD Deep Learning” on page 3-23
- “Object Detection in a Cluttered Scene Using Point Feature Matching” on page 3-32
- “Semantic Segmentation Using Deep Learning” on page 3-43
- “Calculate Segmentation Metrics in Block-Based Workflow” on page 3-59
- “Semantic Segmentation of Multispectral Images Using Deep Learning” on page 3-64
- “3-D Brain Tumor Segmentation Using Deep Learning” on page 3-81
- “Image Category Classification Using Bag of Features” on page 3-93
- “Image Category Classification Using Deep Learning” on page 3-100
- “Image Retrieval Using Customized Bag of Features” on page 3-109
- “Create SSD Object Detection Network” on page 3-115
- “Train YOLO v2 Network for Vehicle Detection” on page 3-118
- “Object Detection Using YOLO v2 Deep Learning” on page 3-123
- “Import Pretrained ONNX YOLO v2 Object Detector” on page 3-133
- “Export YOLO v2 Object Detector to ONNX” on page 3-140
- “Estimate Anchor Boxes From Training Data” on page 3-146
- “Object Detection Using YOLO v3 Deep Learning” on page 3-150
- “Object Detection Using YOLO v2 Deep Learning” on page 3-170
- “Create YOLO v2 Object Detection Network” on page 3-180
- “Train Object Detector Using R-CNN Deep Learning” on page 3-183
- “Object Detection Using Faster R-CNN Deep Learning” on page 3-197
- “Train Classification Network to Classify Object in 3-D Point Cloud” on page 3-207
- “Estimate Body Pose Using Deep Learning” on page 3-217
- “Generate Image from Segmentation Map Using Deep Learning” on page 3-224
- “Create Simple Semantic Segmentation Network in Deep Network Designer” on page 3-242
- “Train ACF-Based Stop Sign Detector” on page 3-247
- “Activity Recognition from Video and Optical Flow Data Using Deep Learning” on page 3-249
- “Train Fast R-CNN Stop Sign Detector” on page 3-273

Point Cloud Classification Using PointNet Deep Learning

This example shows how to train a PointNet network for point cloud classification.

Point cloud data is acquired by a variety of sensors, such as lidar, radar, and depth cameras. These sensors capture 3-D position information about objects in a scene, which is useful for many applications in autonomous driving and augmented reality. For example, discriminating vehicles from pedestrians is critical for planning the path of an autonomous vehicle. However, training robust classifiers with point cloud data is challenging because of the sparsity of data per object, object occlusions, and sensor noise. Deep learning techniques have been shown to address many of these challenges by learning robust feature representations directly from point cloud data. One of the seminal deep learning techniques for point cloud classification is PointNet [1 on page 3-0].

This example trains a PointNet classifier on the Sydney Urban Objects data set created by the University of Sydney [2 on page 3-0]. This data set provides a collection of point cloud data acquired from an urban environment using a lidar sensor. The data set has 100 labeled objects from 14 different categories, such as car, pedestrian, and bus.

Load data set

Download and extract the Sydney Urban Objects data set to a temporary directory.

```
downloadDirectory = tempdir;  
datapath = downloadSydneyUrbanObjects(downloadDirectory);
```

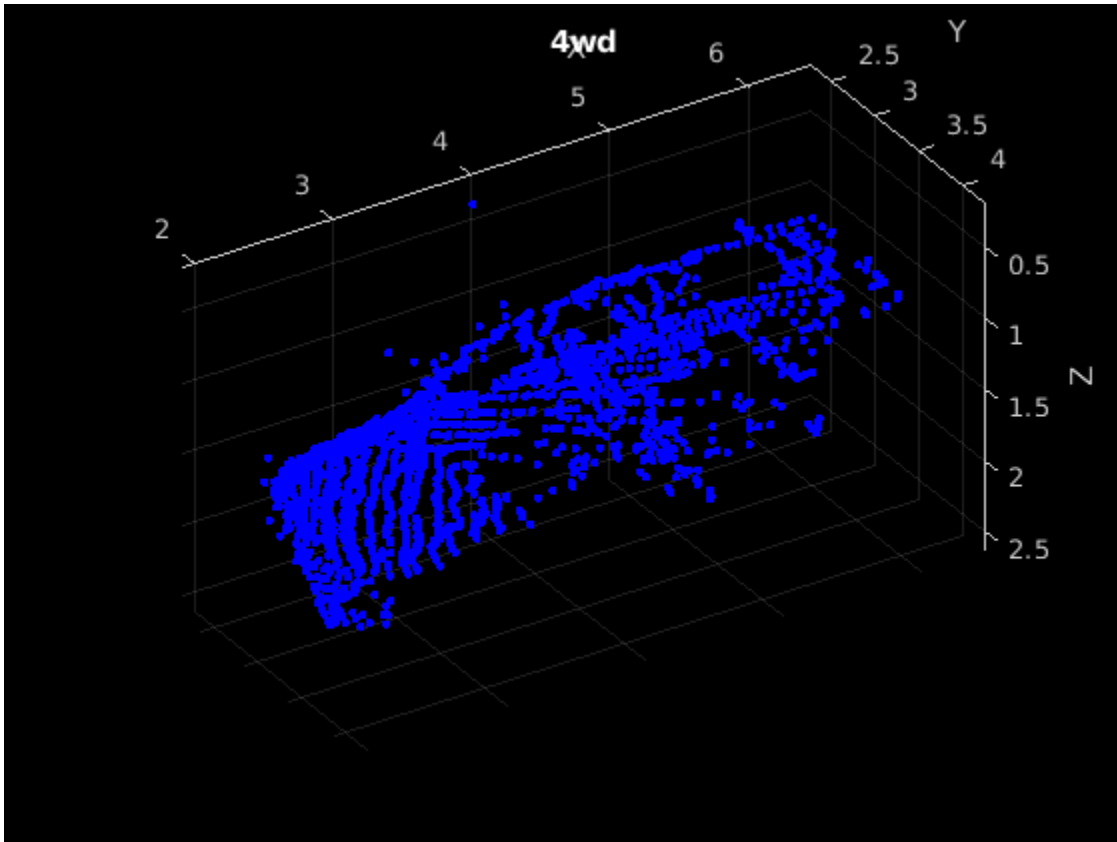
Load the downloaded training and validation data set using the `sydneyUrbanObjectsClassificationDatastore` helper function listed at the end of this example. Use the first three data folds for training and the fourth for validation.

```
foldsTrain = 1:3;  
foldsVal = 4;  
dsTrain = sydneyUrbanObjectsClassificationDatastore(datapath, foldsTrain);  
dsVal = sydneyUrbanObjectsClassificationDatastore(datapath, foldsVal);
```

Read one of the training samples and visualize it using `pcshow`.

```
data = read(dsTrain);  
ptCloud = data{1,1};  
label = data{1,2};
```

```
figure  
pcshow(ptCloud.Location, [0 0 1], "MarkerSize", 40, "VerticalAxisDir", "down")  
xlabel("X")  
ylabel("Y")  
zlabel("Z")  
title(label)
```

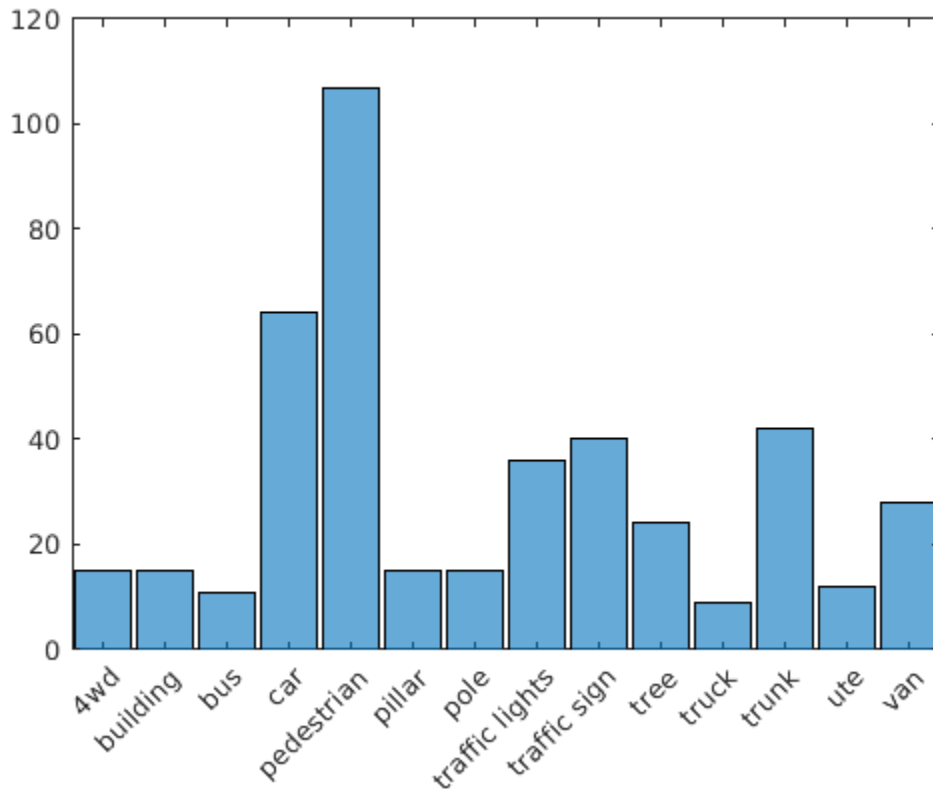


Read the labels and count the number of points assigned to each label to better understand the distribution of labels within the data set.

```
dsLabelCounts = transform(dsTrain,@(data){data{2} data{1}.Count});  
labelCounts = readall(dsLabelCounts);  
labels = vertcat(labelCounts{:,1});  
counts = vertcat(labelCounts{:,2});
```

Next, use a histogram to visualize the class distribution.

```
figure  
histogram(labels)
```



The label histogram shows that the data set is imbalanced and biased towards cars and pedestrians, which can prevent the training of a robust classifier. You can address class imbalance by oversampling the infrequent classes. For the Sydney Urban Objects data set, duplicating files corresponding to the infrequent classes is a simple method to address the class imbalance.

Group the files by label, count the number of observations per class, and use the `randReplicateFiles` helper function, listed at the end of this example, to randomly oversample the files to the desired number of observations per class.

```
rng(0)
[G,classes] = findgroups(labels);
numObservations = splitapply(@numel,labels,G);
desiredNumObservationsPerClass = max(numObservations);
files = splitapply(@(x){randReplicateFiles(x,desiredNumObservationsPerClass)},dsTrain.Files,G);
files = vertcat(files{:});
dsTrain.Files = files;
```

Shuffle the replicated file list so that the same classes of observations are not in every training batch.

```
dsTrain.Files = dsTrain.Files(randperm(length(dsTrain.Files)));
```

Set the mini-batch size of the training and validation datastores to 128.

```
dsTrain.MinibatchSize = 128;
dsVal.MinibatchSize = 128;
```


Data Augmentation

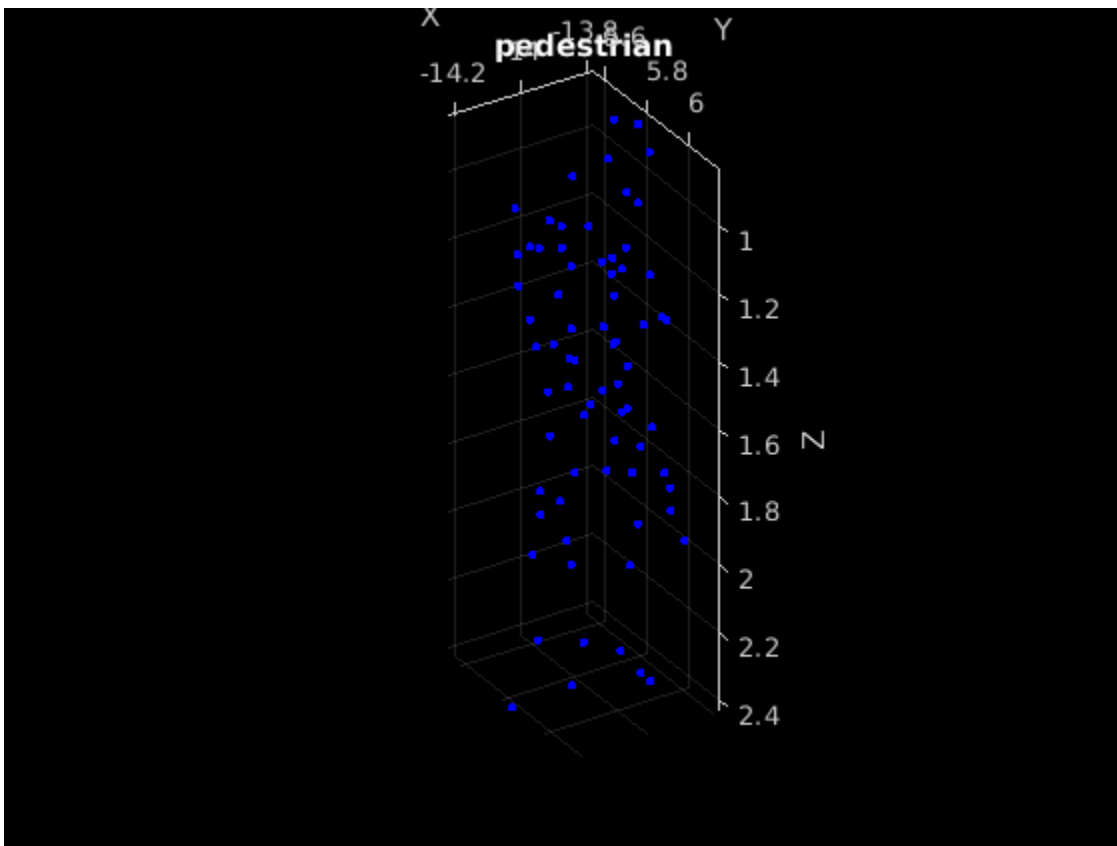
Duplicating the files to address class imbalance increases the likelihood of overfitting the network because much of the training data is identical. To offset this effect, apply data augmentation to the training data using the `transform` and `augmentPointCloud` helper function, which randomly rotates the point cloud, randomly removes points, and randomly jitters points with Gaussian noise.

```
dsTrain = transform(dsTrain,@augmentPointCloud);
```

Preview one of the augmented training samples.

```
data = preview(dsTrain);
ptCloud = data{1,1};
label = data{1,2};
```

```
figure
pcshow(ptCloud.Location,[0 0 1],"MarkerSize",40,"VerticalAxisDir","down")
xlabel("X")
ylabel("Y")
zlabel("Z")
title(label)
```



Note that because the data for measuring the performance of the trained network must be representative of the original data set, data augmentation is not applied to validation or test data.

Data Preprocessing

Two preprocessing steps are required to prepare the point cloud data for training and prediction.

First, to enable batch processing during training, select a fixed number of points from each point cloud. The optimal number of points depends on the data set and the number of points required to accurately capture the shape of the object. To help select the appropriate number of points, compute the minimum, maximum, and mean number of points per class.

```
minPointCount = splitapply(@min,counts,G);
maxPointCount = splitapply(@max,counts,G);
meanPointCount = splitapply(@(x)round(mean(x)),counts,G);
```

```
stats = table(classes,numObservations,minPointCount,maxPointCount,meanPointCount)
```

```
stats=14x5 table
      classes      numObservations      minPointCount      maxPointCount      meanPointCount
      _____      _____      _____      _____      _____
4wd                15                140                1955                751
building           15                193                8455                2708
bus                11                126                11767               2190
car                64                52                 2377                528
pedestrian         107               20                 297                110
pillar             15                80                 751                357
pole              15                13                 253                 90
traffic lights     36                38                 352                161
traffic sign       40                18                 736                126
tree               24                53                 2953               470
truck              9                 445               3013               1376
trunk              42                32                 766                241
ute                12                90                 1380               580
van                28                91                 5809               1125
```

Because of the large amount of intra-class and inter-class variability in the number of points per class, choosing a value that fits all classes is difficult. One heuristic is to choose enough points to adequately capture the shape of the objects while not increasing the computational cost by processing too many points. A value of 1024 provides a good tradeoff between these two facets. You can also select the optimal number of points based on empirical analysis. However, that is beyond the scope of this example. Use the `transform` function to select 1024 points in the training and validation sets.

```
numPoints = 1024;
dsTrain = transform(dsTrain,@(data)selectPoints(data,numPoints));
dsVal = transform(dsVal,@(data)selectPoints(data,numPoints));
```

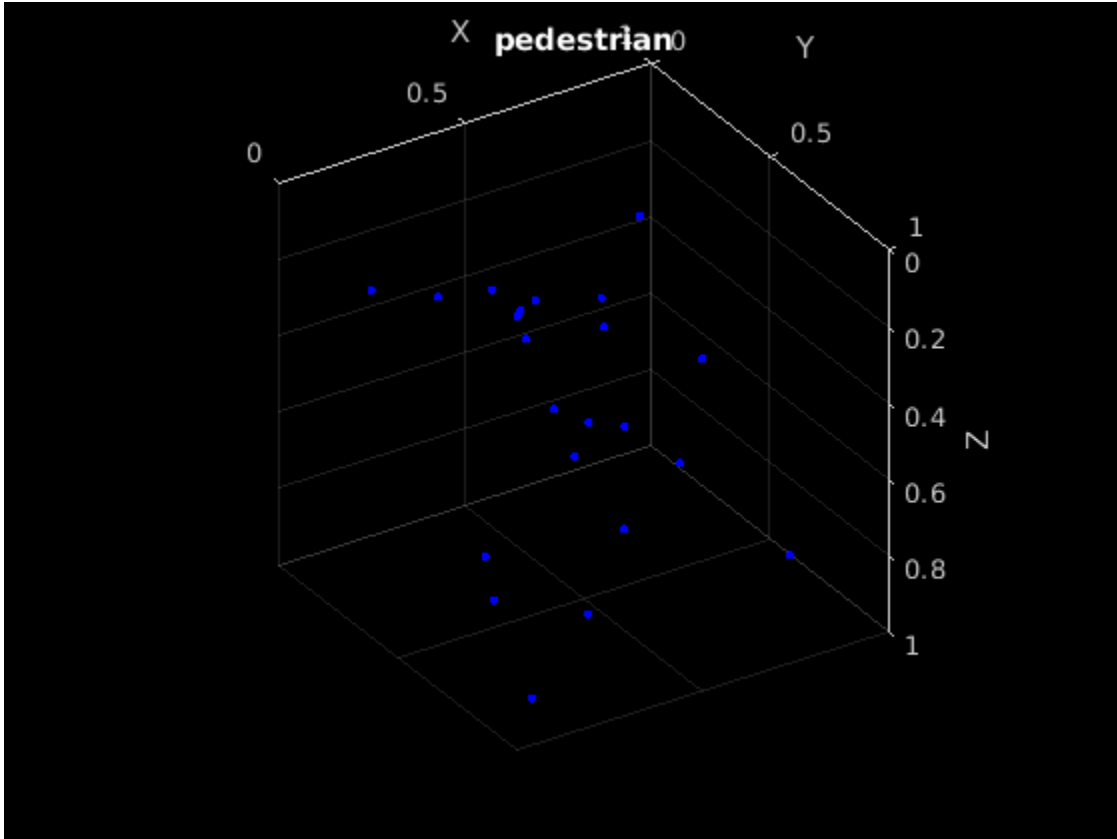
The last preprocessing step is to normalize the point cloud data between 0 and 1 to account for large differences in the range of data values. For example, objects closer to the lidar sensor have smaller values compared to objects that are further away. These differences can hinder the convergence of the network during training. Use `transform` to normalize the point cloud data in the training and validation sets.

```
dsTrain = transform(dsTrain,@preprocessPointCloud);
dsVal = transform(dsVal,@preprocessPointCloud);
```

Preview the augmented and preprocessed training data.

```
data = preview(dsTrain);
figure
pcshow(data{1,1},[0 0 1],"MarkerSize",40,"VerticalAxisDir","down");
```

```
xlabel("X")  
ylabel("Y")  
zlabel("Z")  
title(data{1,2})
```



Define PointNet Model

The PointNet classification model consists of two components. The first component is a point cloud encoder that learns to encode sparse point cloud data into a dense feature vector. The second component is a classifier that predicts the categorical class of each encoded point cloud.

The PointNet encoder model is further composed of four models followed by a max operation.

- 1 Input transform model
- 2 Shared MLP model
- 3 Feature transform model
- 4 Shared MLP model

The shared MLP model is implemented using a series of convolution, batch normalization, and ReLU operations. The convolution operation is configured such that the weights are shared across the input point cloud. The transform model is composed of a shared MLP and a learnable transform matrix that is applied to each point cloud. The shared MLP and the max operation make the PointNet encoder invariant to the order in which the points are processed, while the transform model provides invariance to orientation changes.

Define PointNet Encoder Model Parameters

The shared MLP and transform models are parameterized by the number of input channels and the hidden channel sizes. The values chosen in this example are selected by tuning these hyperparameters on the Sydney Urban Objects data set. Note that if you want to apply PointNet to a different data set, you must perform additional hyperparameter tuning.

Set the input transform model input channel size to three and the hidden channel sizes to 64, 128, and 256 and use the `initializeTransform` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 3;
hiddenChannelSize1 = [64,128];
hiddenChannelSize2 = 256;
[parameters.InputTransform, state.InputTransform] = initializeTransform(inputChannelSize,hiddenChannelSize1,hiddenChannelSize2);
```

Set the first shared MLP model input channel size to three and the hidden channel size to 64 and use the `initializeSharedMLP` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 3;
hiddenChannelSize = [64 64];
[parameters.SharedMLP1,state.SharedMLP1] = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Set the feature transformation model input channel size to 64 and hidden channel sizes to 64, 128, and 256 and use the `initializeTransform` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 64;
hiddenChannelSize1 = [64,128];
hiddenChannelSize2 = 256;
[parameters.FeatureTransform, state.FeatureTransform] = initializeTransform(inputChannelSize,hiddenChannelSize1,hiddenChannelSize2);
```

Set the second shared MLP model input channel size to 64 and the hidden channel size to 64 and use the `initializeSharedMLP` function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 64;
hiddenChannelSize = 64;
[parameters.SharedMLP2,state.SharedMLP2] = initializeSharedMLP(inputChannelSize,hiddenChannelSize);
```

Define PointNet Classifier Model Parameters

The PointNet classifier model consists of a shared MLP, a fully connected operation, and a softmax activation. Set the classifier model input size to 64 and the hidden channel size to 512 and 256 and use the `initializeClassifier` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 64;
hiddenChannelSize = [512,256];
numClasses = numel(classes);
[parameters.ClassificationMLP, state.ClassificationMLP] = initializeClassificationMLP(inputChannelSize,hiddenChannelSize,numClasses);
```

Define PointNet Function

Create the function `pointnetClassifier`, listed in the Model Function section at the end of the example, to compute the outputs of the PointNet model. The function model takes as input the point

cloud data, the learnable model parameters, the model state, and a flag that specifies whether the model returns outputs for training or prediction. The network returns the predictions for classifying the input point cloud.

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function section of the example, that takes as input the model parameters, the model state, and a mini-batch of input data, and returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

Specify Training Options

Train for 40 epochs. Set the initial learning rate to 0.001 and the L2 regularization factor to 0.01.

```
numEpochs = 40;
learnRate = 0.001;
l2Regularization = 0.01;
learnRateDropPeriod = 15;
learnRateDropFactor = 0.5;
```

Initialize the options for Adam optimization.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train PointNet

Train the model using a custom training loop.

Shuffle the data at the beginning of training.

For each iteration:

- Read a batch of data.
- Evaluate the model gradients.
- Apply L2 weight regularization.
- Use `adamupdate` to update the model parameters.
- Update the training progress plot.

At the end of each epoch, evaluate the model against the validation data set and collect confusion metrics to measure classification accuracy as training progresses.

After completing `learnRateDropPeriod` epochs, reduce the learning rate by a factor of `learnRateDropFactor`.

Initialize the moving average of the parameter gradients and the element-wise squares of the gradients used by the Adam optimizer.

```
avgGradients = [];
avgSquaredGradients = [];
```

Train the model if `doTraining` is true. Otherwise, load a pretrained network.

Note that training was verified on an NVIDIA Titan X with 12 GB of GPU memory. If your GPU has less memory, you may run out of memory during training. If this happens, lower the `miniBatchSize`. Training this network takes about 5 minutes. Depending on your GPU hardware, it can take longer.

```
doTraining = false;

if doTraining

    % Use the configureTrainingProgressPlot function, listed at the end of the
    % example, to initialize the training progress plot to display the training
    % loss, training accuracy, and validation accuracy.
    [lossPlotter, trainAccPlotter, valAccPlotter] = initializeTrainingProgressPlot;

    numClasses = numel(classes);
    iteration = 0;
    start = tic;
    for epoch = 1:numEpochs

        % Reset training and validation datastores.
        reset(dsTrain);
        reset(dsVal);

        % Iterate through data set.
        while hasdata(dsTrain)
            iteration = iteration + 1;

            % Read data.
            data = read(dsTrain);

            % Create batch.
            [XTrain, YTrain] = batchData(data);

            % Evaluate the model gradients and loss using dlfeval and the
            % modelGradients function.
            [gradients, loss, state, acc] = dlfeval(@modelGradients, XTrain, YTrain, parameters, state);

            % L2 regularization.
            gradients = dlupdate(@(g,p) g + l2Regularization*p, gradients, parameters);

            % Update the network parameters using the Adam optimizer.
            [parameters, avgGradients, avgSquaredGradients] = adamupdate(parameters, gradients,
                avgGradients, avgSquaredGradients, iteration, ...
                learnRate, gradientDecayFactor, squaredGradientDecayFactor);

            % Update the training progress.
            D = duration(0,0,toc(start), "Format", "hh:mm:ss");
            title(lossPlotter.Parent, "Epoch: " + epoch + ", Elapsed: " + string(D))
            addpoints(lossPlotter, iteration, double(gather(extractdata(loss))))
            addpoints(trainAccPlotter, iteration, acc);
            drawnow
        end

        % Evaluate the model on validation data.
        cmat = sparse(numClasses, numClasses);
        while hasdata(dsVal)

            % Get the next batch of data.
            data = read(dsVal);
```

```

        % Create batch.
        [XVal,YVal] = batchData(data);

        % Compute label predictions.
        isTraining = false;
        YPred = pointnetClassifier(XVal,parameters,state,isTraining);

        % Choose prediction with highest score as the class label for
        % XTest.
        [~,YValLabel] = max(YVal,[],1);
        [~,YPredLabel] = max(YPred,[],1);

        % Collect confusion metrics.
        cmat = aggregateConfusionMetric(cmat,YValLabel,YPredLabel);
    end

    % Update training progress plot with average classification accuracy.
    acc = sum(diag(cmat))./sum(cmat,"all");
    addpoints(valAccPlotter,iteration,acc);

    % Update the learning rate.
    if mod(epoch,learnRateDropPeriod) == 0
        learnRate = learnRate * learnRateDropFactor;
    end

    % Reset training and validation datastores.
    reset(dsTrain);
    reset(dsVal);
end

else
    % Download pretrained model parameters, model state, and validation
    % results.
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/pretrainedPointNet.mat';

    pretrainedNetwork = fullfile(pwd,'pretrainedPointNet.mat');
    if ~exist(pretrainedNetwork,'file')
        disp('Downloading pretrained network (5 MB)...');
        websave(pretrainedNetwork,pretrainedURL);
    end

    % Load pretrained model.
    pretrainedResults = load('pretrainedPointNet.mat');
    parameters = pretrainedResults.parameters;
    state = pretrainedResults.state;
    cmat = pretrainedResults.cmat;

    % Move model parameters to the GPU if possible and convert to a dlarray.
    parameters = prepareForPrediction(parameters,@(x)dlarray(toDevice(x,canUseGPU)));

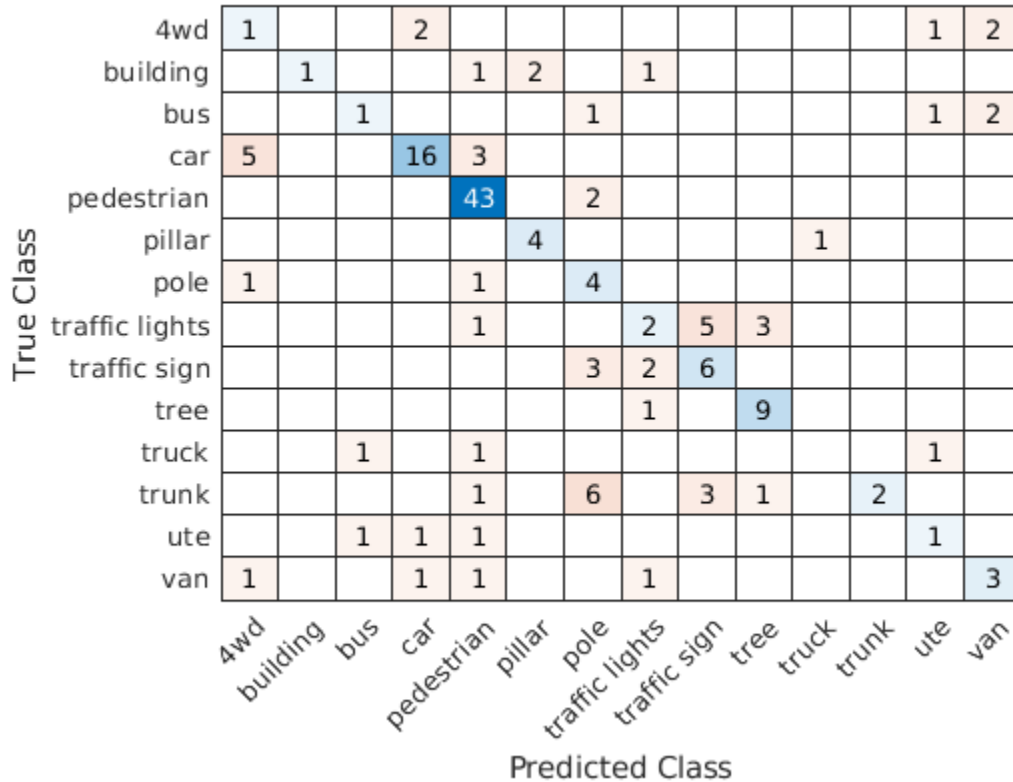
    % Move model state to the GPU if possible.
    state = prepareForPrediction(state,@(x)toDevice(x,canUseGPU));
end

```

Downloading pretrained network (5 MB)...

Display the validation confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```



Compute the mean training and validation accuracy.

```
acc = sum(diag(cmat))./sum(cmat,"all")
```

```
acc = 0.6000
```

Due to the limited number of training samples in the Sydney Urban Objects data set, increasing the validation accuracy beyond 60% is challenging. The model easily overfits the training data in the absence of the augmentation defined in the `augmentPointCloudData` helper function. To improve the robustness of the PointNet classifier, additional training is required.

Classify Point Cloud Data Using PointNet

Load point cloud data with `pcread`, preprocess the point cloud using the same function used during training, and convert the result to a `darray`.

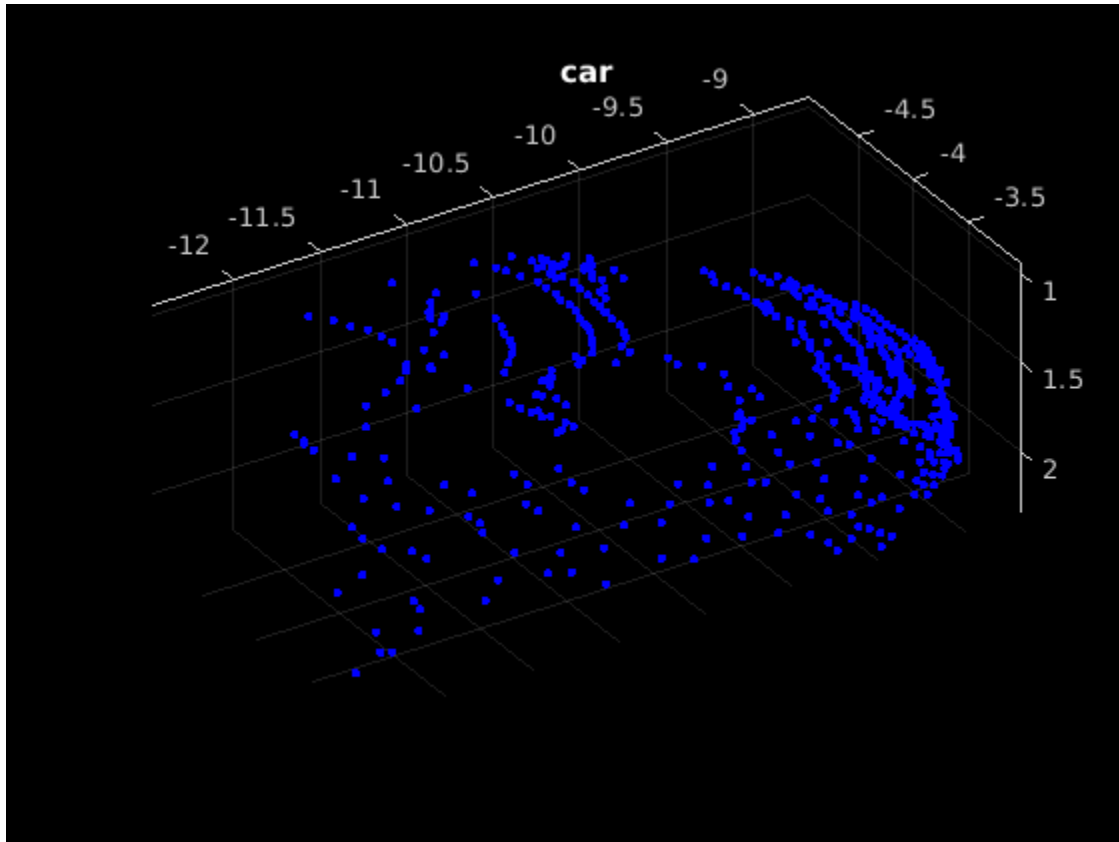
```
ptCloud = pcread("car.pcd");
X = preprocessPointCloud(ptCloud);
dX = darray(X{1},"SCSB");
```

Predict point cloud labels with the `pointnetClassifier` model function.

```
YPred = pointnetClassifier(dX,parameters,state,false);
[~,classIdx] = max(YPred,[],1);
```


Display the point cloud and the predicted label with the highest score.

```
figure
pcshow(ptCloud.Location,[0 0 1],"MarkerSize",40,"VerticalAxisDir","down")
title(classes(classIdx))
```



Model Gradients Function

The `modelGradients` function takes as input a mini-batch of data `dLX`, the corresponding target `dLY`, and the learnable parameters, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss. The loss includes a regularization term designed to ensure the feature transformation matrix predicted by the PointNet encoder is approximately orthogonal. To compute the gradients, evaluate the `modelGradients` function using the `dlfeval` function in the training loop.

```
function [gradients, loss, state, acc] = modelGradients(X,Y,parameters,state)

% Execute the model function.
isTraining = true;
[YPred,state,dLT] = pointnetClassifier(X,parameters,state,isTraining);

% Add regularization term to ensure feature transform matrix is
% approximately orthogonal.
K = size(dLT,1);
B = size(dLT, 4);
I = repelem(eye(K),1,1,1,B);
dLI = dlarray(I,"SSCB");
```

```
treg = mse(dLI,pagemtimes(dLT,permute(dLT,[2 1 3 4])));
factor = 0.001;

% Compute the loss.
loss = crossentropy(YPred,Y) + factor*treg;

% Compute the parameter gradients with respect to the loss.
gradients = dlgradient(loss, parameters);

% Compute training accuracy metric.
[~,YTest] = max(Y,[],1);
[~,YPred] = max(YPred,[],1);
acc = gather(extractdata(sum(YTest == YPred)./numel(YTest)));

end
```

PointNet Classifier Function

The `pointnetClassifier` function takes as input the point cloud data `dLX`, the learnable model parameters, the model state, and the flag `isTraining`, which specifies whether the model returns outputs for training or prediction. Then, the function invokes the PointNet encoder and a multilayer perceptron to extract classification features. During training, dropout is applied after each perceptron operation. After the last perceptron, a `fullyconnect` operation maps the classification features to the number of classes and a softmax activation is used to normalize the output into a probability distribution of labels. The probability distribution, the updated model state, and the feature transformation matrix predicted by the PointNet encoder are returned as outputs.

```
function [dLY,state,dLT] = pointnetClassifier(dLX,parameters,state,isTraining)

% Invoke the PointNet encoder.
[dLY,state,dLT] = pointnetEncoder(dLX,parameters,state,isTraining);

% Invoke the classifier.
p = parameters.ClassificationMLP.Perceptron;
s = state.ClassificationMLP.Perceptron;
for k = 1:numel(p)

    [dLY, s(k)] = perceptron(dLY,p(k),s(k),isTraining);

    % If training, apply inverted dropout with a probability of 0.3.
    if isTraining
        probability = 0.3;
        dropoutScaleFactor = 1 - probability;
        dropoutMask = ( rand(size(dLY), "like", dLY) > probability ) / dropoutScaleFactor;
        dLY = dLY.*dropoutMask;
    end

end

state.ClassificationMLP.Perceptron = s;

% Apply final fully connected and softmax operations.
weights = parameters.ClassificationMLP.FC.Weights;
bias = parameters.ClassificationMLP.FC.Bias;
dLY = fullyconnect(dLY,weights,bias);
dLY = softmax(dLY);

end
```

PointNet Encoder Function

The `pointnetEncoder` function processes the input `dLX` using an input transform, a shared MLP, a feature transform, a second shared MLP, and a max operation, and returns the result of the max operation.

```
function [dLY,state,T] = pointnetEncoder(dLX,parameters,state,isTraining)
% Input transform.
[dLY,state.InputTransform] = dataTransform(dLX,parameters.InputTransform,state.InputTransform,isTraining);

% Shared MLP.
[dLY,state.SharedMLP1.Perceptron] = sharedMLP(dLY,parameters.SharedMLP1.Perceptron,state.SharedMLP1.Perceptron);

% Feature transform.
[dLY,state.FeatureTransform,T] = dataTransform(dLY,parameters.FeatureTransform,state.FeatureTransform,isTraining);

% Shared MLP.
[dLY,state.SharedMLP2.Perceptron] = sharedMLP(dLY,parameters.SharedMLP2.Perceptron,state.SharedMLP2.Perceptron);

% Max operation.
dLY = max(dLY,[],1);
end
```

Shared Multilayer Perceptron Function

The shared multilayer perceptron function processes the input `dLX` using a series of perceptron operations and returns the result of the last perceptron.

```
function [dLY,state] = sharedMLP(dLX,parameters,state,isTraining)
dLY = dLX;
for k = 1:numel(parameters)
    [dLY, state(k)] = perceptron(dLY,parameters(k),state(k),isTraining);
end
end
```

Perceptron Function

The perceptron function processes the input `dLX` using a convolution, a batch normalization, and a relu operation and returns the output of the ReLU operation.

```
function [dLY,state] = perceptron(dLX,parameters,state,isTraining)
% Convolution.
W = parameters.Conv.Weights;
B = parameters.Conv.Bias;
dLY = dlconv(dLX,W,B);

% Batch normalization. Update batch normalization state when training.
offset = parameters.BatchNorm.Offset;
scale = parameters.BatchNorm.Scale;
trainedMean = state.BatchNorm.TrainedMean;
trainedVariance = state.BatchNorm.TrainedVariance;
if isTraining
    [dLY,trainedMean,trainedVariance] = batchnorm(dLY,offset,scale,trainedMean,trainedVariance);

    % Update state.
    state.BatchNorm.TrainedMean = trainedMean;
    state.BatchNorm.TrainedVariance = trainedVariance;
else
```

```

    dLY = batchnorm(dLY,offset,scale,trainedMean,trainedVariance);
end

% ReLU.
dLY = relu(dLY);
end

```

Data Transform Function

The `dataTransform` function processes the input `dLX` using a shared MLP, a max operation, and another shared MLP to predict a transformation matrix `T`. The transformation matrix is applied to the input `dLX` using a batched matrix multiply operation. The function returns the result of the batched matrix multiply and the transformation matrix.

```

function [dLY,state,T] = dataTransform(dLX,parameters,state,isTraining)

% Shared MLP.
[dLY,state.Block1.Perceptron] = sharedMLP(dLX,parameters.Block1.Perceptron,state.Block1.Perceptron);

% Max operation.
dLY = max(dLY,[],1);

% Shared MLP.
[dLY,state.Block2.Perceptron] = sharedMLP(dLY,parameters.Block2.Perceptron,state.Block2.Perceptron);

% Transform net (T-Net). Apply last fully connected operation as W*X to
% predict transformation matrix T.
dLY = extractdata(dLY);
dLY = squeeze(dLY); % N-by-B
T = parameters.Transform * dLY; % K^2-by-B

% Reshape T into a square matrix.
K = sqrt(size(T,1));
T = reshape(T,K,K,1,[]); % [K K 1 B]
T = T + eye(K);

% Apply to input dLX using batch matrix multiply.
X = extractdata(dLX); % [M 1 K B]
[C,B] = size(X,[3 4]);
X = reshape(X,[],C,1,B); % [M K 1 B]
Y = pagetimes(X,T);
dLY = darray(Y,"SCSB");
end

```

Model Parameter Initialization Functions

initializeTransform Function

The `initializeTransform` function takes as input the channel size and the number of hidden channels for the two shared MLPs, and returns the initialized parameters in a struct. The parameters are initialized using He weight initialization [3 on page 3-0].

```

function [params,state] = initializeTransform(inputChannelSize,block1,block2)
[params.Block1,state.Block1] = initializeSharedMLP(inputChannelSize,block1);
[params.Block2,state.Block2] = initializeSharedMLP(block1(end),block2);

% Parameters for the transform matrix.

```

```
params.Transform = darray(zeros(inputChannelSize^2,block2(end)));
end
```

initializeSharedMLP Function

The initializeSharedMLP function takes as input the channel size and the hidden channel size, and returns the initialized parameters in a struct. The parameters are initialized using He weight initialization.

```
function [params,state] = initializeSharedMLP(inputChannelSize,hiddenChannelSize)
weights = initializeWeightsHe([1 1 inputChannelSize hiddenChannelSize(1)]);
bias = zeros(hiddenChannelSize(1),1,"single");
p.Conv.Weights = darray(weights);
p.Conv.Bias = darray(bias);

p.BatchNorm.Offset = darray(zeros(hiddenChannelSize(1),1,"single"));
p.BatchNorm.Scale = darray(ones(hiddenChannelSize(1),1,"single"));

s.BatchNorm.TrainedMean = zeros(hiddenChannelSize(1),1,"single");
s.BatchNorm.TrainedVariance = ones(hiddenChannelSize(1),1,"single");

params.Perceptron(1) = p;
state.Perceptron(1) = s;

for k = 2:numel(hiddenChannelSize)
weights = initializeWeightsHe([1 1 hiddenChannelSize(k-1) hiddenChannelSize(k)]);
bias = zeros(hiddenChannelSize(k),1,"single");
p.Conv.Weights = darray(weights);
p.Conv.Bias = darray(bias);

p.BatchNorm.Offset = darray(zeros(hiddenChannelSize(k),1,"single"));
p.BatchNorm.Scale = darray(ones(hiddenChannelSize(k),1,"single"));

s.BatchNorm.TrainedMean = zeros(hiddenChannelSize(k),1,"single");
s.BatchNorm.TrainedVariance = ones(hiddenChannelSize(k),1,"single");

params.Perceptron(k) = p;
state.Perceptron(k) = s;
end
end
```

initializeClassificationMLP Function

The initializeClassificationMLP function takes as input the channel size, the hidden channel size, and the number of classes and returns the initialized parameters in a struct. The shared MLP is initialized using He weight initialization and the final fully connected operation is initialized using random Gaussian values.

```
function [params,state] = initializeClassificationMLP(inputChannelSize,hiddenChannelSize,numClasses)
[params,state] = initializeSharedMLP(inputChannelSize,hiddenChannelSize);

weights = initializeWeightsGaussian([numClasses hiddenChannelSize(end)]);
bias = zeros(numClasses,1,"single");
params.FC.Weights = darray(weights);
params.FC.Bias = darray(bias);
end
```

initializeWeightsHe Function

The `initializeWeightsHe` function initializes parameters using He initialization.

```
function x = initializeWeightsHe(sz)
fanIn = prod(sz(1:3));
stddev = sqrt(2/fanIn);
x = stddev .* randn(sz);
end
```

initializeWeightsGaussian Function

The `initializeWeightsGaussian` function initializes parameters using Gaussian initialization with a standard deviation of 0.01.

```
function x = initializeWeightsGaussian(sz)
x = randn(sz,"single") .* 0.01;
end
```

Data Preprocessing Functions

preprocessPointCloudData Function

The `preprocessPointCloudData` function extracts the X, Y, Z point data from the input data and normalizes the data between 0 and 1. The function returns the normalized X, Y, Z data.

```
function data = preprocessPointCloud(data)

if ~iscell(data)
    data = {data};
end

numObservations = size(data,1);
for i = 1:numObservations
    % Scale points between 0 and 1.
    xlim = data{i,1}.XLimits;
    ylim = data{i,1}.YLimits;
    zlim = data{i,1}.ZLimits;

    xyzMin = [xlim(1) ylim(1) zlim(1)];
    xyzDiff = [diff(xlim) diff(ylim) diff(zlim)];

    data{i,1} = (data{i,1}.Location - xyzMin) ./ xyzDiff;
end
end
```

selectPoints Function

The `selectPoints` function samples the desired number of points. When the point cloud contains more than the desired number of points, the function uses `pcdsample` to randomly select points. Otherwise, the function replicates data to produce the desired number of points.

```
function data = selectPoints(data,numPoints)
% Select the desired number of points by downsampling or replicating
% point cloud data.
numObservations = size(data,1);
for i = 1:numObservations
    ptCloud = data{i,1};
```

```

if ptCloud.Count > numPoints
    percentage = numPoints/ptCloud.Count;
    data{i,1} = pcdownsampling(ptCloud,"random",percentage);
else
    replicationFactor = ceil(numPoints/ptCloud.Count);
    ind = repmat(1:ptCloud.Count,1,replicationFactor);
    data{i,1} = select(ptCloud,ind(1:numPoints));
end
end
end

```

Data Augmentation Functions

The `augmentPointCloudData` function randomly rotates a point cloud about the z-axis, randomly drops 30% of the points, and randomly jitters the point location with Gaussian noise.

```

function data = augmentPointCloud(data)

numObservations = size(data,1);
for i = 1:numObservations

    ptCloud = data{i,1};

    % Rotate the point cloud about "up axis", which is Z for this data set.
    tform = randomAffine3d(...
        "XReflection", true,...
        "YReflection", true,...
        "Rotation",@randomRotationAboutZ);

    ptCloud = pctransform(ptCloud,tform);

    % Randomly drop out 30% of the points.
    if rand > 0.5
        ptCloud = pcdownsampling(ptCloud,'random',0.3);
    end

    if rand > 0.5
        % Jitter the point locations with Gaussian noise with a mean of 0 and
        % a standard deviation of 0.02 by creating a random displacement field.
        D = 0.02 * randn(size(ptCloud.Location));
        ptCloud = pctransform(ptCloud,D);
    end

    data{i,1} = ptCloud;
end
end

function [rotationAxis,theta] = randomRotationAboutZ()
rotationAxis = [0 0 1];
theta = 2*pi*rand;
end

```

Supporting Functions

aggregateConfusionMetric Function

The `aggregateConfusionMetric` function incrementally fills a confusion matrix based on the predicted results `YPred` and the expected results `YTest`.

```
function cmat = aggregateConfusionMetric(cmat, YTest, YPred)
YTest = gather(extractdata(YTest));
YPred = gather(extractdata(YPred));
[m,n] = size(cmat);
cmat = cmat + full(sparse(YTest, YPred, 1, m, n));
end
```

initializeTrainingProgressPlot Function

The `initializeTrainingProgressPlot` function configures two plots for displaying the training loss, training accuracy, and validation accuracy.

```
function [plotter, trainAccPlotter, valAccPlotter] = initializeTrainingProgressPlot()
% Plot the loss, training accuracy, and validation accuracy.
figure

% Loss plot
subplot(2,1,1)
plotter = animatedline;
xlabel("Iteration")
ylabel("Loss")

% Accuracy plot
subplot(2,1,2)
trainAccPlotter = animatedline('Color','b');
valAccPlotter = animatedline('Color','g');
legend('Training Accuracy', 'Validation Accuracy', 'Location', 'northwest');
xlabel("Iteration")
ylabel("Accuracy")
end
```

replicateFiles Function

The `replicateFiles` function randomly oversamples a set of files and returns a set of files with `numDesired` elements.

```
function files = randReplicateFiles(files, numDesired)
n = numel(files);
ind = randi(n, numDesired, 1);
files = files(ind);
end
```

downloadSydneyUrban0jects Function

The `downloadSydneyUrban0jects` function downloads the data set and saves it to a temporary directory.

```
function datapath = downloadSydneyUrban0jects(dataLoc)

if nargin == 0
    dataLoc = pwd;
end

datapath = string(dataLoc);

url = "http://www.acfr.usyd.edu.au/papers/data/";
name = "sydney-urban-objects-dataset.tar.gz";
```



```

datapath = fullfile(dataLoc, 'sydney-urban-objects-dataset');
if ~exist(datapath, 'dir')
    disp('Downloading Sydney Urban Objects data set...');
    untar(url+name, dataLoc);
end

end

```

batchData Function

The `batchData` function collates input the data read from the `sydneyUrbanObjectsClassificationDatastore` into batches and moves data to the GPU for processing.

```

function [dLX,dLY] = batchData(data)
X = cat(4,data{:},1);
labels = cat(1,data{:},2);
Y = oneHotEncode(labels);

% Cast data to single for processing.
X = single(X);
Y = single(Y);

% Move data to the GPU if possible.
if canUseGPU
    X = gpuArray(X);
    Y = gpuArray(Y);
end

% Return X and Y as dlarray objects.
dLX = dlarray(X, 'SCSB');
dLY = dlarray(Y, 'CB');
end

```

oneHotEncode Function

The `oneHotEncode` function encodes categorical labels into a one-hot numeric vector.

```

function Y = oneHotEncode(labels)
numObservations = numel(labels);
numCategories = numel(categories(labels));
sz = [numCategories, numObservations];
Y = zeros(sz, 'single');
labels = labels';
idx = sub2ind(sz, int32(labels), 1:numObservations);
Y(idx) = 1;
end

```

prepareForPrediction Function

The `prepareForPrediction` function is used to apply a user-defined function to nested structure data. It is used to move model parameter and state data to the GPU.

```

function p = prepareForPrediction(p, fcn)

for i = 1:numel(p)
    p(i) = structfun(@(x)invoke(fcn,x), p(i), 'UniformOutput', 0);
end

```

```
function data = invoke(fcn,data)
    if isstruct(data)
        data = prepareForPrediction(data,fcn);
    else
        data = fcn(data);
    end
end
end

% Move data to the GPU.
function x = toDevice(x,useGPU)
if useGPU
    x = gpuArray(x);
end
end
```

References

- [1] Charles, R. Qi, Hao Su, Mo Kaichun, and Leonidas J. Guibas. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation." In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 77-85. Honolulu, HI: IEEE, 2017. <https://doi.org/10.1109/CVPR.2017.16>.
- [2] de Deuge, Mark, Alastair Quadras, Calvin Hung, and Bertrand Douillard. "Unsupervised Feature Learning for Classification of Outdoor 3D Scans." In *Australasian Conference on Robotics and Automation 2013 (ACRA 13)*. Sydney, Australia: ACRA, 2013.
- [3] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *2015 IEEE International Conference on Computer Vision (ICCV)*, 1026-34. Santiago, Chile: IEEE, 2015. <https://doi.org/10.1109/ICCV.2015.123>.

See Also

More About

- "Getting Started with Point Clouds Using Deep Learning" on page 9-2
- "Define Custom Training Loops, Loss Functions, and Networks" (Deep Learning Toolbox)
- "Specify Training Options in Custom Training Loop" (Deep Learning Toolbox)
- "Train Network Using Custom Training Loop" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)
- "Deep Learning Tips and Tricks" (Deep Learning Toolbox)
- "Automatic Differentiation Background" (Deep Learning Toolbox)

Object Detection Using SSD Deep Learning

This example shows how to train a Single Shot Detector (SSD).

Overview

Deep learning is a powerful machine learning technique that automatically learns image features required for detection tasks. There are several techniques for object detection using deep learning such as Faster R-CNN, You Only Look Once (YOLO v2), and SSD. This example trains an SSD vehicle detector using the `trainSSDObjectDetector` function. For more information, see “Object Detection using Deep Learning”.

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('ssdResNet50VehicleExample_20a.mat','file')
    disp('Downloading pretrained detector (44 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/ssdResNet50VehicleExample_20a.mat';
    websave('ssdResNet50VehicleExample_20a.mat',pretrainedURL);
end
```

Downloading pretrained detector (44 MB)...

Load Dataset

This example uses a small vehicle data set that contains 295 images. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the SSD training procedure, but in practice, more labeled images are needed to train a robust detector.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The training data is stored in a table. The first column contains the path to the image files. The remaining columns contain the ROI labels for vehicles. Display the first few rows of the data.

```
vehicleDataset(1:4,:)
```

ans=4x2 table

imageFilename	vehicle
'vehicleImages/image_00001.jpg'	{1x4 double}
'vehicleImages/image_00002.jpg'	{1x4 double}
'vehicleImages/image_00003.jpg'	{1x4 double}
'vehicleImages/image_00004.jpg'	{1x4 double}

Split the data set into a training set for training the detector and a test set for evaluating the detector. Select 60% of the data for training. Use the rest for evaluation.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices) );
```

```
trainingData = vehicleDataset(shuffledIndices(1:idx),:);  
testData = vehicleDataset(shuffledIndices(idx+1:end),:);
```

Use `imageDatastore` and `boxLabelDatastore` to load the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingData{:, 'imageFilename'});  
blsTrain = boxLabelDatastore(trainingData(:, 'vehicle'));  
  
imdsTest = imageDatastore(testData{:, 'imageFilename'});  
blsTest = boxLabelDatastore(testData(:, 'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,blsTrain);  
testData = combine(imdsTest, blsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);  
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I, 'Rectangle', bbox);  
annotatedImage = imresize(annotatedImage,2);  
figure  
imshow(annotatedImage)
```



Create a SSD Object Detection Network

The SSD object detection network can be thought of as having two sub-networks. A feature extraction network, followed by a detection network.

The feature extraction network is typically a pretrained CNN (see “Pretrained Deep Neural Networks” (Deep Learning Toolbox) for more details). This example uses ResNet-50 for feature extraction. Other pretrained networks such as MobileNet v2 or ResNet-18 can also be used depending on application requirements. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to SSD.

Use the `ssdLayers` function to automatically modify a pretrained ResNet-50 network into a SSD object detection network. `ssdLayers` requires you to specify several inputs that parameterize the SSD network, including the network input size and the number of classes. When choosing the network input size, consider the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image. However, to reduce the computational cost of running this example, the network input size is chosen to be [300 300 3]. During training, `trainSSDObjectDetector` automatically resizes the training images to the network input size.

```
inputSize = [300 300 3];
```

Define number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the SSD object detection network.

```
lgraph = ssdLayers(inputSize, numClasses, 'resnet50');
```

You can visualize the network using `analyzeNetwork` or `DeepNetworkDesigner` from Deep Learning Toolbox™. Note that you can also create a custom SSD network layer-by-layer. For more information, see “Create SSD Object Detection Network” on page 3-115.

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples. Use `transform` to augment the training data by

- Randomly flipping the image and associated box labels horizontally.
- Randomly scale the image, associated box labels.
- Jitter image color.

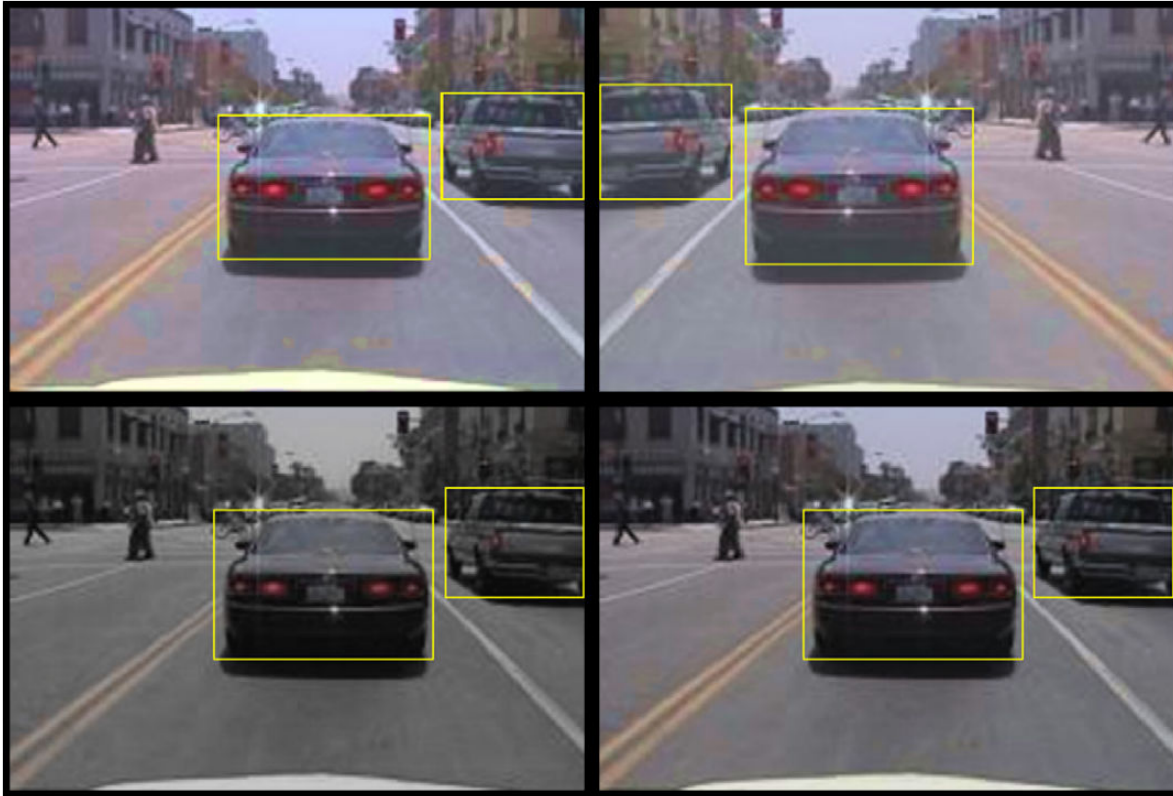
Note that data augmentation is not applied to the test data. Ideally, test data should be representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Visualize augmented training data by reading the same image multiple times.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1}, 'Rectangle', data{2});
```

```
    reset(augmentedTrainingData);  
end  
  
figure  
montage(augmentedData, 'BorderSize', 10)
```



Preprocess Training Data

Preprocess the augmented training data to prepare for training.

```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image and bounding boxes.

```
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I, 'Rectangle', bbox);  
annotatedImage = imresize(annotatedImage, 2);  
figure  
imshow(annotatedImage)
```



Train SSD Object Detector

Use `trainingOptions` to specify network training options. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...  
    'MiniBatchSize', 16, ...  
    'InitialLearnRate', 1e-1, ...  
    'LearnRateSchedule', 'piecewise', ...  
    'LearnRateDropPeriod', 30, ...  
    'LearnRateDropFactor', 0.8, ...  
    'MaxEpochs', 300, ...
```

```
    'VerboseFrequency', 50, ...  
    'CheckpointPath', tempdir, ...  
    'Shuffle', 'every-epoch');
```

Use `trainSSDObjectDetector` function to train SSD object detector if `doTraining` to true. Otherwise, load a pretrained network.

```
if doTraining  
    % Train the SSD detector.  
    [detector, info] = trainSSDObjectDetector(preprocessedTrainingData, lgraph, options);  
else  
    % Load pretrained detector for the example.  
    pretrained = load('ssdResNet50VehicleExample_20a.mat');  
    detector = pretrained.detector;  
end
```

This example is verified on an NVIDIA™ Titan X GPU with 12 GB of memory. If your GPU has less memory, you may run out of memory. If this happens, lower the `'MiniBatchSize'` using the `trainingOptions` function. Training this network took approximately 2 hours using this setup. Training time varies depending on the hardware you use.

As a quick test, run the detector on one test image.

```
data = read(testData);  
I = data{1,1};  
I = imresize(I, inputSize(1:2));  
[bboxes, scores] = detect(detector, I, 'Threshold', 0.4);
```

Display the results.

```
I = insertObjectAnnotation(I, 'rectangle', bboxes, scores);  
figure  
imshow(I)
```




Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (`precision`) and the ability of the detector to find all relevant objects (`recall`).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Test data should be representative of the original data and be left unmodified for unbiased evaluation.

```
preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

```
detectionResults = detect(detector, preprocessedTestData, 'Threshold', 0.4);
```

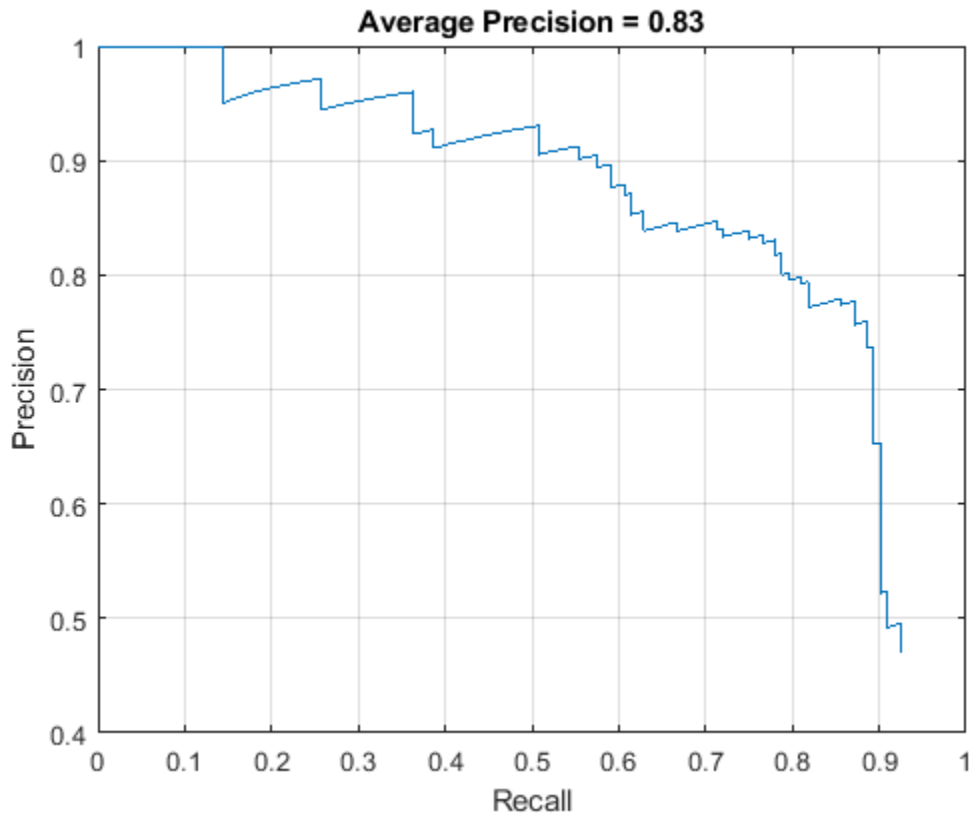
Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults, preprocessedTestData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. Ideally, the precision would be 1 at all recall levels. The use of more data can help improve the average precision, but might require more training time Plot the PR curve.

```
figure
plot(recall,precision)
```

```
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```



Code Generation

Once the detector is trained and evaluated, you can generate code for the `ssdObjectDetector` using GPU Coder™. For more details, see “Code Generation for Object Detection by Using Single Shot Multibox Detector” on page 2-2 example.

Supporting Functions

```
function B = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.
B = cell(size(A));

I = A{1};
sz = size(I);
if numel(sz)==3 && sz(3) == 3
    I = jitterColorHSV(I,...
        'Contrast',0.2,...
        'Hue',0,...
        'Saturation',0.1,...
        'Brightness',0.2);
end
```

```

% Randomly flip and scale image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');
B{1} = imwarp(I,tform,'OutputView',rout);

% Apply same transform to boxes.
[B{2},indices] = bboxwarp(A{2},tform,rout,'OverlapThreshold',0.25);
B{3} = A{3}(indices);

% Return original data only when all boxes are removed by warping.
if isempty(indices)
    B = A;
end
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to the targetSize.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
data{2} = bboxresize(data{2},scale);
end

```

References

[1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single shot multibox detector." In 14th European Conference on Computer Vision, ECCV 2016. Springer Verlag, 2016.

See Also

Apps

Deep Network Designer

Functions

analyzeNetwork | combine | estimateAnchorBoxes | evaluateDetectionPrecision | read | transform

Objects

boxLabelDatastore | imageDatastore

More About

- "Anchor Boxes for Object Detection" on page 14-21
- "Estimate Anchor Boxes From Training Data" on page 3-146
- "Transfer Learning with Deep Network Designer" (Deep Learning Toolbox)
- "Getting Started with Object Detection Using Deep Learning" on page 14-13

Object Detection in a Cluttered Scene Using Point Feature Matching

This example shows how to detect a particular object in a cluttered scene, given a reference image of the object.

Overview

This example presents an algorithm for detecting a specific object based on finding point correspondences between the reference and the target image. It can detect objects despite a scale change or in-plane rotation. It is also robust to small amount of out-of-plane rotation and occlusion.

This method of object detection works best for objects that exhibit non-repeating texture patterns, which give rise to unique feature matches. This technique is not likely to work well for uniformly-colored objects, or for objects containing repeating patterns. Note that this algorithm is designed for detecting a specific object, for example, the elephant in the reference image, rather than any elephant. For detecting objects of a particular category, such as people or faces, see `vision.PeopleDetector` and `vision.CascadeObjectDetector`.

Step 1: Read Images

Read the reference image containing the object of interest.

```
boxImage = imread('stapleRemover.jpg');  
figure;  
imshow(boxImage);  
title('Image of a Box');
```

Image of a Box



Read the target image containing a cluttered scene.

```
sceneImage = imread('clutteredDesk.jpg');  
figure;  
imshow(sceneImage);  
title('Image of a Cluttered Scene');
```

Image of a Cluttered Scene



Step 2: Detect Feature Points

Detect feature points in both images.

```
boxPoints = detectSURFFeatures(boxImage);  
scenePoints = detectSURFFeatures(sceneImage);
```

Visualize the strongest feature points found in the reference image.

```
figure;  
imshow(boxImage);  
title('100 Strongest Feature Points from Box Image');  
hold on;  
plot(selectStrongest(boxPoints, 100));
```

100 Strongest Feature Points from Box Image



Visualize the strongest feature points found in the target image.

```
figure;  
imshow(sceneImage);  
title('300 Strongest Feature Points from Scene Image');  
hold on;  
plot(selectStrongest(scenePoints, 300));
```



Step 3: Extract Feature Descriptors

Extract feature descriptors at the interest points in both images.

```
[boxFeatures, boxPoints] = extractFeatures(boxImage, boxPoints);  
[sceneFeatures, scenePoints] = extractFeatures(sceneImage, scenePoints);
```

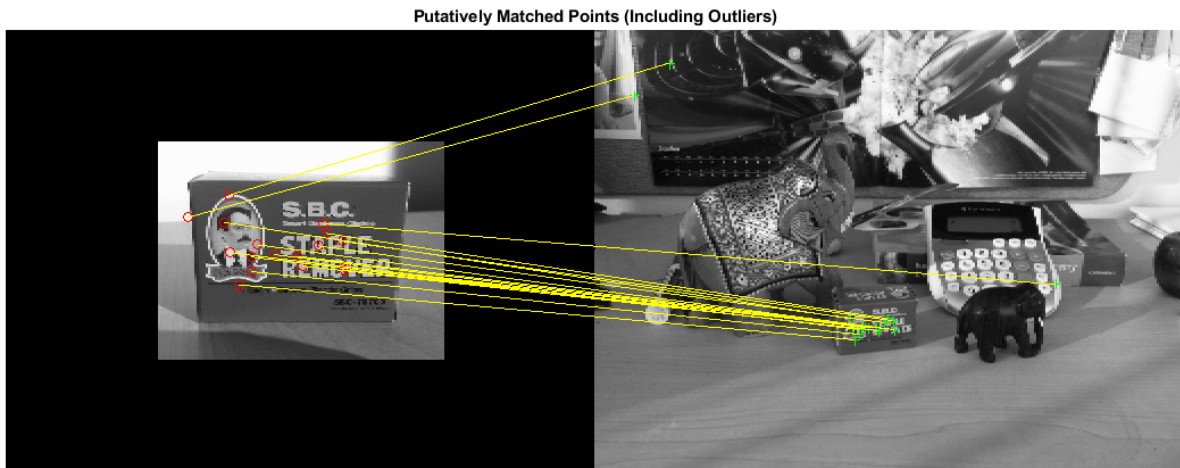
Step 4: Find Putative Point Matches

Match the features using their descriptors.

```
boxPairs = matchFeatures(boxFeatures, sceneFeatures);
```

Display putatively matched features.

```
matchedBoxPoints = boxPoints(boxPairs(:, 1), :);  
matchedScenePoints = scenePoints(boxPairs(:, 2), :);  
figure;  
showMatchedFeatures(boxImage, sceneImage, matchedBoxPoints, ...  
    matchedScenePoints, 'montage');  
title('Putatively Matched Points (Including Outliers)');
```

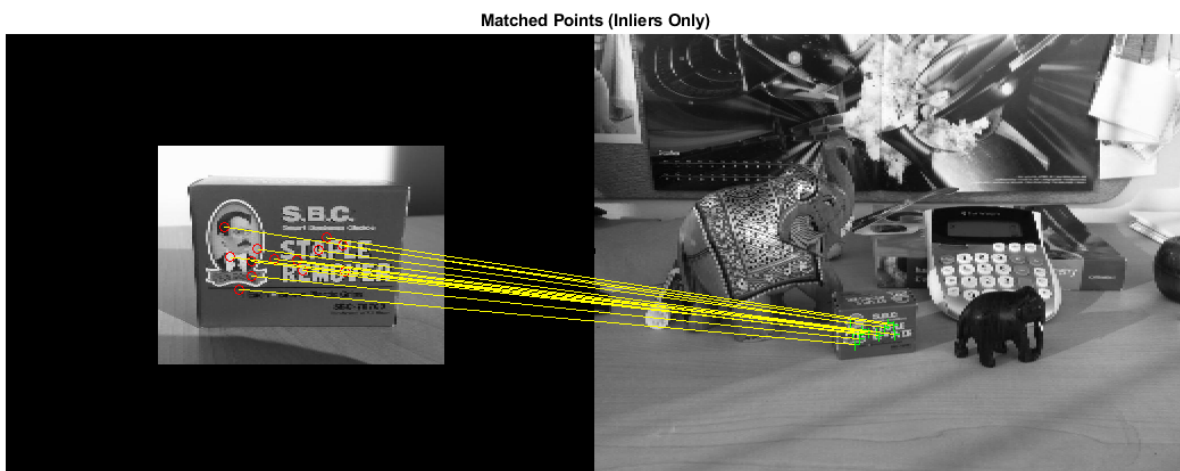
Step 5: Locate the Object in the Scene Using Putative Matches

`estimateGeometricTransform2D` calculates the transformation relating the matched points, while eliminating outliers. This transformation allows us to localize the object in the scene.

```
[tform, inlierIdx] = ...
    estimateGeometricTransform2D(matchedBoxPoints, matchedScenePoints, 'affine');
inlierBoxPoints = matchedBoxPoints(inlierIdx, :);
inlierScenePoints = matchedScenePoints(inlierIdx, :);
```

Display the matching point pairs with the outliers removed

```
figure;
showMatchedFeatures(boxImage, sceneImage, inlierBoxPoints, ...
    inlierScenePoints, 'montage');
title('Matched Points (Inliers Only)');
```



Get the bounding polygon of the reference image.

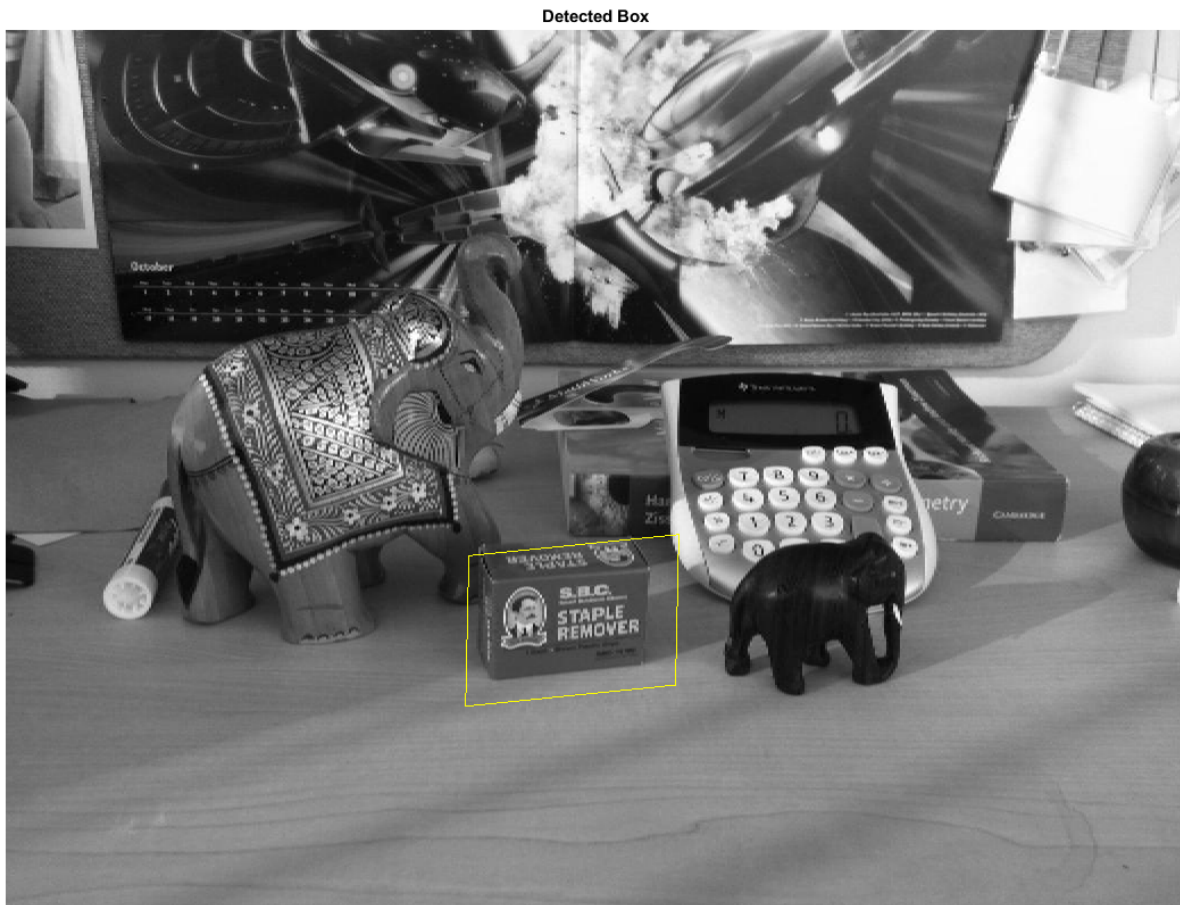
```
boxPolygon = [1, 1;... % top-left
              size(boxImage, 2), 1;... % top-right
              size(boxImage, 2), size(boxImage, 1);... % bottom-right
              1, size(boxImage, 1);... % bottom-left
              1, 1]; % top-left again to close the polygon
```

Transform the polygon into the coordinate system of the target image. The transformed polygon indicates the location of the object in the scene.

```
newBoxPolygon = transformPointsForward(tform, boxPolygon);
```

Display the detected object.

```
figure;
imshow(sceneImage);
hold on;
line(newBoxPolygon(:, 1), newBoxPolygon(:, 2), 'Color', 'y');
title('Detected Box');
```



Step 7: Detect Another Object

Detect a second object by using the same steps as before.

Read an image containing the second object of interest.

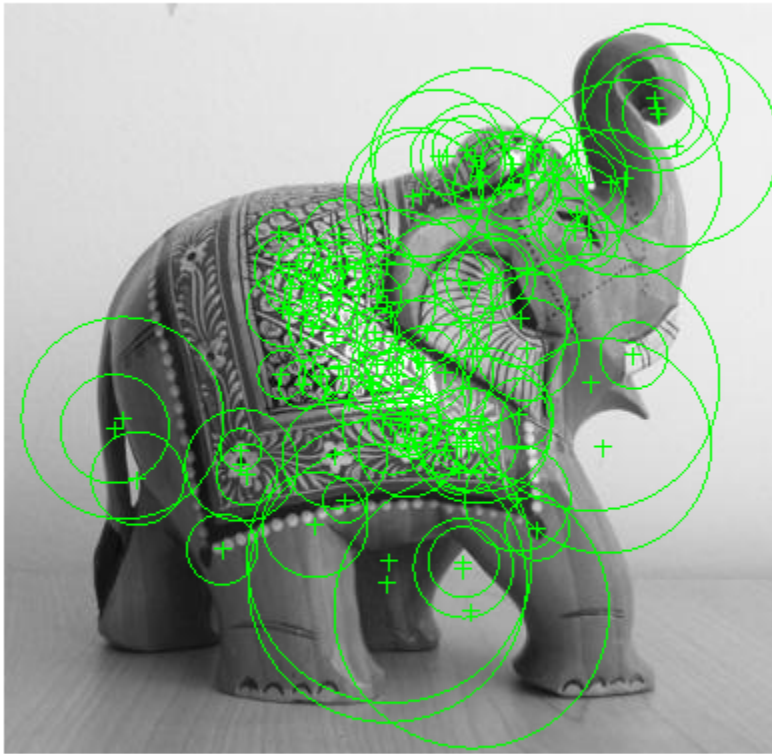
```
elephantImage = imread('elephant.jpg');  
figure;  
imshow(elephantImage);  
title('Image of an Elephant');
```

Image of an Elephant



Detect and visualize feature points.

```
elephantPoints = detectSURFFeatures(elephantImage);  
figure;  
imshow(elephantImage);  
hold on;  
plot(selectStrongest(elephantPoints, 100));  
title('100 Strongest Feature Points from Elephant Image');
```

100 Strongest Feature Points from Elephant Image

Extract feature descriptors.

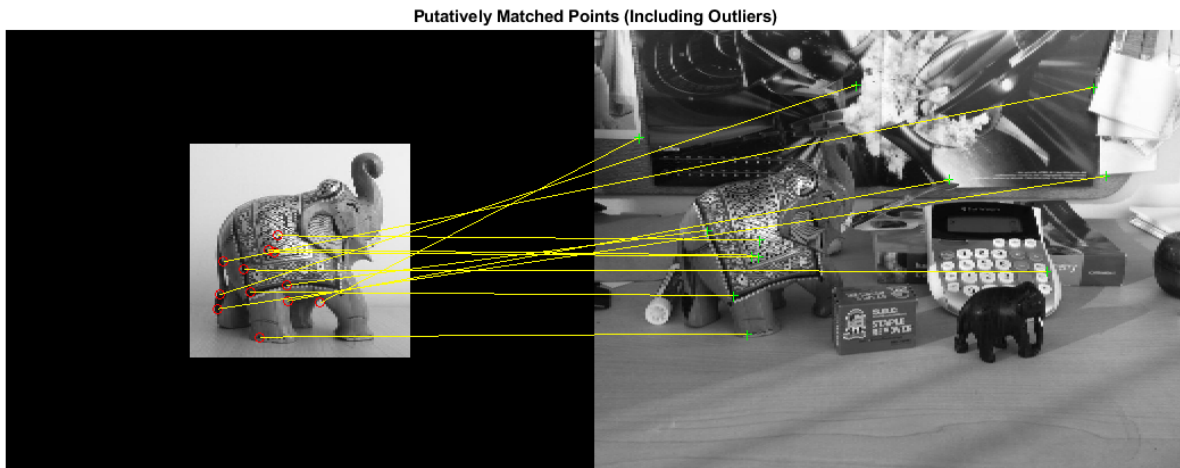
```
[elephantFeatures, elephantPoints] = extractFeatures(elephantImage, elephantPoints);
```

Match Features

```
elephantPairs = matchFeatures(elephantFeatures, sceneFeatures, 'MaxRatio', 0.9);
```

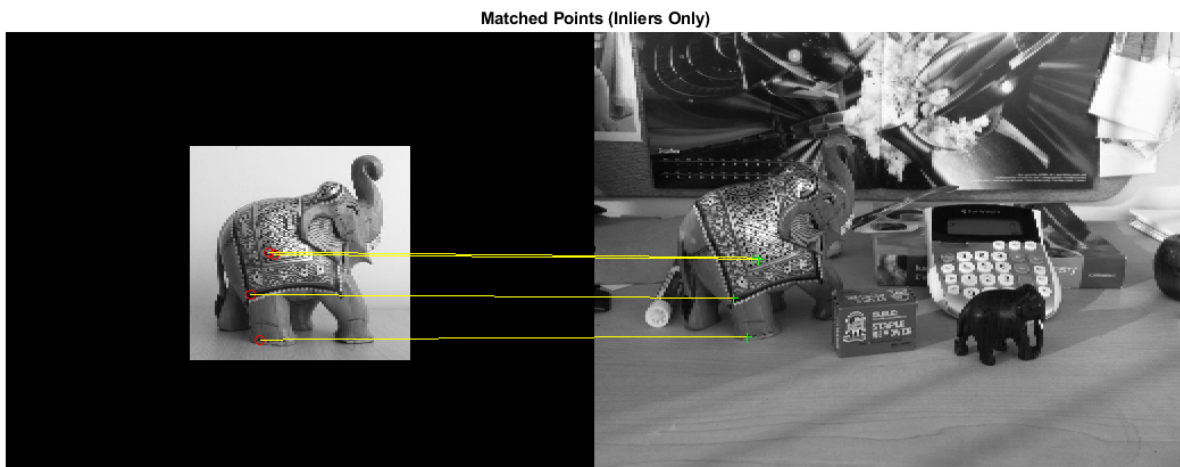
Display putatively matched features.

```
matchedElephantPoints = elephantPoints(elephantPairs(:, 1), :);  
matchedScenePoints = scenePoints(elephantPairs(:, 2), :);  
figure;  
showMatchedFeatures(elephantImage, sceneImage, matchedElephantPoints, ...  
    matchedScenePoints, 'montage');  
title('Putatively Matched Points (Including Outliers)');
```



Estimate Geometric Transformation and Eliminate Outliers

```
[tform, inlierElephantPoints, inlierScenePoints] = ...
    estimateGeometricTransform(matchedElephantPoints, matchedScenePoints, 'affine');
figure;
showMatchedFeatures(elephantImage, sceneImage, inlierElephantPoints, ...
    inlierScenePoints, 'montage');
title('Matched Points (Inliers Only)');
```



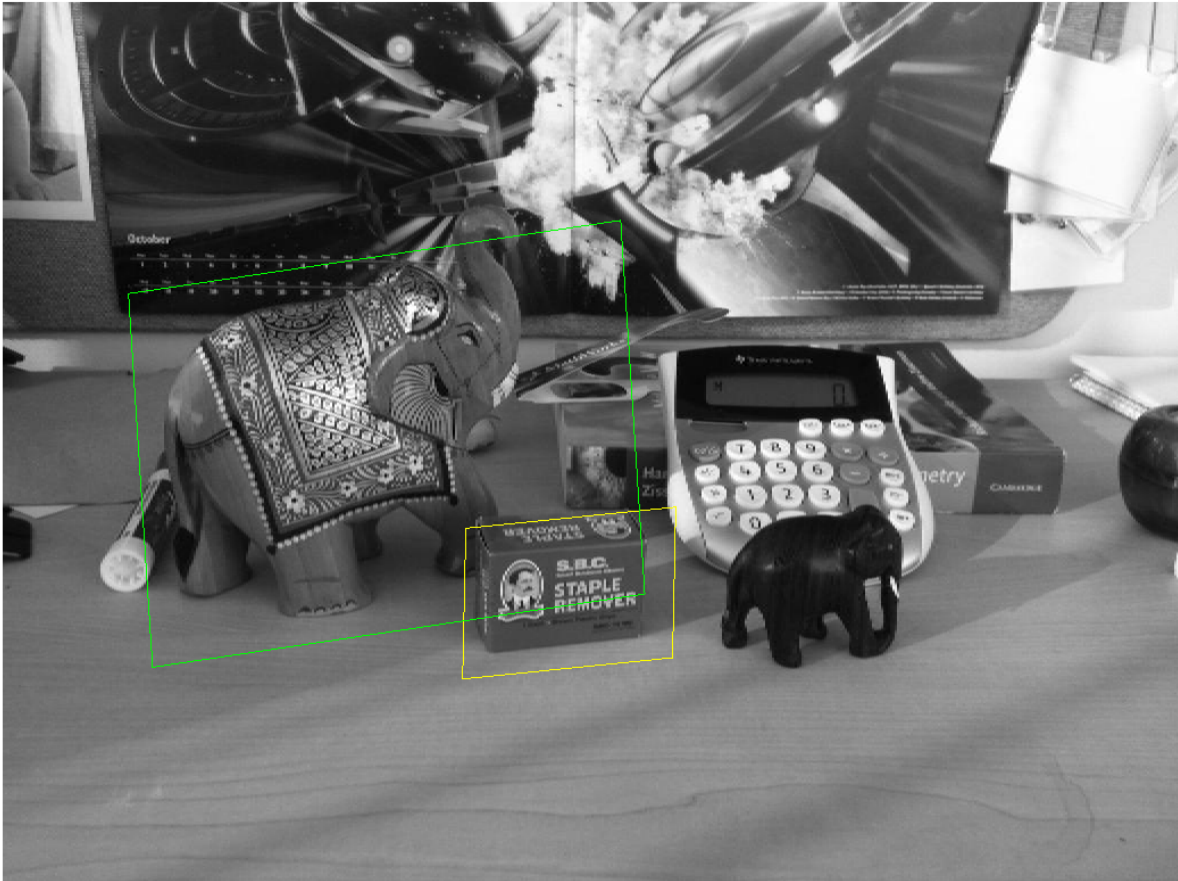
Display Both Objects

```
elephantPolygon = [1, 1;... % top-left
    size(elephantImage, 2), 1;... % top-right
    size(elephantImage, 2), size(elephantImage, 1);... % bottom-right
    1, size(elephantImage, 1);... % bottom-left
    1,1]; % top-left again to close the polygon

newElephantPolygon = transformPointsForward(tform, elephantPolygon);
```

```
figure;  
imshow(sceneImage);  
hold on;  
line(newBoxPolygon(:, 1), newBoxPolygon(:, 2), 'Color', 'y');  
line(newElephantPolygon(:, 1), newElephantPolygon(:, 2), 'Color', 'g');  
title('Detected Elephant and Box');
```

Detected Elephant and Box



Semantic Segmentation Using Deep Learning

This example shows how to train a semantic segmentation network using deep learning.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” on page 14-43.

To illustrate the training procedure, this example trains Deeplab v3+ [1], one type of convolutional neural network (CNN) designed for semantic image segmentation. Other types of networks for semantic segmentation include fully convolutional networks (FCN), SegNet, and U-Net. The training procedure shown here can be applied to those networks too.

This example uses the CamVid dataset [2] from the University of Cambridge for training. This 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.

Setup

This example creates the Deeplab v3+ network with weights initialized from a pre-trained Resnet-18 network. ResNet-18 is an efficient network that is well suited for applications with limited processing resources. Other pretrained networks such as MobileNet v2 or ResNet-50 can also be used depending on application requirements. For more details, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

To get a pretrained Resnet-18, install Deep Learning Toolbox™ Model for Resnet-18 Network. After installation is complete, run the following code to verify that the installation is correct.

```
resnet18();
```

In addition, download a pretrained version of DeepLab v3+. The pretrained model allows you to run the entire example without having to wait for training to complete.

```
pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/deeplabv3plusResnet18CamVid.m';
pretrainedFolder = fullfile(tempdir, 'pretrainedNetwork');
pretrainedNetwork = fullfile(pretrainedFolder, 'deeplabv3plusResnet18CamVid.mat');
if ~exist(pretrainedNetwork, 'file')
    mkdir(pretrainedFolder);
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetwork, pretrainedURL);
end
```

A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for running this example. Use of a GPU requires Parallel Computing Toolbox™.

Download CamVid Dataset

Download the CamVid dataset from the following URLs.

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.zip';
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.zip';

outputFolder = fullfile(tempdir, 'CamVid');
labelsZip = fullfile(outputFolder, 'labels.zip');
imagesZip = fullfile(outputFolder, 'images.zip');
```

```
if ~exist(labelsZip, 'file') || ~exist(imagesZip, 'file')
    mkdir(outputFolder)

    disp('Downloading 16 MB CamVid dataset labels...');
    websave(labelsZip, labelURL);
    unzip(labelsZip, fullfile(outputFolder, 'labels'));

    disp('Downloading 557 MB CamVid dataset images...');
    websave(imagesZip, imageURL);
    unzip(imagesZip, fullfile(outputFolder, 'images'));
end
```

Note: Download time of the data depends on your Internet connection. The commands used above 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.

Load CamVid Images

Use `imageDatastore` to load CamVid images. The `imageDatastore` enables you to efficiently load a large collection of images on disk.

```
imgDir = fullfile(outputFolder, 'images', '701_StillsRaw_full');
imds = imageDatastore(imgDir);
```

Display one of the images.

```
I = readimage(imds, 559);
I = histeq(I);
imshow(I)
```




Load CamVid Pixel-Labeled Images

Use `pixelLabelDatastore` to load CamVid pixel label image data. A `pixelLabelDatastore` encapsulates the pixel label data and the label ID to a class name mapping.

We make training easier, we group the 32 original classes in CamVid to 11 classes. Specify these classes.

```
classes = [  
    "Sky"  
    "Building"  
    "Pole"  
    "Road"  
    "Pavement"  
    "Tree"  
    "SignSymbol"  
    "Fence"  
    "Car"  
    "Pedestrian"  
    "Bicyclist"  
];
```

To reduce 32 classes into 11, multiple classes from the original dataset are grouped together. For example, "Car" is a combination of "Car", "SUVPickupTruck", "Truck_Bus", "Train", and "OtherMoving". Return the grouped label IDs by using the supporting function `camvidPixelLabelIDs`, which is listed at the end of this example.

```
labelIDs = camvidPixelLabelIDs();
```

Use the classes and label IDs to create the `pixelLabelDatastore`.

```
labelDir = fullfile(outputFolder,'labels');  
pxds = pixelLabelDatastore(labelDir,classes,labelIDs);
```

Read and display one of the pixel-labeled images by overlaying it on top of an image.

```
C = readimage(pxds,559);  
cmap = camvidColorMap;  
B = labeloverlay(I,C,'ColorMap',cmap);  
imshow(B)  
pixelLabelColorbar(cmap,classes);
```



Areas with no color overlay do not have pixel labels and are not used during training.

Analyze Dataset Statistics

To see the distribution of class labels in the CamVid dataset, use `countEachLabel`. This function counts the number of pixels by class label.

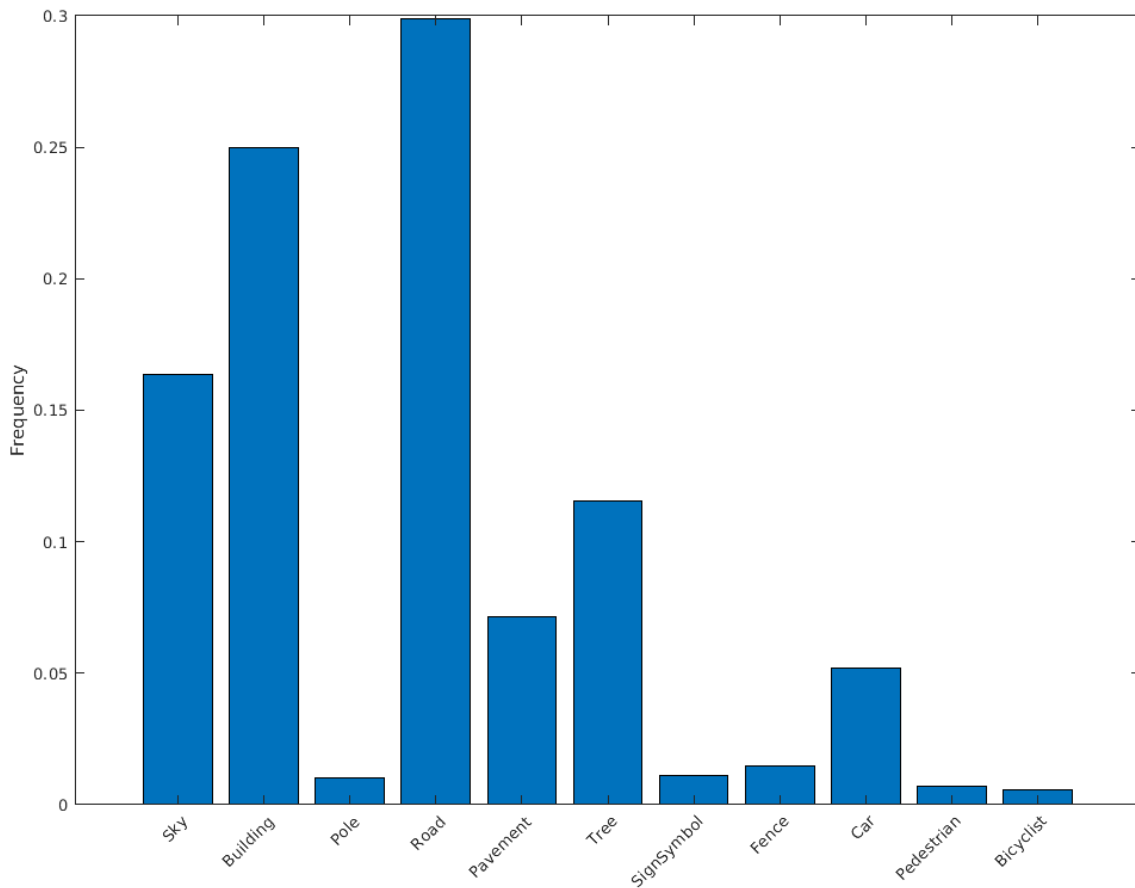
```
tbl = countEachLabel(pxds)
```

```
tbl=11x3 table
      Name          PixelCount  ImagePixelCount
-----
{'Sky'           } 7.6801e+07  4.8315e+08
{'Building'      } 1.1737e+08  4.8315e+08
{'Pole'          } 4.7987e+06  4.8315e+08
{'Road'          } 1.4054e+08  4.8453e+08
{'Pavement'      } 3.3614e+07  4.7209e+08
{'Tree'          } 5.4259e+07  4.479e+08
{'SignSymbol'    } 5.2242e+06  4.6863e+08
{'Fence'         } 6.9211e+06  2.516e+08
{'Car'           } 2.4437e+07  4.8315e+08
{'Pedestrian'    } 3.4029e+06  4.4444e+08
{'Bicyclist'     } 2.5912e+06  2.6196e+08
```

Visualize the pixel counts by class.

```
frequency = tbl.PixelCount/sum(tbl.PixelCount);

bar(1:numel(classes), frequency)
xticks(1:numel(classes))
xticklabels(tbl.Name)
xtickangle(45)
ylabel('Frequency')
```



Ideally, all classes would have an equal number of observations. However, the classes in CamVid are imbalanced, which is a common issue in automotive data-sets of street scenes. Such scenes have more sky, building, and road pixels than pedestrian and bicyclist pixels because sky, buildings and roads cover more area in the image. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes. Later on in this example, you will use class weighting to handle this issue.

The images in the CamVid data set are 720 by 960 in size. Image size is chosen such that a large enough batch of images can fit in memory during training on an NVIDIA™ Titan X with 12 GB of memory. You may need to resize the images to smaller sizes if your GPU does not have sufficient memory or reduce the training batch size.

Prepare Training, Validation, and Test Sets

Deeplab v3+ is trained using 60% of the images from the dataset. The rest of the images are split evenly in 20% and 20% for validation and testing respectively. The following code randomly splits the image and pixel label data into a training, validation and test set.

```
[imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = partitionCamVidData(imds,pxds);
```

The 60/20/20 split results in the following number of training, validation and test images:

```

numTrainingImages = numel(imdsTrain.Files)
numTrainingImages = 421
numValImages = numel(imdsVal.Files)
numValImages = 140
numTestingImages = numel(imdsTest.Files)
numTestingImages = 140

```

Create the Network

Use the `deeplabv3plusLayers` function to create a DeepLab v3+ network based on ResNet-18. Choosing the best network for your application requires empirical analysis and is another level of hyperparameter tuning. For example, you can experiment with different base networks such as ResNet-50 or MobileNet v2, or you can try other semantic segmentation network architectures such as SegNet, fully convolutional networks (FCN), or U-Net.

```

% Specify the network image size. This is typically the same as the traing image sizes.
imageSize = [720 960 3];

% Specify the number of classes.
numClasses = numel(classes);

% Create DeepLab v3+.
lgraph = deeplabv3plusLayers(imageSize, numClasses, "resnet18");

```

Balance Classes Using Class Weighting

As shown earlier, the classes in CamVid are not balanced. To improve training, you can use class weighting to balance the classes. Use the pixel label counts computed earlier with `countEachLabel` and calculate the median frequency class weights.

```

imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq

classWeights = 11x1

    0.3182
    0.2082
    5.0924
    0.1744
    0.7103
    0.4175
    4.5371
    1.8386
    1.0000
    6.6059
    :

```

Specify the class weights using a `pixelClassificationLayer`.

```

pxLayer = pixelClassificationLayer('Name','labels','Classes',tbl.Name,'ClassWeights',classWeights);
lgraph = replaceLayer(lgraph,"classification",pxLayer);

```

Select Training Options

The optimization algorithm used for training is stochastic gradient descent with momentum (SGDM). Use `trainingOptions` (Deep Learning Toolbox) to specify the hyper-parameters used for SGDM.

```
% Define validation data.
pximdsVal = pixelLabelImageDatastore(imdsVal,pxdsVal);

% Define training options.
options = trainingOptions('sgdm', ...
    'LearnRateSchedule','piecewise',...
    'LearnRateDropPeriod',10,...
    'LearnRateDropFactor',0.3,...
    'Momentum',0.9, ...
    'InitialLearnRate',1e-3, ...
    'L2Regularization',0.005, ...
    'ValidationData',pximdsVal,...
    'MaxEpochs',30, ...
    'MiniBatchSize',8, ...
    'Shuffle','every-epoch', ...
    'CheckpointPath', tempdir, ...
    'VerboseFrequency',2,...
    'Plots','training-progress',...
    'ValidationPatience', 4);
```

The learning rate uses a piecewise schedule. The learning rate is reduced by a factor of 0.3 every 10 epochs. This allows the network to learn quickly with a higher initial learning rate, while being able to find a solution close to the local optimum once the learning rate drops.

The network is tested against the validation data every epoch by setting the `'ValidationData'` parameter. The `'ValidationPatience'` is set to 4 to stop training early when the validation accuracy converges. This prevents the network from overfitting on the training dataset.

A mini-batch size of 8 is used to reduce memory usage while training. You can increase or decrease this value based on the amount of GPU memory you have on your system.

In addition, `'CheckpointPath'` is set to a temporary location. This name-value pair enables the saving of network checkpoints at the end of every training epoch. If training is interrupted due to a system failure or power outage, you can resume training from the saved checkpoint. Make sure that the location specified by `'CheckpointPath'` has enough space to store the network checkpoints. For example, saving 100 Deeplab v3+ checkpoints requires ~6 GB of disk space because each checkpoint is 61 MB.

Data Augmentation

Data augmentation is used during training to provide more examples to the network because it helps improve the accuracy of the network. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation. Use the `imageDataAugmenter` (Deep Learning Toolbox) to specify these data augmentation parameters.

```
augmenter = imageDataAugmenter('RandXReflection',true,...
    'RandXTranslation',[-10 10],'RandYTranslation',[-10 10]);
```

`imageDataAugmenter` supports several other types of data augmentation. Choosing among them requires empirical analysis and is another level of hyper-parameter tuning.

Start Training

Combine the training data and data augmentation selections using `pixelLabelImageDatastore`. The `pixelLabelImageDatastore` reads batches of training data, applies data augmentation, and sends the augmented data to the training algorithm.

```
pximds = pixelLabelImageDatastore(imdsTrain,pxdsTrain, ...
    'DataAugmentation',augmenter);
```

Start training using `trainNetwork` (Deep Learning Toolbox) if the `doTraining` flag is true. Otherwise, load a pretrained network.

Note: The training was verified on an NVIDIA™ Titan X with 12 GB of GPU memory. If your GPU has less memory, you may run out of memory during training. If this happens, try setting `'MiniBatchSize'` to 1 in `trainingOptions`, or reducing the network input and resizing the training data using the `'OutputSize'` parameter of `pixelLabelImageDatastore`. Training this network takes about 5 hours. Depending on your GPU hardware, it can take even longer.

```
doTraining = false;
if doTraining
    [net, info] = trainNetwork(pximds,lgraph,options);
else
    data = load(pretrainedNetwork);
    net = data.net;
end
```

Test Network on One Image

As a quick sanity check, run the trained network on one test image.

```
I = readimage(imdsTest,35);
C = semanticseg(I, net);
```

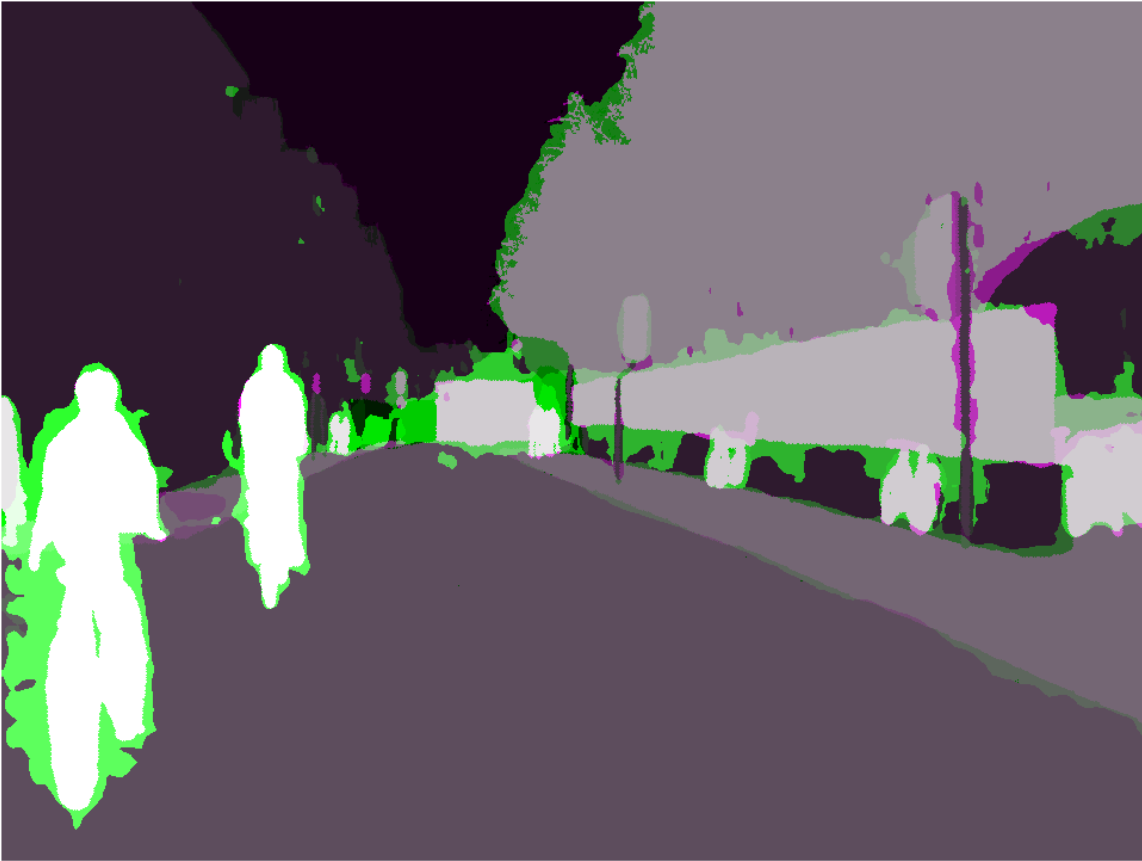
Display the results.

```
B = labeloverlay(I,C,'Colormap',cmap,'Transparency',0.4);
imshow(B)
pixelLabelColorbar(cmap, classes);
```



Compare the results in C with the expected ground truth stored in `pxdsTest`. The green and magenta regions highlight areas where the segmentation results differ from the expected ground truth.

```
expectedResult = readimage(pxdsTest,35);  
actual = uint8(C);  
expected = uint8(expectedResult);  
imshowpair(actual, expected)
```

Visually, the semantic segmentation results overlap well for classes such as road, sky, and building. However, smaller objects like pedestrians and cars are not as accurate. The amount of overlap per class can be measured using the intersection-over-union (IoU) metric, also known as the Jaccard index. Use the `jaccard` function to measure IoU.

```
iou = jaccard(C,expectedResult);
table(classes,iou)
```

```
ans=11x2 table
    classes      iou
    _____  _____
    "Sky"         0.91837
    "Building"    0.84479
    "Pole"        0.31203
    "Road"        0.93698
    "Pavement"    0.82838
    "Tree"        0.89636
    "SignSymbol"  0.57644
    "Fence"       0.71046
    "Car"         0.66688
    "Pedestrian"  0.48417
```

```
"Bicyclist"      0.68431
```

The IoU metric confirms the visual results. Road, sky, and building classes have high IoU scores, while classes such as pedestrian and car have low scores. Other common segmentation metrics include the `dice` and the `bfscore` contour matching score.

Evaluate Trained Network

To measure accuracy for multiple test images, run `semanticseg` on the entire test set. A mini-batch size of 4 is used to reduce memory usage while segmenting images. You can increase or decrease this value based on the amount of GPU memory you have on your system.

```
pxdsResults = semanticseg(imdsTest,net, ...
    'MiniBatchSize',4, ...
    'WriteLocation',tempdir, ...
    'Verbose',false);
```

`semanticseg` returns the results for the test set as a `pixelLabelDatastore` object. The actual pixel label data for each test image in `imdsTest` is written to disk in the location specified by the `'WriteLocation'` parameter. Use `evaluateSemanticSegmentation` to measure semantic segmentation metrics on the test set results.

```
metrics = evaluateSemanticSegmentation(pxdsResults,pxdsTest,'Verbose',false);
```

`evaluateSemanticSegmentation` returns various metrics for the entire dataset, for individual classes, and for each test image. To see the dataset level metrics, inspect `metrics.DataSetMetrics`.

```
metrics.DataSetMetrics
```

```
ans=1x5 table
   GlobalAccuracy   MeanAccuracy   MeanIoU   WeightedIoU   MeanBFScore
   _____   _____   _____   _____   _____
           0.87695           0.85392           0.6302           0.80851           0.65051
```

The dataset metrics provide a high-level overview of the network performance. To see the impact each class has on the overall performance, inspect the per-class metrics using `metrics.ClassMetrics`.

```
metrics.ClassMetrics
```

```
ans=11x3 table
           Accuracy   IoU   MeanBFScore
           _____   _____   _____
   Sky           0.93112   0.90209   0.8952
   Building       0.78453   0.76098   0.58511
   Pole           0.71586   0.21477   0.51439
   Road           0.93024   0.91465   0.76696
   Pavement       0.88466   0.70571   0.70919
   Tree           0.87377   0.76323   0.70875
   SignSymbol     0.79358   0.39309   0.48302
   Fence          0.81507   0.46484   0.48564
   Car            0.90956   0.76799   0.69233
   Pedestrian     0.87629   0.4366   0.60792
```

```
Bicyclist    0.87844    0.60829    0.55089
```

Although the overall dataset performance is quite high, the class metrics show that underrepresented classes such as Pedestrian, Bicyclist, and Car are not segmented as well as classes such as Road, Sky, and Building. Additional data that includes more samples of the underrepresented classes might help improve the results.

Supporting Functions

```
function labelIDs = camvidPixelLabelIDs()
% Return the label IDs corresponding to each class.
%
% The CamVid dataset has 32 classes. Group them into 11 classes following
% the original SegNet training methodology [1].
%
% The 11 classes are:
% "Sky" "Building", "Pole", "Road", "Pavement", "Tree", "SignSymbol",
% "Fence", "Car", "Pedestrian", and "Bicyclist".
%
% CamVid pixel label IDs are provided as RGB color values. Group them into
% 11 classes and return them as a cell array of M-by-3 matrices. The
% original CamVid class names are listed alongside each RGB value. Note
% that the Other/Void class are excluded below.
labelIDs = { ...

    % "Sky"
    [
    128 128 128; ... % "Sky"
    ]

    % "Building"
    [
    000 128 064; ... % "Bridge"
    128 000 000; ... % "Building"
    064 192 000; ... % "Wall"
    064 000 064; ... % "Tunnel"
    192 000 128; ... % "Archway"
    ]

    % "Pole"
    [
    192 192 128; ... % "Column_Pole"
    000 000 064; ... % "TrafficCone"
    ]

    % Road
    [
    128 064 128; ... % "Road"
    128 000 192; ... % "LaneMkgsDriv"
    192 000 064; ... % "LaneMkgsNonDriv"
    ]

    % "Pavement"
    [
    000 000 192; ... % "Sidewalk"
    064 192 128; ... % "ParkingBlock"
    128 128 192; ... % "RoadShoulder"
    ]
}
```

```

    ]

    % "Tree"
    [
    128 128 000; ... % "Tree"
    192 192 000; ... % "VegetationMisc"
    ]

    % "SignSymbol"
    [
    192 128 128; ... % "SignSymbol"
    128 128 064; ... % "Misc_Text"
    000 064 064; ... % "TrafficLight"
    ]

    % "Fence"
    [
    064 064 128; ... % "Fence"
    ]

    % "Car"
    [
    064 000 128; ... % "Car"
    064 128 192; ... % "SUVPickupTruck"
    192 128 192; ... % "Truck_Bus"
    192 064 128; ... % "Train"
    128 064 064; ... % "OtherMoving"
    ]

    % "Pedestrian"
    [
    064 064 000; ... % "Pedestrian"
    192 128 064; ... % "Child"
    064 000 192; ... % "CartLuggagePram"
    064 128 064; ... % "Animal"
    ]

    % "Bicyclist"
    [
    000 128 192; ... % "Bicyclist"
    192 000 192; ... % "MotorcycleScooter"
    ]

    };
end

function pixellLabelColorbar(cmap, classNames)
% Add a colorbar to the current axis. The colorbar is formatted
% to display the class names with the color.

colormap(gca,cmap)

% Add colorbar to current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
c.TickLabels = classNames;

```

```

numClasses = size(cmap,1);

% Center tick labels.
c.Ticks = 1/(numClasses*2):1/numClasses:1;

% Remove tick mark.
c.TickLength = 0;
end

function cmap = camvidColorMap()
% Define the colormap used by CamVid dataset.

cmap = [
    128 128 128   % Sky
    128  0  0    % Building
    192 192 192   % Pole
    128  64 128   % Road
    60  40 222    % Pavement
    128 128  0    % Tree
    192 128 128   % SignSymbol
    64  64 128    % Fence
    64  0 128     % Car
    64  64  0     % Pedestrian
    0 128 192     % Bicyclist
];

% Normalize between [0 1].
cmap = cmap ./ 255;
end

function [imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = partitionCamVidData(imds)
% Partition CamVid data by randomly selecting 60% of the data for training. The
% rest is used for testing.

% Set initial random state for example reproducibility.
rng(0);
numFiles = numel(imds.Files);
shuffledIndices = randperm(numFiles);

% Use 60% of the images for training.
numTrain = round(0.60 * numFiles);
trainingIdx = shuffledIndices(1:numTrain);

% Use 20% of the images for validation
numVal = round(0.20 * numFiles);
valIdx = shuffledIndices(numTrain+1:numTrain+numVal);

% Use the rest for testing.
testIdx = shuffledIndices(numTrain+numVal+1:end);

% Create image datastores for training and test.
trainingImages = imds.Files(trainingIdx);
valImages = imds.Files(valIdx);
testImages = imds.Files(testIdx);

imdsTrain = imageDatastore(trainingImages);
imdsVal = imageDatastore(valImages);
imdsTest = imageDatastore(testImages);

```

```
% Extract class and label IDs info.
classes = pxds.ClassNames;
labelIDs = camvidPixelLabelIDs();

% Create pixel label datastores for training and test.
trainingLabels = pxds.Files(trainingIdx);
valLabels = pxds.Files(valIdx);
testLabels = pxds.Files(testIdx);

pxdsTrain = pixelLabelDatastore(trainingLabels, classes, labelIDs);
pxdsVal = pixelLabelDatastore(valLabels, classes, labelIDs);
pxdsTest = pixelLabelDatastore(testLabels, classes, labelIDs);
end
```

References

- [1] Chen, Liang-Chieh et al. "Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation." ECCV (2018).
- [2] Brostow, G. J., J. Fauqueur, and R. Cipolla. "Semantic object classes in video: A high-definition ground truth database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp 88-97.

See Also

[countEachLabel](#) | [evaluateSemanticSegmentation](#) | [imageDataAugmenter](#) | [labeloverlay](#) | [pixelClassificationLayer](#) | [pixelLabelDatastore](#) | [pixelLabelImageDatastore](#) | [segnetLayers](#) | [semanticseg](#) | [trainNetwork](#) | [trainingOptions](#)

More About

- "Getting Started with Semantic Segmentation Using Deep Learning" on page 14-43
- "Label Pixels for Semantic Segmentation" on page 14-53
- "Deep Learning in MATLAB" (Deep Learning Toolbox)
- "Pretrained Deep Neural Networks" (Deep Learning Toolbox)

Calculate Segmentation Metrics in Block-Based Workflow

This example shows how to calculate the semantic segmentation confusion matrix for individual blocks in a categorical `bigimage` object, then calculate global and block segmentation metrics.

Load a pretrained network that performs binary segmentation of triangles against a background.

```
load('triangleSegmentationNetwork');
```

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.

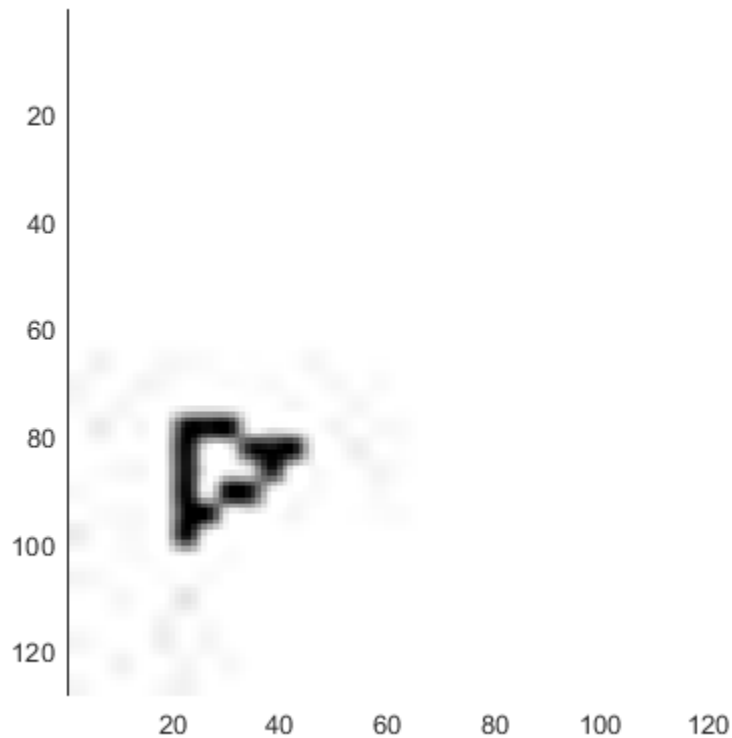
```
testImagesDir = fullfile(dataSetDir,'testImages');
```

Read three test images. Resize each image by a factor of four, convert it to data type `double`, then create a `bigimage` object. A `bigimage` supports block-based image processing workflows.

```
numImages = 3;  
for idx = 1:numImages  
    im = imread(fullfile(testImagesDir,['image_' '00' num2str(idx) '.jpg']));  
    im = imresize(im,4);  
    testImages(idx) = bigimage(im);  
end
```

Display the first test image.

```
bigimageshow(testImages(1))
```



Define the location of the ground truth labels.

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

Define the class names and their associated label IDs.

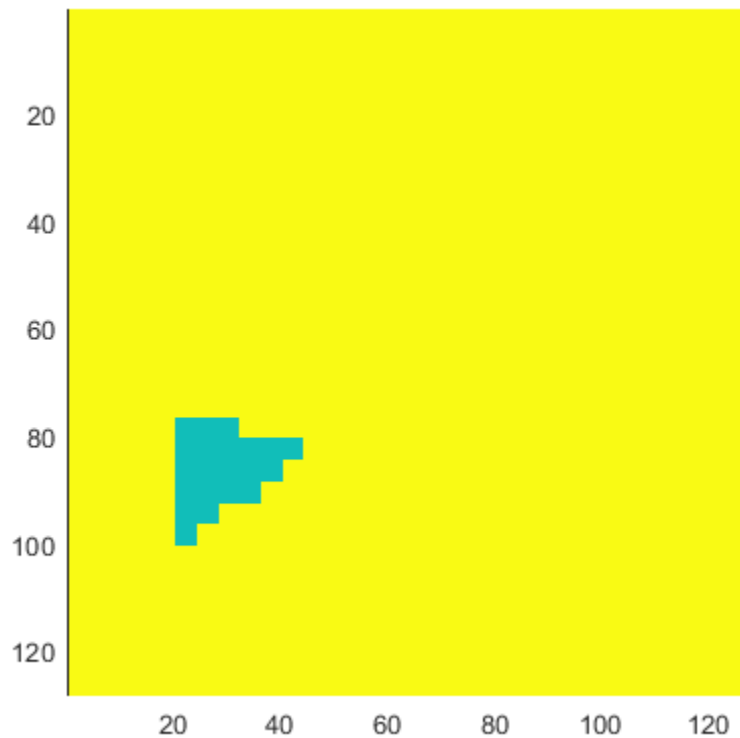
```
classNames = ["triangle", "background"];  
labelIDs   = [255 0];
```

Read in the ground truth labels for each test image. Create a categorical `bigimage` object from the ground truth label.

```
for idx = 1:numImages  
    gtLabel = imread(fullfile(testLabelsDir, ['labeled_image_' '00' num2str(idx) '.png']));  
    gtLabel = imresize(gtLabel, 4, 'nearest');  
    groundTruthImages(idx) = bigimage(gtLabel, ...  
        'Classes', classNames, 'PixelLabelIDs', labelIDs, 'UndefinedID', 1);  
end
```

Display the first ground truth image.

```
bigimageshow(groundTruthImages(1))
```

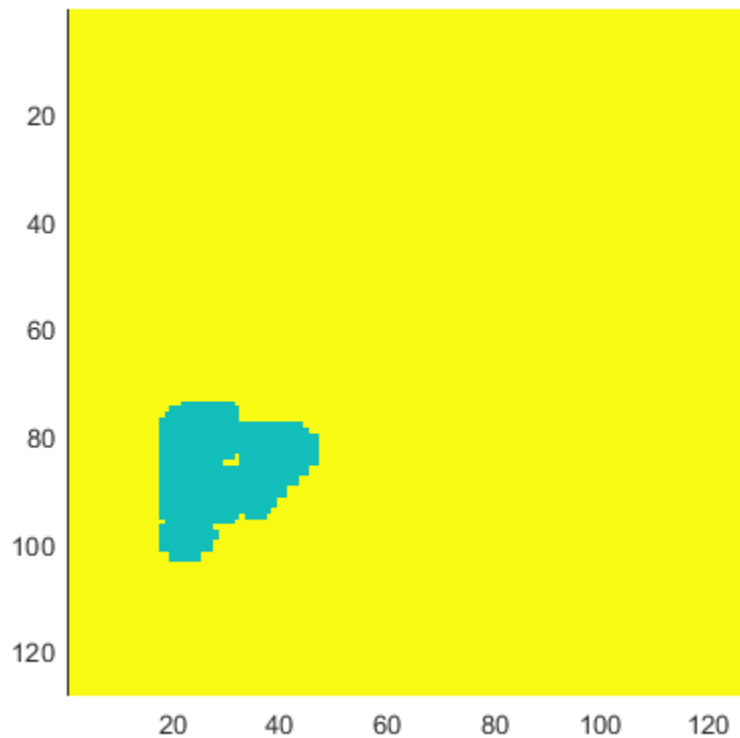
For each test image, use the `apply` function to process each block. The `apply` function performs the operations specified by the helper function `segmentAndCalculateBlockMetrics`, which is defined at the end of this example. The function performs semantic segmentation of each block and calculates the confusion matrix between the predicted and ground truth labels.

```
blockSize = [32 32];
datasetConfMat = table();
for idx = 1:numImages
    [segmentedImages(idx),blockConfMatOneImage] = apply(testImages(idx),1, ...
        @(block,labeledImageBlock,blockInfo) segmentAndCalculateBlockMetrics(block,labeledImageBlock,
        groundTruthImages(idx),'PadPartialBlocks',true, ...
        'BlockSize',blockSize,'UseParallel',false,'IncludeBlockInfo',true);

    % Add an image number corresponding to the block results for each image
    blockConfMatOneImage.ImageNumber = idx.*ones(height(blockConfMatOneImage),1);
    datasetConfMat = [datasetConfMat;blockConfMatOneImage];
end
```

Display the first segmented image.

```
bigimageshow(segmentedImages(1))
```



Evaluate the data set metrics and block metrics for the segmentation.

```
[metrics,blockMetrics] = evaluateSemanticSegmentation(datasetConfMat,classNames,'Metrics','all')
```

```
Evaluating semantic segmentation results
```

```
-----
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU.
* Processed 3 images.
* Finalizing... Done.
* Data set metrics:
```

GlobalAccuracy	MeanAccuracy	MeanIoU	WeightedIoU
0.95428	0.82739	0.69927	0.92533

Calculate the Jaccard score for all images.

```
jaccardSimilarity = metrics.ImageMetrics.MeanIoU
```

```
jaccardSimilarity = 3x1
```

```
0.7664
0.7277
0.6538
```

Supporting Function

The `segmentAndCalculateBlockMetrics` function performs semantic segmentation of a single block then calculates the confusion matrix of the predicted and ground truth labels.

```
function [outputLabeledImageBlock,blockConfMatPerBlock] = segmentAndCalculateBlockMetrics(block, net)

    outputLabeledImageBlock = semanticseg(block,net);

    confusionMatrix = segmentationConfusionMatrix(outputLabeledImageBlock,labeledImageBlock);

    % blockConfMatPerBlock is a struct with confusion matrices and
    % blockInfo. Use the struct with evaluateSemanticSegmentation to
    % calculate metrics and aggregate block-based results.
    blockConfMatPerBlock.ConfusionMatrix = confusionMatrix;
    blockConfMatPerBlock.BlockInfo = blockInfo;
end
```

See Also

[apply](#) | [bigimage](#) | [evaluateSemanticSegmentation](#) | [segmentationConfusionMatrix](#) | [semanticSegmentationMetrics](#)

Related Examples

- “Semantic Segmentation Using Deep Learning” on page 3-43

More About

- “Getting Started with Semantic Segmentation Using Deep Learning” on page 14-43

Semantic Segmentation of Multispectral Images Using Deep Learning

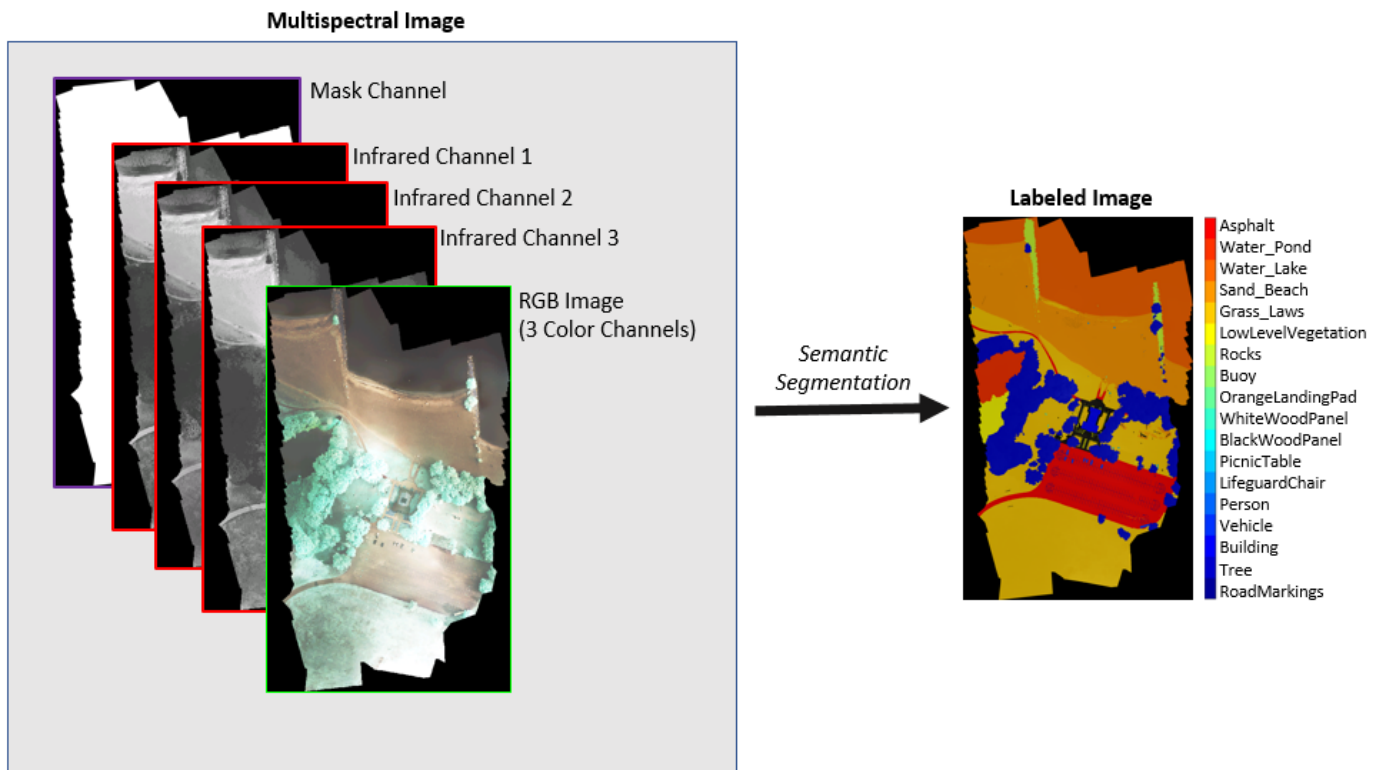
This example shows how to train a U-Net convolutional neural network to perform semantic segmentation of a multispectral image with seven channels: three color channels, three near-infrared channels, and a mask.

The example shows how to train a U-Net network and also provides a pretrained U-Net network. If you choose to train the U-Net network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended (requires Parallel Computing Toolbox™).

Introduction

Semantic segmentation involves labeling each pixel in an image with a class. One application of semantic segmentation is tracking deforestation, which is the change in forest cover over time. Environmental agencies track deforestation to assess and quantify the environmental and ecological health of a region.

Deep-learning-based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One challenge is differentiating classes with similar visual characteristics, such as trying to classify a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.



This example shows how to use deep-learning-based semantic segmentation techniques to calculate the percentage vegetation cover in a region from a set of multispectral images.

Download Data

This example uses a high-resolution multispectral data set to train the network [1 on page 3-0]. The image set was captured using a drone over the Hamlin Beach State Park, NY. The data contains labeled training, validation, and test sets, with 18 object class labels. The size of the data file is ~3.0 GB.

Download the MAT-file version of the data set using the `downloadHamlinBeachMSIData` helper function. This function is attached to the example as a supporting file.

```
imageDir = tempdir;
url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
downloadHamlinBeachMSIData(url, imageDir);
```

In addition, download a pretrained version of U-Net for this dataset using the `downloadTrainedUnet` helper function. This function is attached to the example as a supporting file. The pretrained model enables you to run the entire example without having to wait for training to complete.

```
trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
downloadTrainedUnet(trainedUnet_url, imageDir);
```

Inspect Training Data

Load the data set into the workspace.

```
load(fullfile(imageDir, 'rit18_data', 'rit18_data.mat'));
```

Examine the structure of the data.

```
whos train_data val_data test_data
```

Name	Size	Bytes	Class	Attributes
test_data	7x12446x7654	1333663576	uint16	
train_data	7x9393x5642	741934284	uint16	
val_data	7x8833x6918	855493716	uint16	

The multispectral image data is arranged as *numChannels-by-width-by-height* arrays. However, in MATLAB®, multichannel images are arranged as *width-by-height-by-numChannels* arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`. This function is attached to the example as a supporting file.

```
train_data = switchChannelsToThirdPlane(train_data);
val_data = switchChannelsToThirdPlane(val_data);
test_data = switchChannelsToThirdPlane(test_data);
```

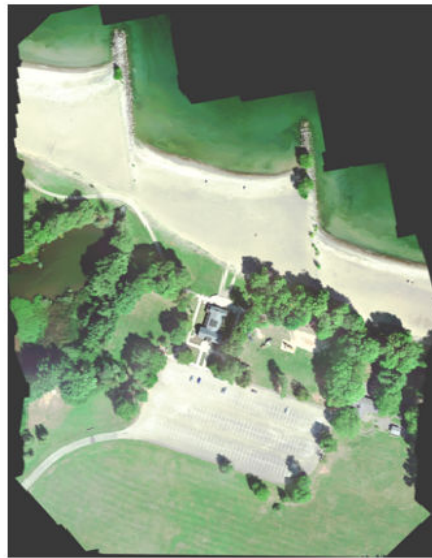
Confirm that the data has the correct structure.

```
whos train_data val_data test_data
```

Name	Size	Bytes	Class	Attributes
test_data	12446x7654x7	1333663576	uint16	
train_data	9393x5642x7	741934284	uint16	
val_data	8833x6918x7	855493716	uint16	

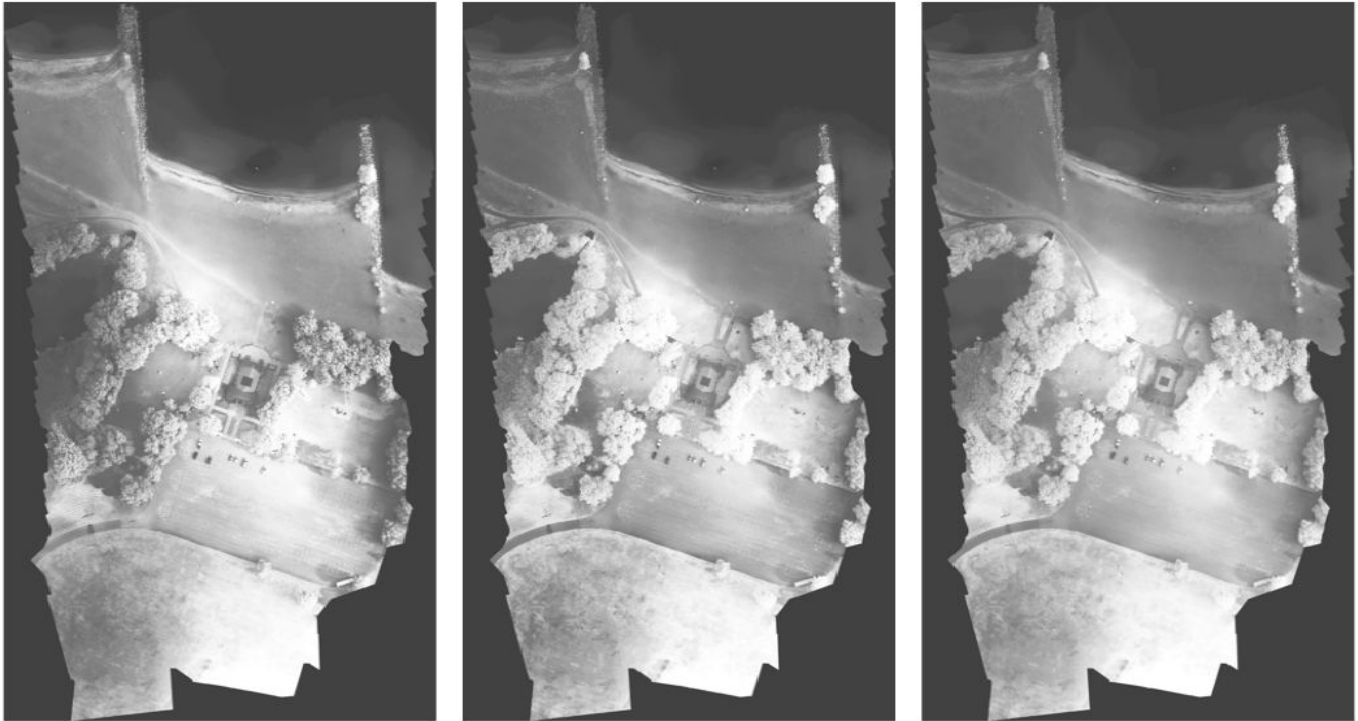
The RGB color channels are the 3rd, 2nd and 1st image channels. Display the color component of the training, validation, and test images as a montage. To make the images appear brighter on the screen, equalize their histograms by using the `histeq` function.

```
figure
montage(...
    {histeq(train_data(:,:, [3 2 1])), ...
    histeq(val_data(:,:, [3 2 1])), ...
    histeq(test_data(:,:, [3 2 1]))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('RGB Component of Training Image (Left), Validation Image (Center), and Test Image (Right)')
```



Display the last three histogram-equalized channels of the training data as a montage. These channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. For example, the trees near the center of the second channel image show more detail than the trees in the other two channels.

```
figure
montage(...
    {histeq(train_data(:,:,4)), ...
    histeq(train_data(:,:,5)), ...
    histeq(train_data(:,:,6))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('IR Channels 1 (Left), 2, (Center), and 3 (Right) of Training Image')
```



Channel 7 is a mask that indicates the valid segmentation region. Display the mask for the training, validation, and test images.

```
figure
montage(...
    {train_data(:,:,7), ...
    val_data(:,:,7), ...
    test_data(:,:,7)}, ...
    'BorderSize',10,'BackgroundColor','white')
title('Mask of Training Image (Left), Validation Image (Center), and Test Image (Right)')
```



The labeled images contain the ground truth data for the segmentation, with each pixel assigned to one of the 18 classes. Get a list of the classes with their corresponding IDs.

```
disp(classes)
```

```
0. Other Class/Image Border
1. Road Markings
2. Tree
3. Building
4. Vehicle (Car, Truck, or Bus)
5. Person
6. Lifeguard Chair
7. Picnic Table
8. Black Wood Panel
9. White Wood Panel
10. Orange Landing Pad
11. Water Buoy
12. Rocks
13. Other Vegetation
14. Grass
15. Sand
16. Water (Lake)
17. Water (Pond)
18. Asphalt (Parking Lot/Walkway)
```

Create a vector of class names.

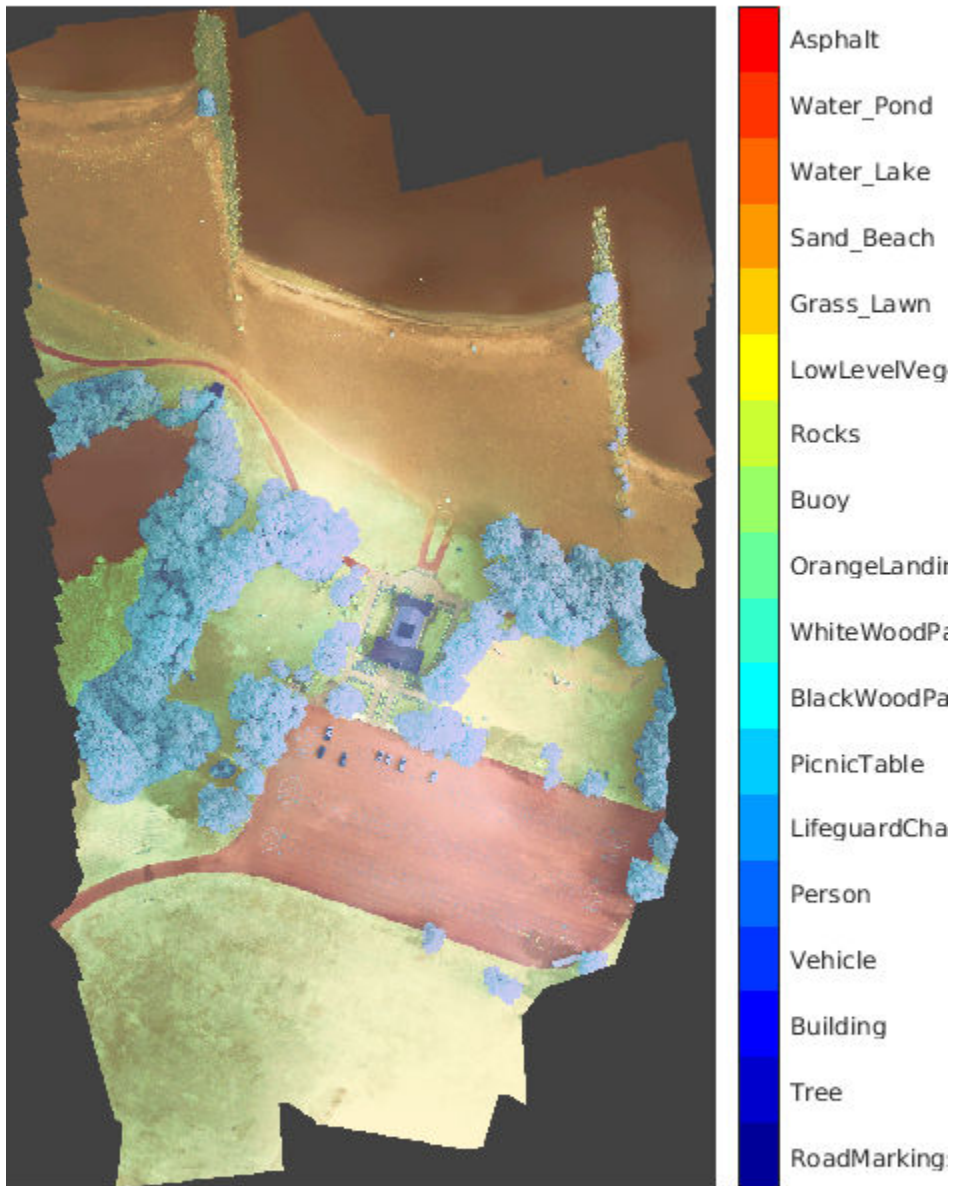
```
classNames = [ "RoadMarkings", "Tree", "Building", "Vehicle", "Person", ...
               "LifeguardChair", "PicnicTable", "BlackWoodPanel", ...
               "WhiteWoodPanel", "OrangeLandingPad", "Buoy", "Rocks", ...
```



```
"LowLevelVegetation", "Grass_Lawn", "Sand_Beach", ...  
"Water_Lake", "Water_Pond", "Asphalt"];
```

Overlay the labels on the histogram-equalized RGB training image. Add a colorbar to the image.

```
cmap = jet(numel(classNames));  
B = labeloverlay(histeq(train_data(:,:,4:6)),train_labels,'Transparency',0.8,'Colormap',cmap);  
  
figure  
title('Training Labels')  
imshow(B)  
N = numel(classNames);  
ticks = 1/(N*2):1/N:1;  
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','none')  
colormap(cmap)
```



Save the training data as a MAT file and the training labels as a PNG file.

```
save('train_data.mat','train_data');
imwrite(train_labels,'train_labels.png');
```

Create Random Patch Extraction Datastore for Training

Use a random patch extraction datastore to feed the training data to the network. This datastore extracts multiple corresponding random patches from an image datastore and pixel label datastore that contain ground truth images and pixel label data. Patching is a common technique to prevent running out of memory for large images and to effectively increase the amount of available training data.

Begin by storing the training images from 'train_data.mat' in an `imageDatastore`. Because the MAT file format is a nonstandard image format, you must use a MAT file reader to enable reading the image data. You can use the helper MAT file reader, `matReader`, that extracts the first six channels from the training data and omits the last channel containing the mask. This function is attached to the example as a supporting file.

```
imds = imageDatastore('train_data.mat','FileExtensions','.mat','ReadFcn',@matReader);
```

Create a `pixelLabelDatastore` to store the label patches containing the 18 labeled regions.

```
pixelLabelIds = 1:18;
pxds = pixelLabelDatastore('train_labels.png',classNames,pixelLabelIds);
```

Create a `randomPatchExtractionDatastore` from the image datastore and the pixel label datastore. Each mini-batch contains 16 patches of size 256-by-256 pixels. One thousand mini-batches are extracted at each iteration of the epoch.

```
dsTrain = randomPatchExtractionDatastore(imds,pxds,[256,256],'PatchesPerImage',16000);
```

The random patch extraction datastore `dsTrain` provides mini-batches of data to the network at each iteration of the epoch. Preview the datastore to explore the data.

```
inputBatch = preview(dsTrain);
disp(inputBatch)
```

InputImage	ResponsePixelLabelImage
{256×256×6 uint16}	{256×256 categorical}
{256×256×6 uint16}	{256×256 categorical}
{256×256×6 uint16}	{256×256 categorical}
{256×256×6 uint16}	{256×256 categorical}
{256×256×6 uint16}	{256×256 categorical}
{256×256×6 uint16}	{256×256 categorical}
{256×256×6 uint16}	{256×256 categorical}
{256×256×6 uint16}	{256×256 categorical}

Create U-Net Network Layers

This example uses a variation of the U-Net network. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image [2 on page 3-0]. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

This example modifies the U-Net to use zero-padding in the convolutions, so that the input and the output to the convolutions have the same size. Use the helper function, `createUnet`, to create a U-

Net with a few preselected hyperparameters. This function is attached to the example as a supporting file.

```
inputTileSize = [256,256,6];
lgraph = createUnet(inputTileSize);
disp(lgraph.Layers)
```

58x1 Layer array with layers:

1	'ImageInputLayer'	Image Input	256x256x6 images v
2	'Encoder-Section-1-Conv-1'	Convolution	64 3x3x6 convolut
3	'Encoder-Section-1-ReLU-1'	ReLU	ReLU
4	'Encoder-Section-1-Conv-2'	Convolution	64 3x3x64 convolu
5	'Encoder-Section-1-ReLU-2'	ReLU	ReLU
6	'Encoder-Section-1-MaxPool'	Max Pooling	2x2 max pooling w
7	'Encoder-Section-2-Conv-1'	Convolution	128 3x3x64 convolu
8	'Encoder-Section-2-ReLU-1'	ReLU	ReLU
9	'Encoder-Section-2-Conv-2'	Convolution	128 3x3x128 convo
10	'Encoder-Section-2-ReLU-2'	ReLU	ReLU
11	'Encoder-Section-2-MaxPool'	Max Pooling	2x2 max pooling w
12	'Encoder-Section-3-Conv-1'	Convolution	256 3x3x128 convo
13	'Encoder-Section-3-ReLU-1'	ReLU	ReLU
14	'Encoder-Section-3-Conv-2'	Convolution	256 3x3x256 convo
15	'Encoder-Section-3-ReLU-2'	ReLU	ReLU
16	'Encoder-Section-3-MaxPool'	Max Pooling	2x2 max pooling w
17	'Encoder-Section-4-Conv-1'	Convolution	512 3x3x256 convo
18	'Encoder-Section-4-ReLU-1'	ReLU	ReLU
19	'Encoder-Section-4-Conv-2'	Convolution	512 3x3x512 convo
20	'Encoder-Section-4-ReLU-2'	ReLU	ReLU
21	'Encoder-Section-4-DropOut'	Dropout	50% dropout
22	'Encoder-Section-4-MaxPool'	Max Pooling	2x2 max pooling w
23	'Mid-Conv-1'	Convolution	1024 3x3x512 convo
24	'Mid-ReLU-1'	ReLU	ReLU
25	'Mid-Conv-2'	Convolution	1024 3x3x1024 conv
26	'Mid-ReLU-2'	ReLU	ReLU
27	'Mid-DropOut'	Dropout	50% dropout
28	'Decoder-Section-1-UpConv'	Transposed Convolution	512 2x2x1024 transp
29	'Decoder-Section-1-UpReLU'	ReLU	ReLU
30	'Decoder-Section-1-DepthConcatenation'	Depth concatenation	Depth concatenati
31	'Decoder-Section-1-Conv-1'	Convolution	512 3x3x1024 convo
32	'Decoder-Section-1-ReLU-1'	ReLU	ReLU
33	'Decoder-Section-1-Conv-2'	Convolution	512 3x3x512 convo
34	'Decoder-Section-1-ReLU-2'	ReLU	ReLU
35	'Decoder-Section-2-UpConv'	Transposed Convolution	256 2x2x512 transp
36	'Decoder-Section-2-UpReLU'	ReLU	ReLU
37	'Decoder-Section-2-DepthConcatenation'	Depth concatenation	Depth concatenati
38	'Decoder-Section-2-Conv-1'	Convolution	256 3x3x512 convo
39	'Decoder-Section-2-ReLU-1'	ReLU	ReLU
40	'Decoder-Section-2-Conv-2'	Convolution	256 3x3x256 convo
41	'Decoder-Section-2-ReLU-2'	ReLU	ReLU
42	'Decoder-Section-3-UpConv'	Transposed Convolution	128 2x2x256 transp
43	'Decoder-Section-3-UpReLU'	ReLU	ReLU
44	'Decoder-Section-3-DepthConcatenation'	Depth concatenation	Depth concatenati
45	'Decoder-Section-3-Conv-1'	Convolution	128 3x3x256 convo
46	'Decoder-Section-3-ReLU-1'	ReLU	ReLU
47	'Decoder-Section-3-Conv-2'	Convolution	128 3x3x128 convo
48	'Decoder-Section-3-ReLU-2'	ReLU	ReLU
49	'Decoder-Section-4-UpConv'	Transposed Convolution	64 2x2x128 transp

50	'Decoder-Section-4-UpReLU'	ReLU	ReLU
51	'Decoder-Section-4-DepthConcatenation'	Depth concatenation	Depth concatenation
52	'Decoder-Section-4-Conv-1'	Convolution	64 3x3x128 convolu
53	'Decoder-Section-4-ReLU-1'	ReLU	ReLU
54	'Decoder-Section-4-Conv-2'	Convolution	64 3x3x64 convolu
55	'Decoder-Section-4-ReLU-2'	ReLU	ReLU
56	'Final-ConvolutionLayer'	Convolution	18 1x1x64 convolu
57	'Softmax-Layer'	Softmax	softmax
58	'Segmentation-Layer'	Pixel Classification Layer	Cross-entropy loss

Select Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SGDM by using the `trainingOptions` (Deep Learning Toolbox) function.

Training a deep network is time-consuming. Accelerate the training by specifying a high learning rate. However, this can cause the gradients of the network to explode or grow uncontrollably, preventing the network from training successfully. To keep the gradients in a meaningful range, enable gradient clipping by specifying `'GradientThreshold'` as `0.05`, and specify `'GradientThresholdMethod'` to use the L2-norm of the gradients.

```
initialLearningRate = 0.05;
maxEpochs = 150;
minibatchSize = 16;
l2reg = 0.0001;

options = trainingOptions('sgdm',...
    'InitialLearnRate',initialLearningRate, ...
    'Momentum',0.9,...
    'L2Regularization',l2reg,...
    'MaxEpochs',maxEpochs,...
    'MiniBatchSize',minibatchSize,...
    'LearnRateSchedule','piecewise',...
    'Shuffle','every-epoch',...
    'GradientThresholdMethod','l2norm',...
    'GradientThreshold',0.05, ...
    'Plots','training-progress', ...
    'VerboseFrequency',20);
```

Train the Network

After configuring the training options and the random patch extraction datastore, train the U-Net network by using the `trainNetwork` (Deep Learning Toolbox) function. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained U-Net network.

Note: Training takes about 20 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    [net,info] = trainNetwork(dsTrain,lgraph,options);
```

```
    save(['multispectralUnet-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat'],'net','options');  
else  
    load(fullfile(imageDir,'trainedUnet','multispectralUnet.mat'));  
end
```

You can now use the U-Net to semantically segment the multispectral image.

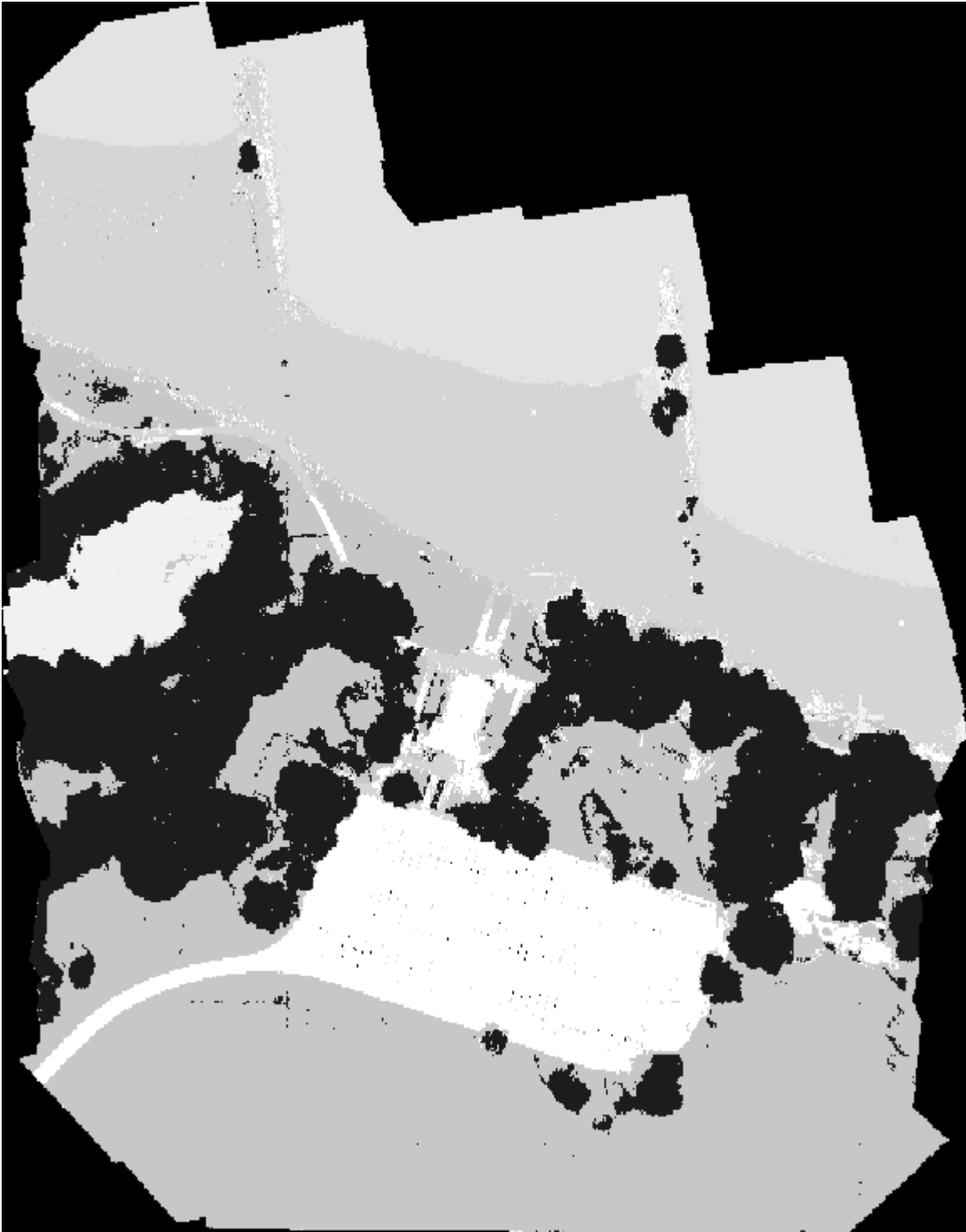
Predict Results on Test Data

To perform the forward pass on the trained network, use the helper function, `segmentImage`, with the validation data set. This function is attached to the example as a supporting file. `segmentImage` performs segmentation on image patches using the `semanticseg` function.

```
predictPatchSize = [1024 1024];  
segmentedImage = segmentImage(val_data,net,predictPatchSize);
```

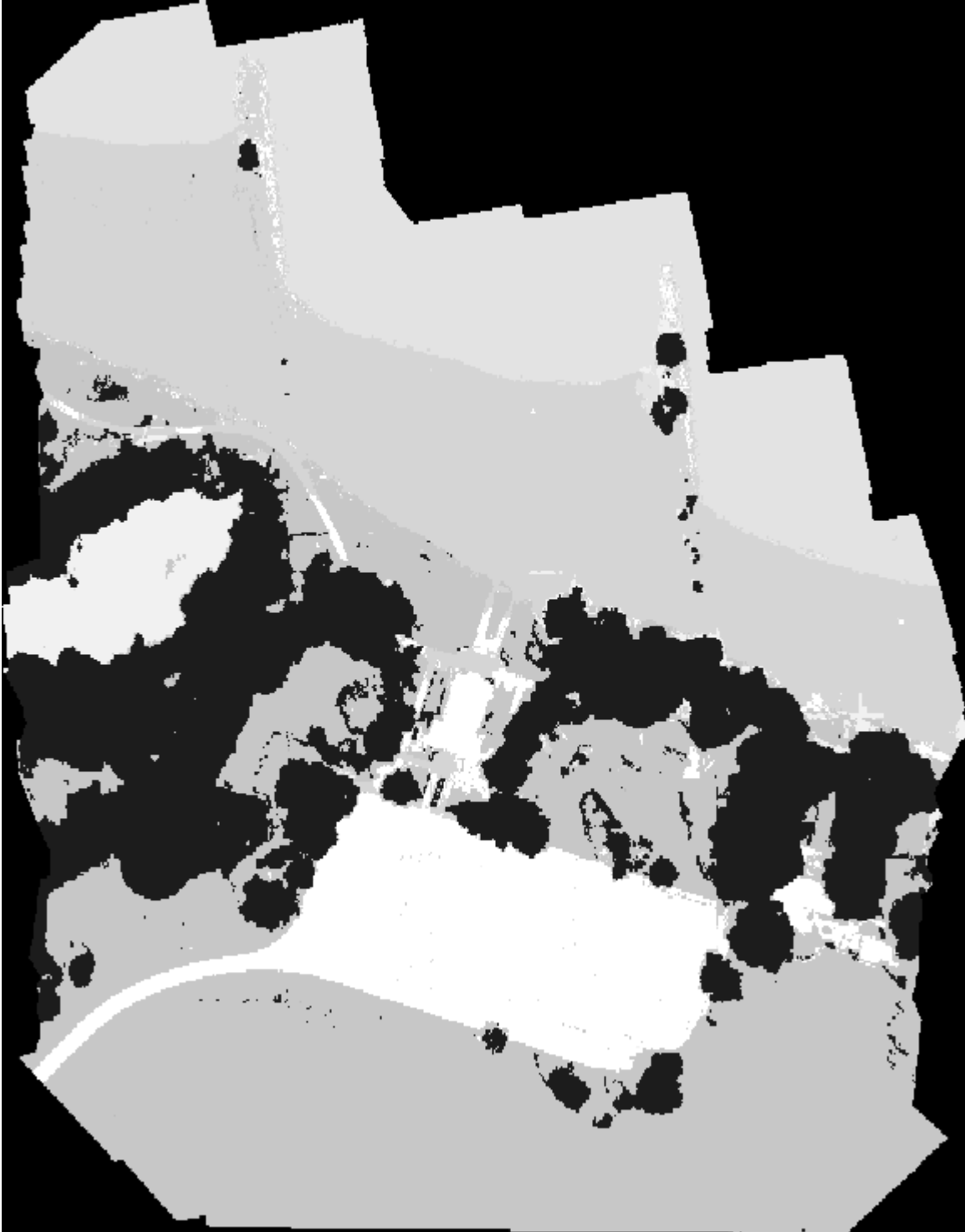
To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the validation data.

```
segmentedImage = uint8(val_data(:,:,7)~=0) .* segmentedImage;  
  
figure  
imshow(segmentedImage,[])  
title('Segmented Image')
```



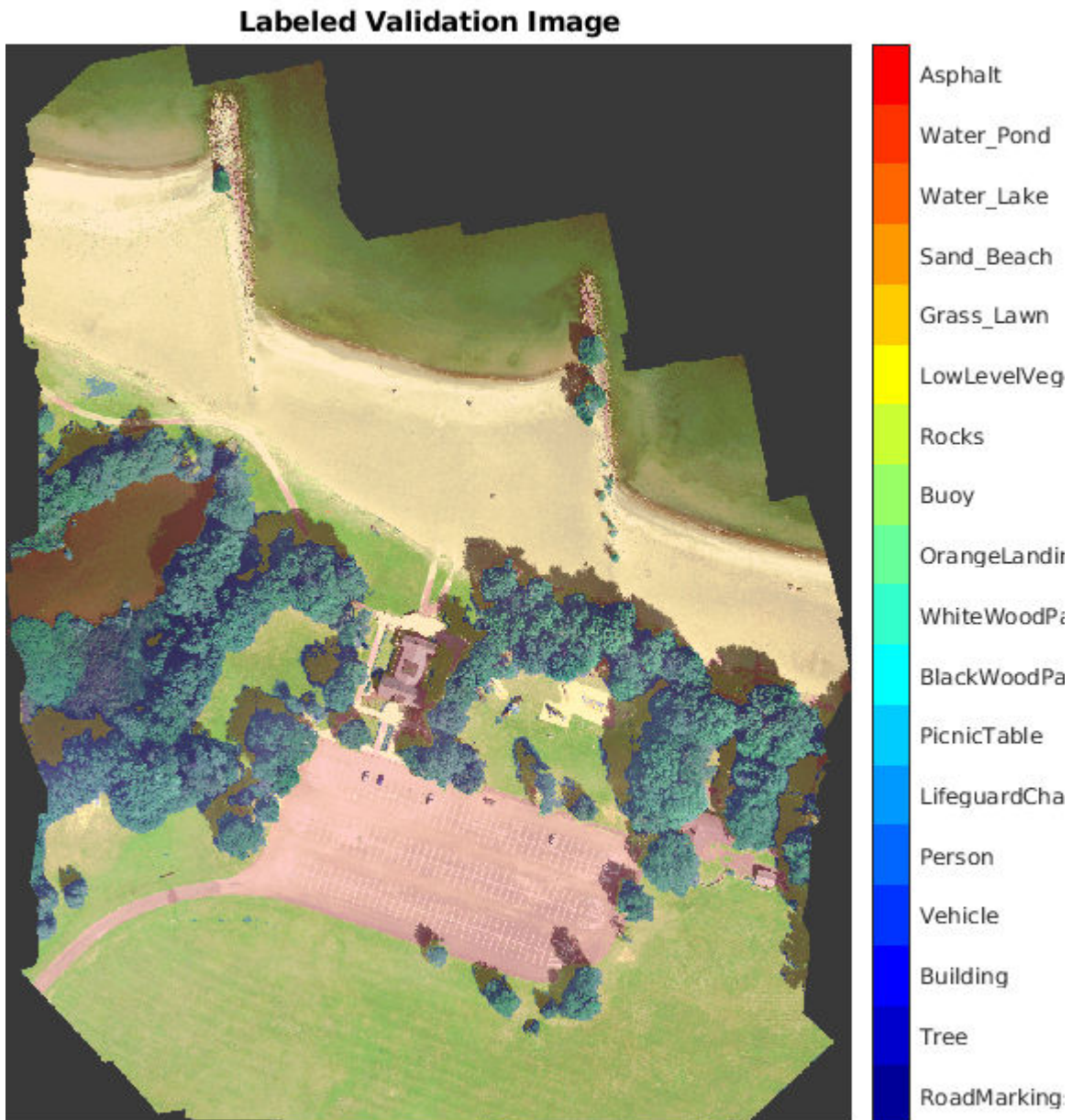
The output of semantic segmentation is noisy. Perform post image processing to remove noise and stray pixels. Use the `medfilt2` function to remove salt-and-pepper noise from the segmentation. Visualize the segmented image with the noise removed.

```
segmentedImage = medfilt2(segmentedImage,[7,7]);  
imshow(segmentedImage,[]);  
title('Segmented Image with Noise Removed')
```



Overlay the segmented image on the histogram-equalized RGB validation image.


```
B = labeloverlay(histeq(val_data(:,:, [3 2 1])), segmentedImage, 'Transparency', 0.8, 'Colormap', cmap);  
  
figure  
imshow(B)  
title('Labeled Validation Image')  
colorbar('TickLabels', cellstr(classNames), 'Ticks', ticks, 'TickLength', 0, 'TickLabelInterpreter', 'none')  
colormap(cmap)
```



Save the segmented image and ground truth labels as PNG files. These will be used to compute accuracy metrics.

```
imwrite(segmentedImage, 'results.png');  
imwrite(val_labels, 'gtruth.png');
```

Quantify Segmentation Accuracy

Create a `pixelLabelDatastore` for the segmentation results and the ground truth labels.

```
pxdsResults = pixelLabelDatastore('results.png',classNames,pixelLabelIds);
pxdsTruth = pixelLabelDatastore('gtruth.png',classNames,pixelLabelIds);
```

Measure the global accuracy of the semantic segmentation by using the `evaluateSemanticSegmentation` function.

```
ssm = evaluateSemanticSegmentation(pxdsResults,pxdsTruth,'Metrics','global-accuracy');
```

```
Evaluating semantic segmentation results
```

```
-----
```

```
* Selected metrics: global accuracy.
* Processed 1 images.
* Finalizing... Done.
* Data set metrics:
```

```
GlobalAccuracy
```

```
-----
0.90698
```

The global accuracy score indicates that just over 90% of the pixels are classified correctly.

Calculate Extent of Vegetation Cover

The final goal of this example is to calculate the extent of vegetation cover in the multispectral image.

Find the number of pixels labeled vegetation. The label IDs 2 ("Trees"), 13 ("LowLevelVegetation"), and 14 ("Grass_Lawn") are the vegetation classes. Also find the total number of valid pixels by summing the pixels in the ROI of the mask image.

```
vegetationClassIds = uint8([2,13,14]);
vegetationPixels = ismember(segmentedImage(:),vegetationClassIds);
validPixels = (segmentedImage~=0);
```

```
numVegetationPixels = sum(vegetationPixels(:));
numValidPixels = sum(validPixels(:));
```

Calculate the percentage of vegetation cover by dividing the number of vegetation pixels by the number of valid pixels.

```
percentVegetationCover = (numVegetationPixels/numValidPixels)*100;
fprintf('The percentage of vegetation cover is %3.2f%%.',percentVegetationCover);
```

```
The percentage of vegetation cover is 51.72%.
```

References

[1] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918. 2017.

[2] Ronneberger, O., P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." CoRR, abs/1505.04597. 2015.

See Also

`evaluateSemanticSegmentation` | `histeq` | `imageDatastore` | `pixelLabelDatastore` | `randomPatchExtractionDatastore` | `semanticseg` | `trainNetwork` | `trainingOptions` | `UNETLayers`

More About

- "Getting Started with Semantic Segmentation Using Deep Learning" on page 14-43
- "Semantic Segmentation Using Deep Learning" on page 3-43
- "Datastores for Deep Learning" (Deep Learning Toolbox)

External Websites

- <https://github.com/rmkemker/RIT-18>

3-D Brain Tumor Segmentation Using Deep Learning

This example shows how to train a 3-D U-Net neural network and perform semantic segmentation of brain tumors from 3-D medical images. The example shows how to train a 3-D U-Net network and also provides a pretrained network. Use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for 3-D semantic segmentation (requires Parallel Computing Toolbox™).

Introduction

Semantic segmentation involves labeling each pixel in an image or voxel of a 3-D volume with a class. This example illustrates the use of deep learning methods to perform binary semantic segmentation of brain tumors in magnetic resonance imaging (MRI) scans. In this binary segmentation, each pixel is labeled as tumor or background.

This example performs brain tumor segmentation using a 3-D U-Net architecture [1 on page 3-0]. U-Net is a fast, efficient and simple network that has become popular in the semantic segmentation domain.

One challenge of medical image segmentation is the amount of memory needed to store and process 3-D volumes. Training a network on the full input volume is impractical due to GPU resource constraints. This example solves the problem by training the network on image patches. The example uses an overlap-tile strategy to stitch test patches into a complete segmented test volume. The example avoids border artifacts by using the valid part of the convolution in the neural network [5 on page 3-0].

A second challenge of medical image segmentation is class imbalance in the data that hampers training when using conventional cross entropy loss. This example solves the problem by using a weighted multiclass Dice loss function [4 on page 3-0]. Weighting the classes helps to counter the influence of larger regions on the Dice score, making it easier for the network to learn how to segment smaller regions.

Download Training, Validation, and Test Data

This example uses the BraTS data set [2 on page 3-0]. The BraTS data set contains MRI scans of brain tumors, namely gliomas, which are the most common primary brain malignancies. The size of the data file is ~7 GB. If you do not want to download the BraTS data set, then go directly to the Download Pretrained Network and Sample Test Set on page 3-0 section in this example.

Create a directory to store the BraTS data set.

```
imageDir = fullfile(tempdir, 'BraTS');
if ~exist(imageDir, 'dir')
    mkdir(imageDir);
end
```

To download the BraTS data, go to the Medical Segmentation Decathlon website and click the "Download Data" link. Download the "Task01_BrainTumour.tar" file [3 on page 3-0]. Unzip the TAR file into the directory specified by the `imageDir` variable. When unzipped successfully, `imageDir` will contain a directory named `Task01_BrainTumour` that has three subdirectories: `imagesTr`, `imagesTs`, and `labelsTr`.

The data set contains 750 4-D volumes, each representing a stack of 3-D images. Each 4-D volume has size 240-by-240-by-155-by-4, where the first three dimensions correspond to height, width, and

depth of a 3-D volumetric image. The fourth dimension corresponds to different scan modalities. The data set is divided into 484 training volumes with voxel labels and 266 test volumes. The test volumes do not have labels so this example does not use the test data. Instead, the example splits the 484 training volumes into three independent sets that are used for training, validation, and testing.

Preprocess Training and Validation Data

To train the 3-D U-Net network more efficiently, preprocess the MRI data using the helper function `preprocessBraTSdataset`. This function is attached to the example as a supporting file.

The helper function performs these operations:

- Crop the data to a region containing primarily the brain and tumor. Cropping the data reduces the size of data while retaining the most critical part of each MRI volume and its corresponding labels.
- Normalize each modality of each volume independently by subtracting the mean and dividing by the standard deviation of the cropped brain region.
- Split the 484 training volumes into 400 training, 29 validation, and 55 test sets.

Preprocessing the data can take about 30 minutes to complete.

```
sourceDataLoc = [imageDir filesep 'Task01_BrainTumour'];
preprocessDataLoc = fullfile(tempdir, 'BraTS', 'preprocessedDataset');
preprocessBraTSdataset(preprocessDataLoc, sourceDataLoc);
```

Create Random Patch Extraction Datastore for Training and Validation

Use a random patch extraction datastore to feed the training data to the network and to validate the training progress. This datastore extracts random patches from ground truth images and corresponding pixel label data. Patching is a common technique to prevent running out of memory when training with arbitrarily large volumes.

Create an `imageDatastore` to store the 3-D image data. Because the MAT-file format is a nonstandard image format, you must use a MAT-file reader to enable reading the image data. You can use the helper MAT-file reader, `matRead`. This function is attached to the example as a supporting file.

```
volReader = @(x) matRead(x);
volLoc = fullfile(preprocessDataLoc, 'imagesTr');
volds = imageDatastore(volLoc, ...
    'FileExtensions', '.mat', 'ReadFcn', volReader);
```

Create a `pixelLabelDatastore` to store the labels.

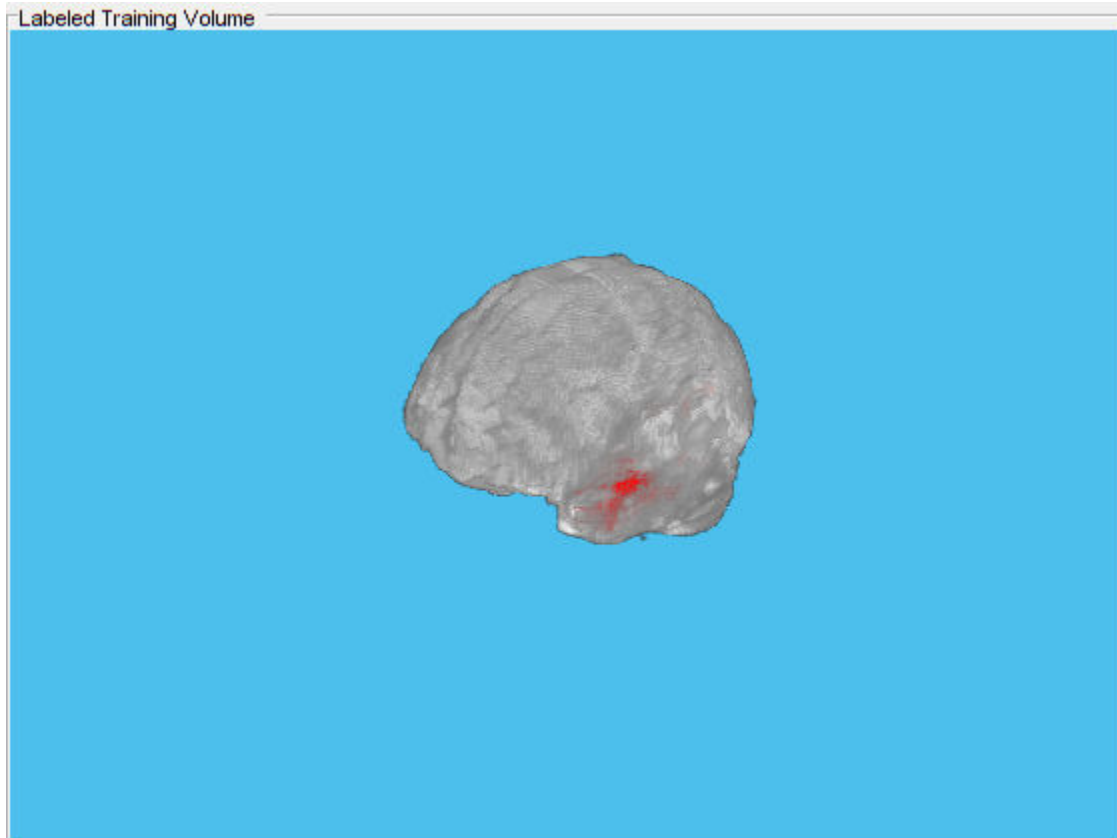
```
lblLoc = fullfile(preprocessDataLoc, 'labelsTr');
classNames = ["background", "tumor"];
pixelLabelID = [0 1];
pxds = pixelLabelDatastore(lblLoc, classNames, pixelLabelID, ...
    'FileExtensions', '.mat', 'ReadFcn', volReader);
```

Preview one image volume and label. Display the labeled volume using the `labelvolshow` function. Make the background fully transparent by setting the visibility of the background label (1) to 0.

```
volume = preview(volds);
label = preview(pxds);

viewPnl = uipanel(figure, 'Title', 'Labeled Training Volume');
```

```
hPred = labelvolshow(label,volume(:,:,,1),'Parent',viewPnl, ...
    'LabelColor',[0 0 0;1 0 0]);
hPred.LabelVisibility(1) = 0;
```



Create a `randomPatchExtractionDatastore` that contains the training image and pixel label data. Specify a patch size of 132-by-132-by-132 voxels. Specify `'PatchesPerImage'` to extract 16 randomly positioned patches from each pair of volumes and labels during training. Specify a mini-batch size of 8.

```
patchSize = [132 132 132];
patchPerImage = 16;
miniBatchSize = 8;
patchds = randomPatchExtractionDatastore(volds,pxds,patchSize, ...
    'PatchesPerImage',patchPerImage);
patchds.MiniBatchSize = miniBatchSize;
```

Follow the same steps to create a `randomPatchExtractionDatastore` that contains the validation image and pixel label data. You can use validation data to evaluate whether the network is continuously learning, underfitting, or overfitting as time progresses.

```
volLocVal = fullfile(preprocessDataLoc,'imagesVal');
voldsVal = imageDatastore(volLocVal, ...
    'FileExtensions','.mat','ReadFcn',volReader);

lblLocVal = fullfile(preprocessDataLoc,'labelsVal');
pxdsVal = pixelLabelDatastore(lblLocVal,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

```
dsVal = randomPatchExtractionDatastore(voldsVal,pxdsVal,patchSize, ...  
    'PatchesPerImage',patchPerImage);  
dsVal.MinibatchSize = miniBatchSize;
```

Augment the training and validation data by using the `transform` function with custom preprocessing operations specified by the helper function `augmentAndCrop3dPatch`. This function is attached to the example as a supporting file.

The `augmentAndCrop3dPatch` function performs these operations:

- 1 Randomly rotate and reflect training data to make the training more robust. The function does not rotate or reflect validation data.
- 2 Crop response patches to the output size of the network, 44-by-44-by-44 voxels.

```
dataSource = 'Training';  
dsTrain = transform(patchds,@(patchIn)augmentAndCrop3dPatch(patchIn,dataSource));
```

```
dataSource = 'Validation';  
dsVal = transform(dsVal,@(patchIn)augmentAndCrop3dPatch(patchIn,dataSource));
```

Set Up 3-D U-Net Layers

This example uses the 3-D U-Net network [1 on page 3-0]. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. A batch normalization layer is introduced before each ReLU layer. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

Create a default 3-D U-Net network by using the `unetLayers` function. Specify two class segmentation. Also specify valid convolution padding to avoid border artifacts when using the overlap-tile strategy for prediction of the test volumes.

```
inputPatchSize = [132 132 132 4];  
numClasses = 2;  
[lgraph,outPatchSize] = unet3dLayers(inputPatchSize,numClasses,'ConvolutionPadding','valid');
```

To better segment smaller tumor regions and reduce the influence of larger background regions, this example uses a `dicePixelClassificationLayer`. Replace the pixel classification layer with the Dice pixel classification layer.

```
outputLayer = dicePixelClassificationLayer('Name','Output');  
lgraph = replaceLayer(lgraph,'Segmentation-Layer',outputLayer);
```

The data has already been normalized in the Preprocess Training and Validation Data on page 3-0 section of this example. Data normalization in the `image3dInputLayer` (Deep Learning Toolbox) is unnecessary, so replace the input layer with an input layer that does not have data normalization.

```
inputLayer = image3dInputLayer(inputPatchSize,'Normalization','none','Name','ImageInputLayer');  
lgraph = replaceLayer(lgraph,'ImageInputLayer',inputLayer);
```

Alternatively, you can modify the 3-D U-Net network by using Deep Network Designer App from Deep Learning Toolbox™.

Plot the graph of the updated 3-D U-Net network.


```
analyzeNetwork(lgraph)
```

Specify Training Options

Train the network using the adam optimization solver. Specify the hyperparameter settings using the `trainingOptions` (Deep Learning Toolbox) function. The initial learning rate is set to $5e-4$ and gradually decreases over the span of training. You can experiment with the `MiniBatchSize` property based on your GPU memory. To maximize GPU memory utilization, favor large input patches over a large batch size. Note that batch normalization layers are less effective for smaller values of `MiniBatchSize`. Tune the initial learning rate based on the `MiniBatchSize`.

```
options = trainingOptions('adam', ...
    'MaxEpochs',50, ...
    'InitialLearnRate',5e-4, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',5, ...
    'LearnRateDropFactor',0.95, ...
    'ValidationData',dsVal, ...
    'ValidationFrequency',400, ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'MiniBatchSize',miniBatchSize);
```

Download Pretrained Network and Sample Test Set

Optionally, download a pretrained version of 3-D U-Net and five sample test volumes and their corresponding labels from the BraTS data set [3 on page 3-0]. The pretrained model and sample data enable you to perform segmentation on test data without downloading the full data set or waiting for the network to train.

```
trained3DUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/brainTumor3DUnetValid.ma
sampleData_url = 'https://www.mathworks.com/supportfiles/vision/data/sampleBraTSTestSetValid.tar
```

```
imageDir = fullfile(tempdir,'BraTS');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end
```

```
downloadTrained3DUnetSampleData(trained3DUnet_url,sampleData_url,imageDir);
```

```
Downloading pretrained 3-D U-Net for BraTS data set.
This will take several minutes to download...
Done.
```

```
Downloading sample BraTS test dataset.
This will take several minutes to download and unzip...
Done.
```

Train Network

After configuring the training options and the data source, train the 3-D U-Net network by using the `trainNetwork` (Deep Learning Toolbox) function. To train the network, set the `doTraining` variable in the following code to `true`. A CUDA capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` variable in the following code as `false`, then the example returns a pretrained 3-D U-Net network.

Note: Training takes about 30 hours on a multi-GPU system with 4 NVIDIA™ Titan Xp GPUs and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now, 'dd-mmm-yyyy-HH-MM-SS');
    [net,info] = trainNetwork(dsTrain,lgraph,options);
    save(['trained3DUNetValid-' modelDateTime '-Epoch-' num2str(options.MaxEpochs) '.mat'], 'net')
else
    inputPatchSize = [132 132 132 4];
    outPatchSize = [44 44 44 2];
    load(fullfile(imageDir, 'trained3DUNet', 'brainTumor3DUNetValid.mat'));
end
```

You can now use the U-Net to semantically segment brain tumors.

Perform Segmentation of Test Data

A GPU is highly recommended for performing semantic segmentation of the image volumes (requires Parallel Computing Toolbox™).

Select the source of test data that contains ground truth volumes and labels for testing. If you keep the `useFullTestSet` variable in the following code as `false`, then the example uses five volumes for testing. If you set the `useFullTestSet` variable to `true`, then the example uses 55 test images selected from the full data set.

```
useFullTestSet = false;
if useFullTestSet
    volLocTest = fullfile(preprocessDataLoc, 'imagesTest');
    lblLocTest = fullfile(preprocessDataLoc, 'labelsTest');
else
    volLocTest = fullfile(imageDir, 'sampleBraTSTestSetValid', 'imagesTest');
    lblLocTest = fullfile(imageDir, 'sampleBraTSTestSetValid', 'labelsTest');
    classNames = ["background", "tumor"];
    pixelLabelID = [0 1];
end
```

The `voldsTest` variable stores the ground truth test images. The `pxdsTest` variable stores the ground truth labels.

```
volReader = @(x) matRead(x);
voldsTest = imageDatastore(volLocTest, ...
    'FileExtensions', '.mat', 'ReadFcn', volReader);
pxdsTest = pixelLabelDatastore(lblLocTest, classNames, pixelLabelID, ...
    'FileExtensions', '.mat', 'ReadFcn', volReader);
```

Use the overlap-tile strategy to predict the labels for each test volume. Each test volume is padded to make the input size a multiple of the output size of the network and compensates for the effects of valid convolution. The overlap-tile algorithm selects overlapping patches, predicts the labels for each patch by using the `semanticseg` function, and then recombines the patches.

```
id = 1;
while hasdata(voldsTest)
    disp(['Processing test volume ' num2str(id)]);

    tempGroundTruth = read(pxdsTest);
    groundTruthLabels{id} = tempGroundTruth{1};
end
```

```

vol{id} = read(voldsTest);

% Use reflection padding for the test image.
% Avoid padding of different modalities.
volSize = size(vol{id},(1:3));
padSizePre = (inputPatchSize(1:3)-outPatchSize(1:3))/2;
padSizePost = (inputPatchSize(1:3)-outPatchSize(1:3))/2 + (outPatchSize(1:3)-mod(volSize,outPatchSize(1:3)));
volPaddedPre = padarray(vol{id},padSizePre,'symmetric','pre');
volPadded = padarray(volPaddedPre,padSizePost,'symmetric','post');
[heightPad,widthPad,depthPad,~] = size(volPadded);
[height,width,depth,~] = size(vol{id});

tempSeg = categorical(zeros([height,width,depth],'uint8'),[0;1],classNames);

% Overlap-tile strategy for segmentation of volumes.
for k = 1:outPatchSize(3):depthPad-inputPatchSize(3)+1
    for j = 1:outPatchSize(2):widthPad-inputPatchSize(2)+1
        for i = 1:outPatchSize(1):heightPad-inputPatchSize(1)+1
            patch = volPadded( i:i+inputPatchSize(1)-1,...
                               j:j+inputPatchSize(2)-1,...
                               k:k+inputPatchSize(3)-1,:);
            patchSeg = semanticseg(patch,net);
            tempSeg(i:i+outPatchSize(1)-1, ...
                   j:j+outPatchSize(2)-1, ...
                   k:k+outPatchSize(3)-1) = patchSeg;
        end
    end
end

% Crop out the extra padded region.
tempSeg = tempSeg(1:height,1:width,1:depth);

% Save the predicted volume result.
predictedLabels{id} = tempSeg;
id=id+1;
end

Processing test volume 1
Processing test volume 2
Processing test volume 3
Processing test volume 4
Processing test volume 5

```

Compare Ground Truth Against Network Prediction

Select one of the test images to evaluate the accuracy of the semantic segmentation. Extract the first modality from the 4-D volumetric data and store this 3-D volume in the variable `vol3d`.

```

volId = 1;
vol3d = vol{volId}(:,:,,1);

```

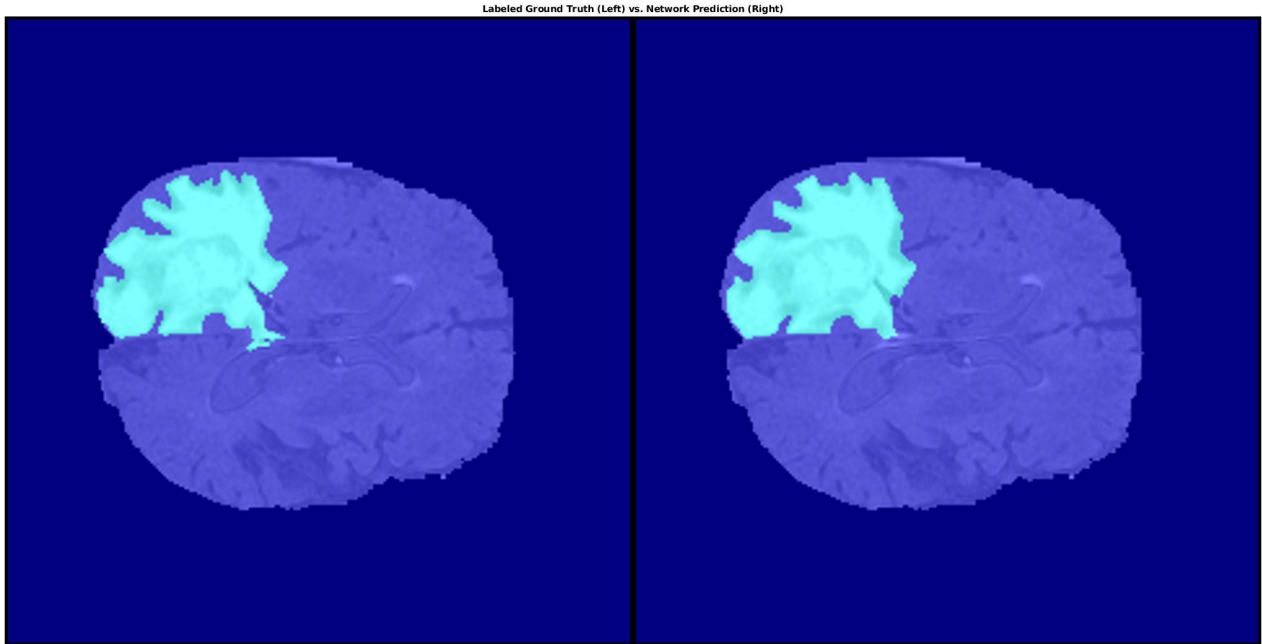
Display in a montage the center slice of the ground truth and predicted labels along the depth direction.

```

zID = size(vol3d,3)/2;
zSliceGT = labeloverlay(vol3d(:,:,zID),groundTruthLabels{volId}(:,:,zID));
zSlicePred = labeloverlay(vol3d(:,:,zID),predictedLabels{volId}(:,:,zID));

```

```
figure
montage({zSliceGT,zSlicePred},'Size',[1 2],'BorderSize',5)
title('Labeled Ground Truth (Left) vs. Network Prediction (Right)')
```

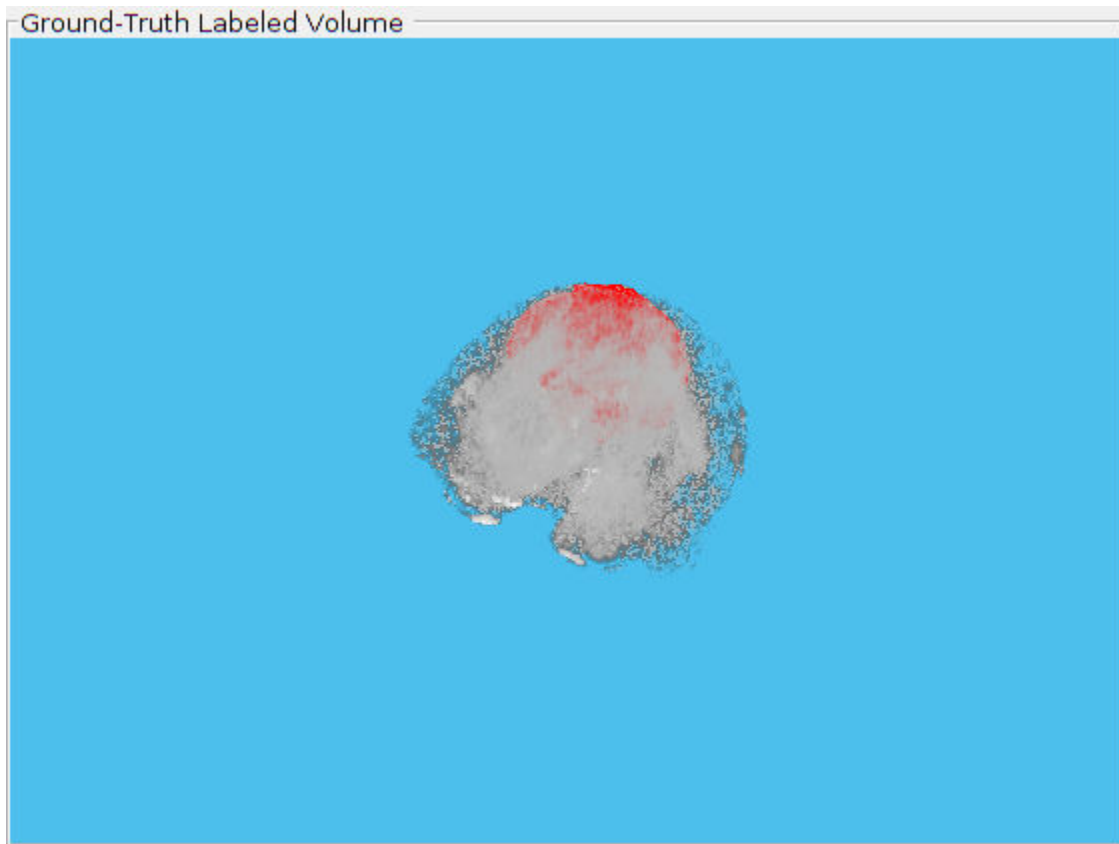


Display the ground-truth labeled volume using the `labelvolshow` function. Make the background fully transparent by setting the visibility of the background label (1) to 0. Because the tumor is inside the brain tissue, make some of the brain voxels transparent, so that the tumor is visible. To make some brain voxels transparent, specify the volume threshold as a number in the range [0, 1]. All normalized volume intensities below this threshold value are fully transparent. This example sets the volume threshold as less than 1 so that some brain pixels remain visible, to give context to the spatial location of the tumor inside the brain.

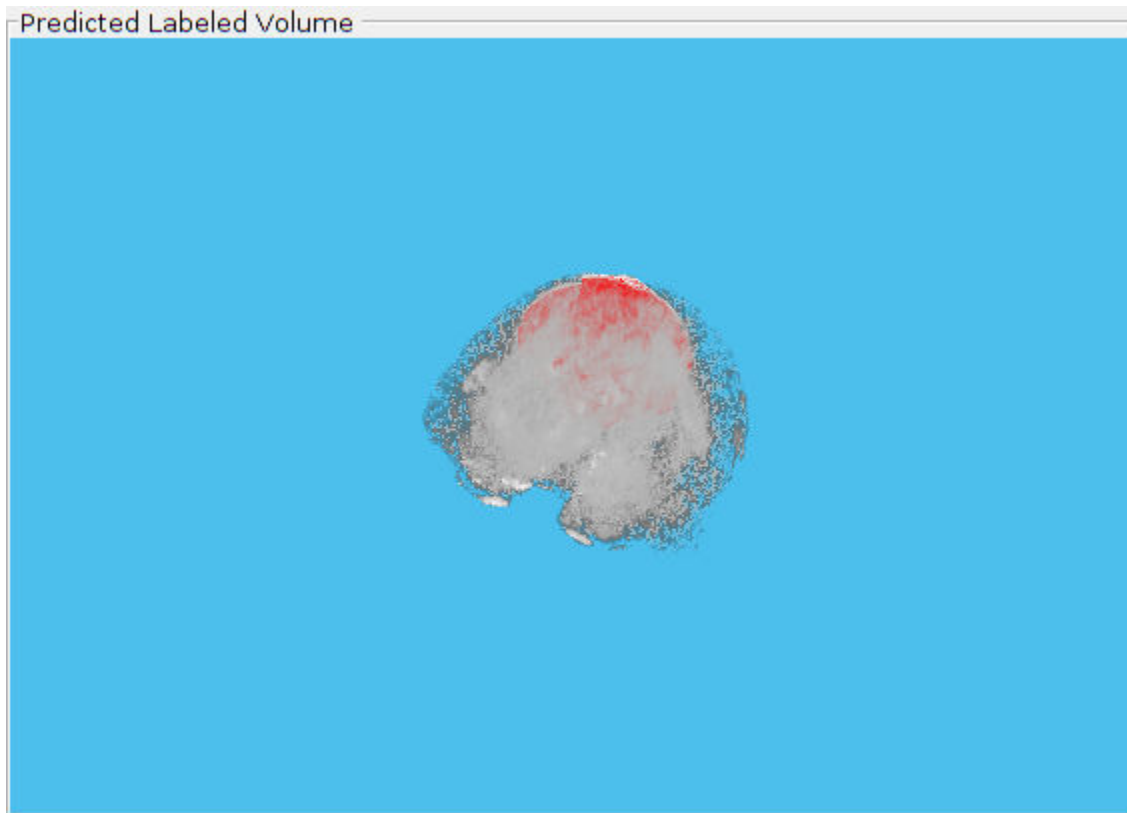
```
viewPnlTruth = uipanel(figure,'Title','Ground-Truth Labeled Volume');
hTruth = labelvolshow(groundTruthLabels{volId},vol3d,'Parent',viewPnlTruth, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
hTruth.LabelVisibility(1) = 0;
```

For the same volume, display the predicted labels.

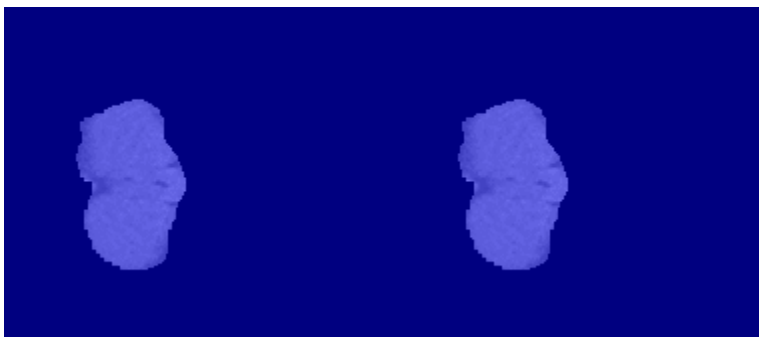
```
viewPnlPred = uipanel(figure,'Title','Predicted Labeled Volume');
hPred = labelvolshow(predictedLabels{volId},vol3d,'Parent',viewPnlPred, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
```



```
hPred.LabelVisibility(1) = 0;
```



This image shows the result of displaying slices sequentially across the one of the volume. The labeled ground truth is on the left and the network prediction is on the right.



Quantify Segmentation Accuracy

Measure the segmentation accuracy using the dice function. This function computes the Dice similarity coefficient between the predicted and ground truth segmentations.

```
diceResult = zeros(length(voldsTest.Files),2);  
for j = 1:length(vol)  
    diceResult(j,:) = dice(groundTruthLabels{j},predictedLabels{j});  
end
```

Calculate the average Dice score across the set of test volumes.

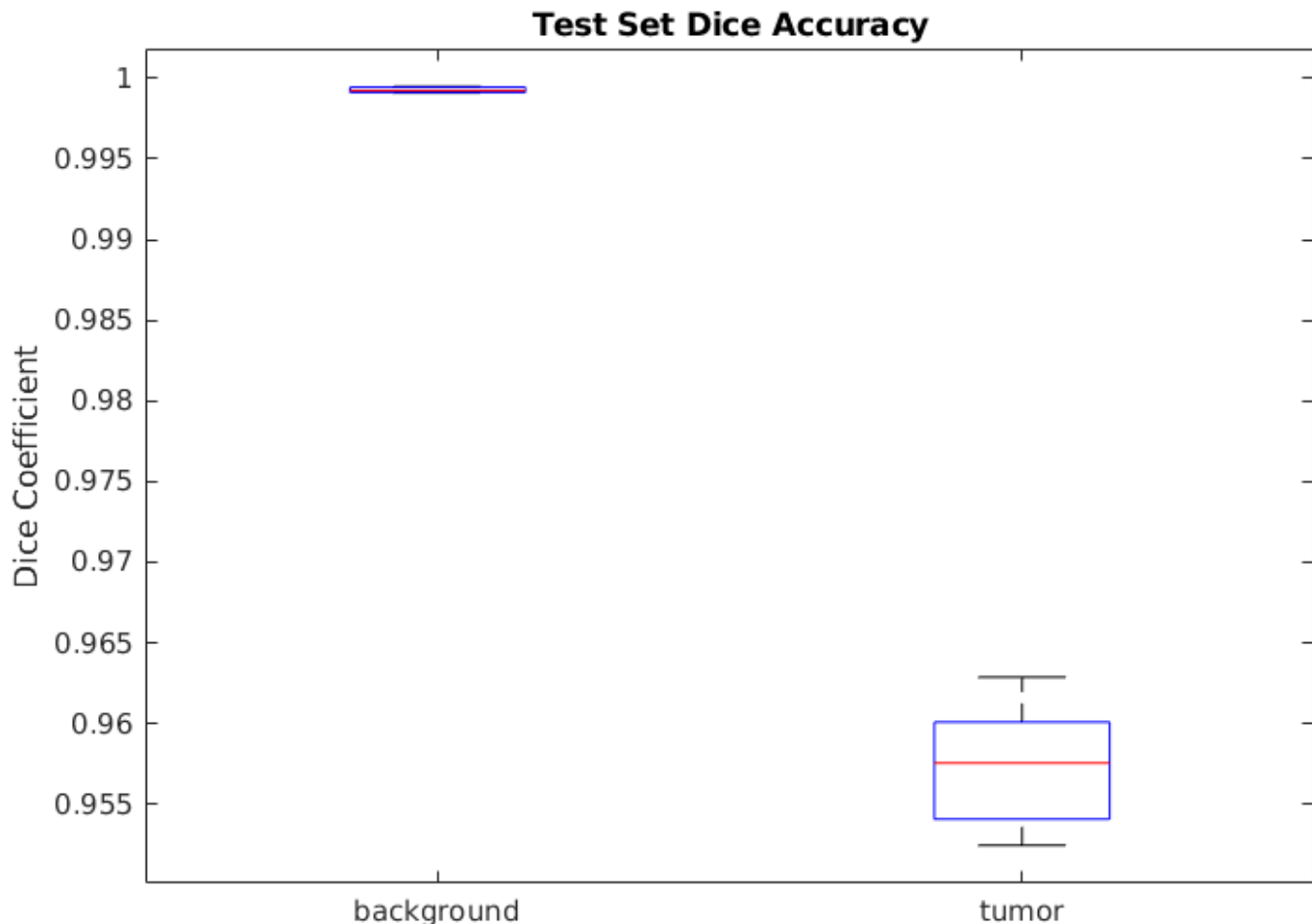
```
meanDiceBackground = mean(diceResult(:,1));
disp(['Average Dice score of background across ',num2str(j), ...
     ' test volumes = ',num2str(meanDiceBackground)])
```

Average Dice score of background across 5 test volumes = 0.9993

```
meanDiceTumor = mean(diceResult(:,2));
disp(['Average Dice score of tumor across ',num2str(j), ...
     ' test volumes = ',num2str(meanDiceTumor)])
```

Average Dice score of tumor across 5 test volumes = 0.9585

The figure shows a **boxplot** (Statistics and Machine Learning Toolbox) that visualizes statistics about the Dice scores across the set of five sample test volumes. The red lines in the plot show the median Dice value for the classes. The upper and lower bounds of the blue box indicate the 25th and 75th percentiles, respectively. Black whiskers extend to the most extreme data points not considered outliers.



If you have Statistics and Machine Learning Toolbox™, then you can use the `boxplot` function to visualize statistics about the Dice scores across all your test volumes. To create a boxplot, set the `createBoxplot` variable in the following code to `true`.

```
createBoxplot = false;
if createBoxplot
```

```
figure
boxplot(diceResult)
title('Test Set Dice Accuracy')
xticklabels(classNames)
ylabel('Dice Coefficient')
end
```

References

[1] Çiçek, Ö., A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger. "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation." In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention - MICCAI 2016*. Athens, Greece, Oct. 2016, pp. 424-432.

[2] Isensee, F., P. Kickingereder, W. Wick, M. Bendszus, and K. H. Maier-Hein. "Brain Tumor Segmentation and Radiomics Survival Prediction: Contribution to the BRATS 2017 Challenge." In *Proceedings of BrainLes: International MICCAI Brainlesion Workshop*. Quebec City, Canada, Sept. 2017, pp. 287-297.

[3] "Brain Tumours". *Medical Segmentation Decathlon*. <http://medicaldecathlon.com/>

The BraTS dataset is provided by Medical Segmentation Decathlon under the CC-BY-SA 4.0 license. All warranties and representations are disclaimed; see the license for details. MathWorks® has modified the data set linked in the Download Pretrained Network and Sample Test Set on page 3-0 section of this example. The modified sample dataset has been cropped to a region containing primarily the brain and tumor and each channel has been normalized independently by subtracting the mean and dividing by the standard deviation of the cropped brain region.

[4] Sudre, C. H., W. Li, T. Vercauteren, S. Ourselin, and M. J. Cardoso. "Generalised Dice Overlap as a Deep Learning Loss Function for Highly Unbalanced Segmentations." *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support: Third International Workshop*. Quebec City, Canada, Sept. 2017, pp. 240-248.

[5] Ronneberger, O., P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015*. Munich, Germany, Oct. 2015, pp. 234-241. Available at arXiv:1505.04597.

See Also

```
dicePixelClassificationLayer | imageDatastore | pixelLabelDatastore |
randomPatchExtractionDatastore | semanticseg | trainNetwork | trainingOptions |
transform
```

More About

- "Preprocess Volumes for Deep Learning" (Deep Learning Toolbox)
- "Datastores for Deep Learning" (Deep Learning Toolbox)
- "List of Deep Learning Layers" (Deep Learning Toolbox)

Image Category Classification Using Bag of Features

This example shows how to use a bag of features approach for image category classification. This technique is also often referred to as bag of words. Visual image categorization is a process of assigning a category label to an image under test. Categories may contain images representing just about anything, for example, dogs, cats, trains, boats.

Load Image Dataset

Unzip a collection of images to use for this example.

```
unzip('MerchData.zip');
```

Load the image collection using an `imageDatastore` to help you manage the data. Because `imageDatastore` operates on image file locations, and therefore does not load all the images into memory, it is safe to use on large image file collections.

```
imds = imageDatastore('MerchData', 'IncludeSubfolders', true, 'LabelSource', 'foldernames');
```

You can easily inspect the number of images per category as well as category labels as shown below:

```
tbl = countEachLabel(imds)
```

```
tbl=5x2 table
```

Label	Count
MathWorks Cap	15
MathWorks Cube	15
MathWorks Playing Cards	15
MathWorks Screwdriver	15
MathWorks Torch	15

Note that the labels were derived from directory names used to construct the `ImageDatastore`, but can be customized by manually setting the `Labels` property of the `ImageDatastore` object. Next, display a few of the images to get a sense of the type of images being used.

```
figure
montage(imds.Files(1:16:end))
```



Note that for the bag of features approach to be effective, the majority of the object must be visible in the image.

Prepare Training and Validation Image Sets

Separate the sets into training and validation data. Pick 60% of images from each set for the training data and the remainder, 40%, for the validation data. Randomize the split to avoid biasing the results.

```
[trainingSet, validationSet] = splitEachLabel(imds, 0.6, 'randomize');
```

The above call returns two `imageDatastore` objects ready for training and validation tasks.

Create a Visual Vocabulary and Train an Image Category Classifier

Bag of words is a technique adapted to computer vision from the world of natural language processing. Since images do not actually contain discrete words, we first construct a "vocabulary" of `extractFeatures` features representative of each image category.

This is accomplished with a single call to `bagOfFeatures` function, which:

- 1 extracts SURF features from all images in all image categories
- 2 constructs the visual vocabulary by reducing the number of features through quantization of feature space using K-means clustering

```
bag = bagOfFeatures(trainingSet);
```

```
Creating Bag-Of-Features.
```

```
-----
```

```
* Image category 1: MathWorks Cap
* Image category 2: MathWorks Cube
* Image category 3: MathWorks Playing Cards
* Image category 4: MathWorks Screwdriver
* Image category 5: MathWorks Torch
* Selecting feature point locations using the Grid method.
* Extracting SURF features from the selected feature point locations.
** The GridStep is [8 8] and the BlockWidth is [32 64 96 128].

* Extracting features from 45 images...done. Extracted 141120 features.

* Keeping 80 percent of the strongest features from each category.

* Using K-Means clustering to create a 500 word visual vocabulary.
* Number of features      : 112895
* Number of clusters (K)  : 500

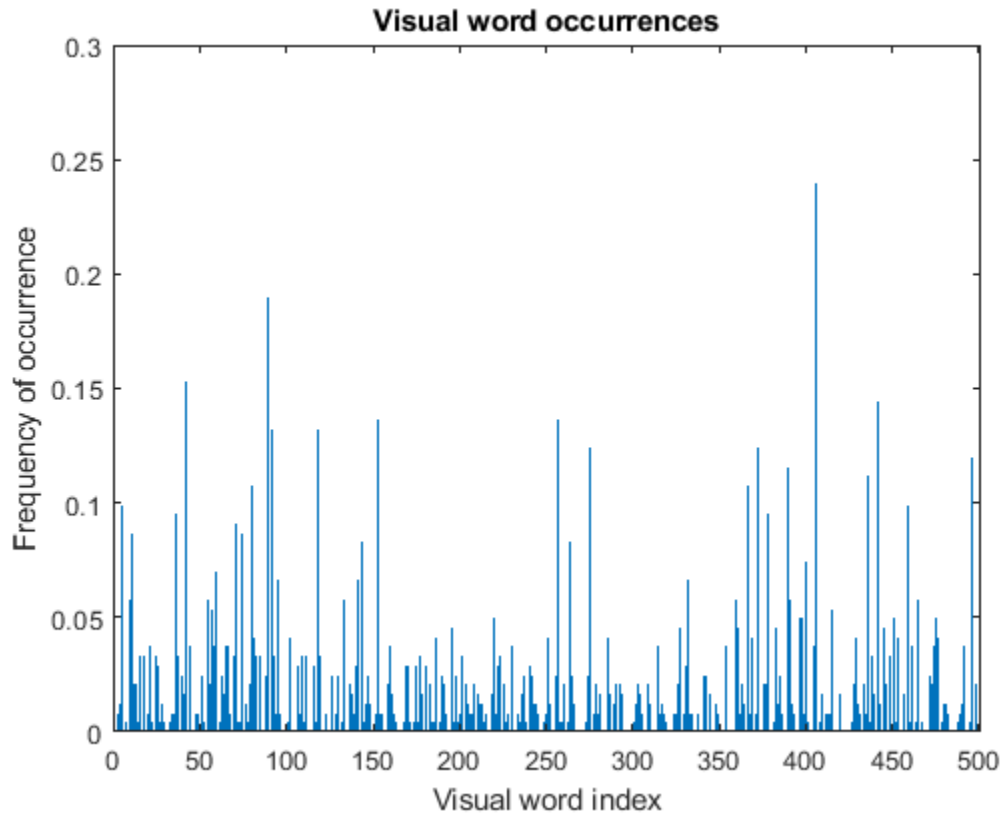
* Initializing cluster centers...100.00%.
* Clustering...completed 22/100 iterations (~0.43 seconds/iteration)...converged in 22 iterations

* Finished creating Bag-Of-Features
```

Additionally, the `bagOfFeatures` object provides an `encode` method for counting the visual word occurrences in an image. It produced a histogram that becomes a new and reduced representation of an image.

```
img = readimage(imds, 1);
featureVector = encode(bag, img);

% Plot the histogram of visual word occurrences
figure
bar(featureVector)
title('Visual word occurrences')
xlabel('Visual word index')
ylabel('Frequency of occurrence')
```



This histogram forms a basis for training a classifier and for the actual image classification. In essence, it encodes an image into a feature vector.

Encoded training images from each category are fed into a classifier training process invoked by the `trainImageCategoryClassifier` function. Note that this function relies on the multiclass linear SVM classifier from the Statistics and Machine Learning Toolbox™.

```
categoryClassifier = trainImageCategoryClassifier(trainingSet, bag);
```

```
Training an image category classifier for 5 categories.
```

```
-----
* Category 1: MathWorks Cap
* Category 2: MathWorks Cube
* Category 3: MathWorks Playing Cards
* Category 4: MathWorks Screwdriver
* Category 5: MathWorks Torch

* Encoding features for 45 images...done.

* Finished training the category classifier. Use evaluate to test the classifier on a test set.
```

The above function utilizes the `encode` method of the input `bag` object to formulate feature vectors representing each image category from the `trainingSet`.

Evaluate Classifier Performance

Now that we have a trained classifier, `categoryClassifier`, let's evaluate it. As a sanity check, let's first test it with the training set, which should produce near perfect confusion matrix, i.e. ones on the diagonal.

```
confMatrix = evaluate(categoryClassifier, trainingSet);
```

```
Evaluating image category classifier for 5 categories.
```

```
-----
* Category 1: MathWorks Cap
* Category 2: MathWorks Cube
* Category 3: MathWorks Playing Cards
* Category 4: MathWorks Screwdriver
* Category 5: MathWorks Torch
```

```
* Evaluating 45 images...done.
```

```
* Finished evaluating all the test sets.
```

```
* The confusion matrix for this test set is:
```

KNOWN	PREDICTED			
	MathWorks Cap	MathWorks Cube	MathWorks Playing Cards	MathWorks Torch
MathWorks Cap	1.00	0.00	0.00	0.00
MathWorks Cube	0.00	0.89	0.00	0.00
MathWorks Playing Cards	0.00	0.00	1.00	0.00
MathWorks Screwdriver	0.00	0.00	0.00	1.00
MathWorks Torch	0.00	0.00	0.00	0.00

```
* Average Accuracy is 0.98.
```

Next, let's evaluate the classifier on the `validationSet`, which was not used during the training. By default, the `evaluate` function returns the confusion matrix, which is a good initial indicator of how well the classifier is performing.

```
confMatrix = evaluate(categoryClassifier, validationSet);
```

```
Evaluating image category classifier for 5 categories.
```

```
-----
* Category 1: MathWorks Cap
* Category 2: MathWorks Cube
* Category 3: MathWorks Playing Cards
* Category 4: MathWorks Screwdriver
* Category 5: MathWorks Torch
```

```
* Evaluating 30 images...done.
```

```
* Finished evaluating all the test sets.
```

```
* The confusion matrix for this test set is:
```

PREDICTED

KNOWN	MathWorks Cap	MathWorks Cube	MathWorks Playing Cards	MathWorks Torch
MathWorks Cap	1.00	0.00	0.00	0.00
MathWorks Cube	0.00	0.67	0.17	0.17
MathWorks Playing Cards	0.00	0.00	1.00	0.00
MathWorks Screwdriver	0.00	0.00	0.00	1.00
MathWorks Torch	0.17	0.00	0.00	0.00

* Average Accuracy is 0.90.

```
% Compute average accuracy  
mean(diag(confMatrix))
```

```
ans = 0.9000
```

You can tune `bagOfFeatures` hyperparameters and continue evaluating the trained classifier until you are satisfied with the results. Additional statistics can be derived using the rest of arguments returned by the `evaluate` function. See help for `imageCategoryClassifier/evaluate`.

Try the Newly Trained Classifier on Test Images

You can now apply the newly trained classifier to categorize new images.

```
img = imread(fullfile('MerchData', 'MathWorks Cap', 'Hat_0.jpg'));  
figure  
imshow(img)
```



```
[labelIdx, scores] = predict(categoryClassifier, img);
```

```
% Display the string label  
categoryClassifier.Labels(labelIdx)
```

```
ans = 1x1 cell array  
    {'MathWorks Cap'}
```

Image Category Classification Using Deep Learning

This example shows how to use a pretrained Convolutional Neural Network (CNN) as a feature extractor for training an image category classifier.

Overview

A Convolutional Neural Network (CNN) is a powerful machine learning technique from the field of deep learning. CNNs are trained using large collections of diverse images. From these large collections, CNNs can learn rich feature representations for a wide range of images. These feature representations often outperform hand-crafted features such as HOG, LBP, or SURF. An easy way to leverage the power of CNNs, without investing time and effort into training, is to use a pretrained CNN as a feature extractor.

In this example, images from a Flowers Dataset[5] are classified into categories using a multiclass linear SVM trained with CNN features extracted from the images. This approach to image category classification follows the standard practice of training an off-the-shelf classifier using features extracted from images. For example, the “Image Category Classification Using Bag of Features” on page 3-93 example uses SURF features within a bag of features framework to train a multiclass SVM. The difference here is that instead of using image features such as HOG or SURF, features are extracted using a CNN.

Note: This example requires Deep Learning Toolbox™, Statistics and Machine Learning Toolbox™, and Deep Learning Toolbox™ Model for ResNet-50 Network .

Using a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for running this example. Use of a GPU requires the Parallel Computing Toolbox™.

Download Image Data

The category classifier will be trained on images from a Flowers Dataset [5].

```
% Location of the compressed data set
url = 'http://download.tensorflow.org/example_images/flower_photos.tgz';

% Store the output in a temporary folder
downloadFolder = tempdir;
filename = fullfile(downloadFolder, 'flower_dataset.tgz');
```

Note: Download time of the data depends on your internet connection. The next set of commands use MATLAB to download the data and will block MATLAB. 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.

```
% Uncompressed data set
imageFolder = fullfile(downloadFolder, 'flower_photos');

if ~exist(imageFolder, 'dir') % download only once
    disp('Downloading Flower Dataset (218 MB)...');
    websave(filename, url);
    untar(filename, downloadFolder)
end
```


Load Images

Load the dataset using an `ImageDatastore` to help you manage the data. Because `ImageDatastore` operates on image file locations, images are not loaded into memory until read, making it efficient for use with large image collections.

```
imds = imageDatastore(imageFolder, 'LabelSource', 'foldernames', 'IncludeSubfolders',true);
```

Below, you can see an example image from one of the categories included in the dataset. The displayed image is by Mario.

```
% Find the first instance of an image for each category  
daisy = find(imds.Labels == 'daisy', 1);  
  
figure  
imshow(readimage(imds,daisy))
```



The `imds` variable now contains the images and the category labels associated with each image. The labels are automatically assigned from the folder names of the image files. Use `countEachLabel` to summarize the number of images per category.

```
tbl = countEachLabel(imds)
```

```
tbl=5x2 table  
    Label    Count  
-----  
    daisy     633  
    dandelion  898  
    roses     641  
    sunflowers 699
```

```
tulips      799
```

Because `imds` above contains an unequal number of images per category, let's first adjust it, so that the number of images in the training set is balanced.

```
% Determine the smallest amount of images in a category
minSetCount = min(tbl{: ,2});

% Limit the number of images to reduce the time it takes
% run this example.
maxNumImages = 100;
minSetCount = min(maxNumImages,minSetCount);

% Use splitEachLabel method to trim the set.
imds = splitEachLabel(imds, minSetCount, 'randomize');

% Notice that each set now has exactly the same number of images.
countEachLabel(imds)
```

```
ans=5x2 table
  Label      Count
  -----  -
  daisy      100
  dandelion  100
  roses      100
  sunflowers 100
  tulips     100
```

Load pretrained Network

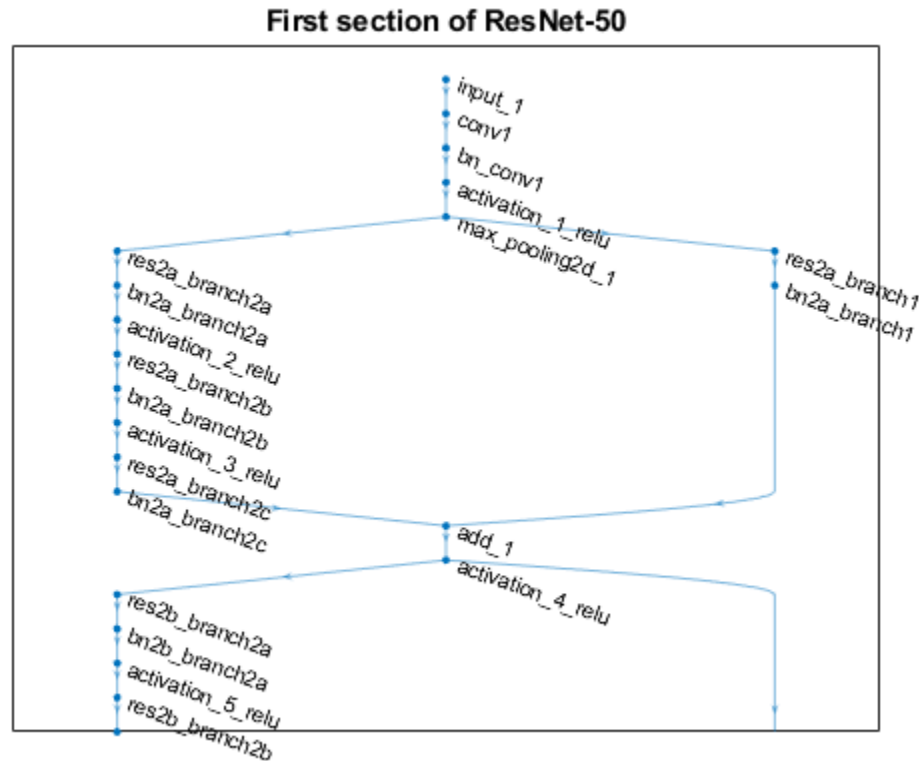
There are several pretrained networks that have gained popularity. Most of these have been trained on the ImageNet dataset, which has 1000 object categories and 1.2 million training images[1]. "ResNet-50" is one such model and can be loaded using the `resnet50` function from Neural Network Toolbox™. Using `resnet50` requires that you first install `resnet50` (Deep Learning Toolbox).

```
% Load pretrained network
net = resnet50();
```

Other popular networks trained on ImageNet include AlexNet, GoogLeNet, VGG-16 and VGG-19 [3], which can be loaded using `alexnet`, `googlenet`, `vgg16`, and `vgg19` from the Deep Learning Toolbox™.

Use `plot` to visualize the network. Because this is a large network, adjust the display window to show just the first section.

```
% Visualize the first section of the network.
figure
plot(net)
title('First section of ResNet-50')
set(gca, 'YLim', [150 170]);
```



The first layer defines the input dimensions. Each CNN has a different input size requirements. The one used in this example requires image input that is 224-by-224-by-3.

```
% Inspect the first layer
```

```
net.Layers(1)
```

```
ans =
```

```
ImageInputLayer with properties:
```

```
    Name: 'input_1'  
  InputSize: [224 224 3]
```

```
Hyperparameters
```

```
  DataAugmentation: 'none'  
    Normalization: 'zerocenter'  
NormalizationDimension: 'auto'  
          Mean: [224×224×3 single]
```

The intermediate layers make up the bulk of the CNN. These are a series of convolutional layers, interspersed with rectified linear units (ReLU) and max-pooling layers [2]. Following these layers are 3 fully-connected layers.

The final layer is the classification layer and its properties depend on the classification task. In this example, the CNN model that was loaded was trained to solve a 1000-way classification problem. Thus the classification layer has 1000 classes from the ImageNet dataset.

```
% Inspect the last layer
net.Layers(end)

ans =
  ClassificationOutputLayer with properties:

      Name: 'ClassificationLayer_fc1000'
      Classes: [1000x1 categorical]
      OutputSize: 1000

  Hyperparameters
      LossFunction: 'crossentropyex'
```

```
% Number of class names for ImageNet classification task
numel(net.Layers(end).ClassNames)
```

```
ans = 1000
```

Note that the CNN model is not going to be used for the original classification task. It is going to be re-purposed to solve a different classification task on the Flowers Dataset.

Prepare Training and Test Image Sets

Split the sets into training and validation data. Pick 30% of images from each set for the training data and the remainder, 70%, for the validation data. Randomize the split to avoid biasing the results. The training and test sets will be processed by the CNN model.

```
[trainingSet, testSet] = splitEachLabel(imds, 0.3, 'randomize');
```

Pre-process Images For CNN

As mentioned earlier, `net` can only process RGB images that are 224-by-224. To avoid re-saving all the images to this format, use an `augmentedImageDatastore` to resize and convert any grayscale images to RGB on-the-fly. The `augmentedImageDatastore` can be used for additional data augmentation as well when used for network training.

```
% Create augmentedImageDatastore from training and test sets to resize
% images in imds to the size required by the network.
imageSize = net.Layers(1).InputSize;
augmentedTrainingSet = augmentedImageDatastore(imageSize, trainingSet, 'ColorPreprocessing', 'gray2rgb');
augmentedTestSet = augmentedImageDatastore(imageSize, testSet, 'ColorPreprocessing', 'gray2rgb');
```

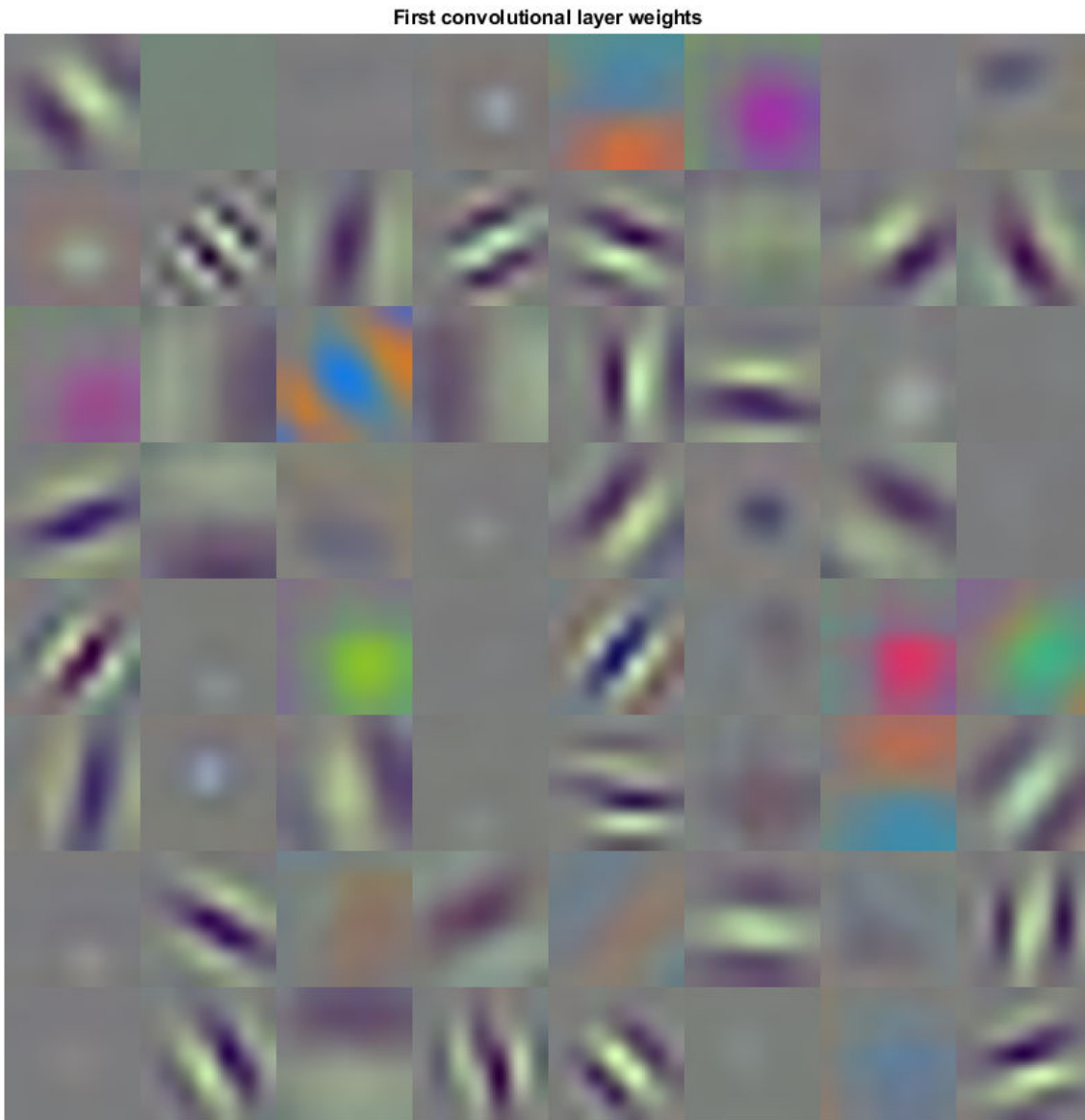
Extract Training Features Using CNN

Each layer of a CNN produces a response, or activation, to an input image. However, there are only a few layers within a CNN that are suitable for image feature extraction. The layers at the beginning of the network capture basic image features, such as edges and blobs. To see this, visualize the network filter weights from the first convolutional layer. This can help build up an intuition as to why the features extracted from CNNs work so well for image recognition tasks. Note that visualizing features from deeper layer weights can be done using `deepDreamImage` from Deep Learning Toolbox™.

```
% Get the network weights for the second convolutional layer
w1 = net.Layers(2).Weights;
```

```
% Scale and resize the weights for visualization
```

```
w1 = mat2gray(w1);  
w1 = imresize(w1,5);  
  
% Display a montage of network weights. There are 96 individual sets of  
% weights in the first layer.  
figure  
montage(w1)  
title('First convolutional layer weights')
```



Notice how the first layer of the network has learned filters for capturing blob and edge features. These "primitive" features are then processed by deeper network layers, which combine the early features to form higher level image features. These higher level features are better suited for

recognition tasks because they combine all the primitive features into a richer image representation [4].

You can easily extract features from one of the deeper layers using the `activations` method. Selecting which of the deep layers to choose is a design choice, but typically starting with the layer right before the classification layer is a good place to start. In `net`, this layer is named `'fc1000'`. Let's extract training features using that layer.

```
featureLayer = 'fc1000';
trainingFeatures = activations(net, augmentedTrainingSet, featureLayer, ...
    'MiniBatchSize', 32, 'OutputAs', 'columns');
```

Note that the `activations` function automatically uses a GPU for processing if one is available, otherwise, a CPU is used.

In the code above, the `'MiniBatchSize'` is set 32 to ensure that the CNN and image data fit into GPU memory. You may need to lower the `'MiniBatchSize'` if your GPU runs out of memory. Also, the `activations` output is arranged as columns. This helps speed-up the multiclass linear SVM training that follows.

Train A Multiclass SVM Classifier Using CNN Features

Next, use the CNN image features to train a multiclass SVM classifier. A fast Stochastic Gradient Descent solver is used for training by setting the `fitcecoc` function's `'Learners'` parameter to `'Linear'`. This helps speed-up the training when working with high-dimensional CNN feature vectors.

```
% Get training labels from the trainingSet
trainingLabels = trainingSet.Labels;

% Train multiclass SVM classifier using a fast linear solver, and set
% 'ObservationsIn' to 'columns' to match the arrangement used for training
% features.
classifier = fitcecoc(trainingFeatures, trainingLabels, ...
    'Learners', 'Linear', 'Coding', 'onevsall', 'ObservationsIn', 'columns');
```

Evaluate Classifier

Repeat the procedure used earlier to extract image features from `testSet`. The test features can then be passed to the classifier to measure the accuracy of the trained classifier.

```
% Extract test features using the CNN
testFeatures = activations(net, augmentedTestSet, featureLayer, ...
    'MiniBatchSize', 32, 'OutputAs', 'columns');

% Pass CNN image features to trained classifier
predictedLabels = predict(classifier, testFeatures, 'ObservationsIn', 'columns');

% Get the known labels
testLabels = testSet.Labels;

% Tabulate the results using a confusion matrix.
confMat = confusionmat(testLabels, predictedLabels);

% Convert confusion matrix into percentage form
confMat = bsxfun(@rdivide, confMat, sum(confMat, 2))

confMat = 5x5
```

```

0.8571    0.0286    0.0286    0.0714    0.0143
0.0571    0.8286         0    0.0571    0.0571
0.0143         0    0.7714    0.0714    0.1429
0.0286    0.0571    0.0571    0.8000    0.0571
         0         0    0.2000    0.0286    0.7714

```

```

% Display the mean accuracy
mean(diag(confMat))

```

```
ans = 0.8057
```

Apply the Trained Classifier On One Test Image

Apply the trained classifier to categorize new images. Read one of the "daisy" test images.

```

testImage = readimage(testSet,1);
testLabel = testSet.Labels(1)

```

```

testLabel = categorical
    daisy

```

Extract image features using the CNN.

```

% Create augmentedImageDatastore to automatically resize the image when
% image features are extracted using activations.
ds = augmentedImageDatastore(imageSize, testImage, 'ColorPreprocessing', 'gray2rgb');

```

```

% Extract image features using the CNN
imageFeatures = activations(net, ds, featureLayer, 'OutputAs', 'columns');

```

Make a prediction using the classifier.

```

% Make a prediction using the classifier
predictedLabel = predict(classifier, imageFeatures, 'ObservationsIn', 'columns')

```

```

predictedLabel = categorical
    daisy

```

References

[1] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009.

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[3] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[4] Donahue, Jeff, et al. "Decaf: A deep convolutional activation feature for generic visual recognition." arXiv preprint arXiv:1310.1531 (2013).

[5] Tensorflow: How to Retrain an Image Classifier for New Categories.

See Also

`activations` | `alexnet` | `classificationLayer` | `confusionmat` | `convolution2dLayer` | `countEachLabel` | `deepDreamImage` | `fitcecoc` | `fullyConnectedLayer` | `imageInputLayer` | `maxPooling2dLayer` | `predict` | `reluLayer`

More About

- “Image Category Classification Using Bag of Features” on page 3-93
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

Image Retrieval Using Customized Bag of Features

This example shows how to create a Content Based Image Retrieval (CBIR) system using a customized bag-of-features workflow.

Introduction

Content Based Image Retrieval (CBIR) systems are used to find images that are visually similar to a query image. The application of CBIR systems can be found in many areas such as a web-based product search, surveillance, and visual place identification. A common technique used to implement a CBIR system is bag of visual words, also known as bag of features [1,2]. Bag of features is a technique adapted to image retrieval from the world of document retrieval. Instead of using actual words as in document retrieval, bag of features uses image features as the visual words that describe an image.

Image features are an important part of CBIR systems. These image features are used to gauge similarity between images and can include global image features such as color, texture, and shape. Image features can also be local image features such as speeded up robust features (SURF), histogram of gradients (HOG), or local binary patterns (LBP). The benefit of the bag-of-features approach is that the type of features used to create the visual word vocabulary can be customized to fit the application.

The speed and efficiency of image search is also important in CBIR systems. For example, it may be acceptable to perform a brute force search in a small collection of images of less than a 100 images, where features from the query image are compared to features from each image in the collection. For larger collections, a brute force search is not feasible and more efficient search techniques must be used. The bag of features provides a concise encoding scheme to represent a large collection of images using a sparse set of visual word histograms. This enables compact storage and efficient search through an inverted index data structure.

The Computer Vision Toolbox™ provides a customizable bag-of-features framework to implement an image retrieval system. The following steps outline the procedure:

- 1 Select the Image Features for Retrieval
- 2 Create a Bag Of Features
- 3 Index the Images
- 4 Search for Similar Images

In this example, you will go through these steps to create an image retrieval system for searching a flower dataset [3]. This dataset contains about 3670 images of 5 different types of flowers.

Download this dataset for use in the rest of this example.

```
% Location of the compressed data set
url = 'http://download.tensorflow.org/example_images/flower_photos.tgz';

% Store the output in a temporary folder
downloadFolder = tempdir;
filename = fullfile(downloadFolder, 'flower_dataset.tgz');
```

Note that downloading the dataset from the web can take a very long time depending on your Internet connection. The commands below will block MATLAB for that period of time. Alternatively, you can use your web browser to first download the set to your local disk. If you choose that route, re-point the 'url' variable above to the file that you downloaded.

```
% Uncompressed data set
imageFolder = fullfile(downloadFolder, 'flower_photos');

if ~exist(imageFolder, 'dir') % download only once
    disp('Downloading Flower Dataset (218 MB)...');
    websave(filename, url);
    untar(filename, downloadFolder)
end

flowerImageSet = imageDatastore(imageFolder, 'LabelSource', 'foldernames', 'IncludeSubfolders', true);

% Total number of images in the data set
numel(flowerImageSet.Files)

ans = 3670
```

Step 1 - Select the Image Features for Retrieval

The type of feature used for retrieval depends on the type of images within the collection. For example, if searching an image collection made up of scenes (beaches, cities, highways), it is preferable to use a global image feature, such as a color histogram that captures the color content of the entire scene. However, if the goal is to find specific objects within the image collections, then local image features extracted around object keypoints are a better choice.

Let's start by viewing one of images to get an idea of how to approach the problem.

```
% Display a one of the flower images
figure
I = imread(flowerImageSet.Files{1});
imshow(I);
```



The displayed image is by Mario.

In this example, the goal is to search for similar flowers in the dataset using the color information in the query image. A simple image feature based on the spatial layout of color is a good place to start.

The following function describes the algorithm used to extract color features from a given image. This function will be used as a “extractorFcn” within `bagOfFeatures` to extract color features.

type `exampleBagOfFeaturesColorExtractor.m`

```
function [features, metrics] = exampleBagOfFeaturesColorExtractor(I)
% Example color layout feature extractor. Designed for use with bagOfFeatures.
%
% Local color layout features are extracted from truecolor image, I and
% returned in features. The strength of the features are returned in
% metrics.

[~,~,P] = size(I);

isColorImage = P == 3;

if isColorImage

    % Convert RGB images to the L*a*b* colorspace. The L*a*b* colorspace
    % enables you to easily quantify the visual differences between colors.
    % Visually similar colors in the L*a*b* colorspace will have small
    % differences in their L*a*b* values.
    Ilab = rgb2lab(I);

    % Compute the "average" L*a*b* color within 16-by-16 pixel blocks. The
    % average value is used as the color portion of the image feature. An
    % efficient method to approximate this averaging procedure over
    % 16-by-16 pixel blocks is to reduce the size of the image by a factor
    % of 16 using IMRESIZE.
    Ilab = imresize(Ilab, 1/16);

    % Note, the average pixel value in a block can also be computed using
    % standard block processing or integral images.

    % Reshape L*a*b* image into "number of features"-by-3 matrix.
    [Mr,Nr,~] = size(Ilab);
    colorFeatures = reshape(Ilab, Mr*Nr, []);

    % L2 normalize color features
    rowNorm = sqrt(sum(colorFeatures.^2,2));
    colorFeatures = bsxfun(@rdivide, colorFeatures, rowNorm + eps);

    % Augment the color feature by appending the [x y] location within the
    % image from which the color feature was extracted. This technique is
    % known as spatial augmentation. Spatial augmentation incorporates the
    % spatial layout of the features within an image as part of the
    % extracted feature vectors. Therefore, for two images to have similar
    % color features, the color and spatial distribution of color must be
    % similar.

    % Normalize pixel coordinates to handle different image sizes.
    xnorm = linspace(-0.5, 0.5, Nr);
    ynorm = linspace(-0.5, 0.5, Mr);
    [x, y] = meshgrid(xnorm, ynorm);
```

```
% Concatenate the spatial locations and color features.
features = [colorFeatures y(:) x(:)];

% Use color variance as feature metric.
metrics = var(colorFeatures(:,1:3),0,2);
else

% Return empty features for non-color images. These features are
% ignored by bagOfFeatures.
features = zeros(0,5);
metrics = zeros(0,1);
end
```

Step 2 - Create a Bag Of Features

With the feature type defined, the next step is to learn the visual vocabulary within the `bagOfFeatures` using a set of training images. The code shown below picks a random subset of images from the dataset for training and then trains `bagOfFeatures` using the 'CustomExtractor' option.

Set `doTraining` to false to load a pretrained `bagOfFeatures`. `doTraining` is set to false because the training process takes several minutes. The rest of the example uses a pre-trained `bagOfFeatures` to save time. If you wish to recreate `colorBag` locally, set `doTraining` to true and consider "Computer Vision Toolbox Preferences" to reduce processing time.

```
doTraining = false;

if doTraining
    %Pick a random subset of the flower images
    trainingSet = splitEachLabel(flowerImageSet, 0.6, 'randomized');

    % Create a custom bag of features using the 'CustomExtractor' option
    colorBag = bagOfFeatures(trainingSet, ...
        'CustomExtractor', @exampleBagOfFeaturesColorExtractor, ...
        'VocabularySize', 5000);
else
    % Load a pretrained bagOfFeatures
    load('savedColorBagOfFeatures.mat','colorBag');
end
```

Step 3 - Index the Images

Now that the `bagOfFeatures` is created, the entire flower image set can be indexed for search. The indexing procedure extracts features from each image using the custom extractor function from step 1. The extracted features are encoded into a visual word histogram and added into the image index.

```
if doTraining
    % Create a search index
    flowerImageIndex = indexImages(flowerImageSet,colorBag,'SaveFeatureLocations',false);
else
    % Load a saved index
    load('savedColorBagOfFeatures.mat','flowerImageIndex');
end
```

Because the indexing step processes thousands of images, the rest of this example uses a saved index to save time. You may recreate the index locally by setting `doTraining` to true.

Step 4 - Search for Similar Images

The final step is to use the `retrieveImages` function to search for similar images.

```
% Define a query image
queryImage = readimage(flowerImageSet,200);

figure
imshow(queryImage)
```



The displayed image is by RetinaFunk.

```
% Search for the top 5 images with similar color content
[imageIDs, scores] = retrieveImages(queryImage, flowerImageIndex, 'NumResults',5);
```

`retrieveImages` returns the image IDs and the scores of each result. The scores are sorted from best to worst.

```
scores
scores = 5x1
    0.4822
    0.2143
    0.1389
    0.1384
    0.1320
```

The `imageIDs` correspond to the images within the image set that are similar to the query image.

```
% Display results using montage.
figure
montage(flowerImageSet.Files(imageIDs), 'ThumbnailSize', [200 200])
```



The displayed images are by RetinaFunk, Jenny Downing, Mayeesherr, daBinsi, and Steve Snodgrass.

Conclusion

This example showed you how to customize the `bagOfFeatures` and how to use `indexImages` and `retrieveImages` to create an image retrieval system based on color features. The techniques shown here may be extended to other feature types by further customizing the features used within `bagOfFeatures`.

References

- [1] Sivic, J., Zisserman, A.: Video Google: A text retrieval approach to object matching in videos. In: ICCV. (2003) 1470-1477
- [2] Philbin, J., Chum, O., Isard, M., A., J.S., Zisserman: Object retrieval with large vocabularies and fast spatial matching. In: CVPR. (2007)
- [3] TensorFlow: How to Retrain an Image Classifier for New Categories.

Create SSD Object Detection Network

This example shows how to modify a pretrained MobileNet v2 network to create a SSD object detection network.

The procedure to convert a pretrained network into a SSD network is similar to the transfer learning procedure for image classification:

- 1 Load the pretrained network.
- 2 Select one or more layers from the pretrained network to use for feature extraction.
- 3 Remove all layers after the feature extraction layers
- 4 Add new layers to support the object detection task.

Load Pretrained Network

Load a pretrained MobileNet v2 network using `mobilenetv2`. This requires the Deep Learning Toolbox Model for MobileNet v2 Network™ support package. If this support package is not installed, then the function provides a download link. After you load the network, convert the network into a `layerGraph` object so that you can manipulate the layers.

```
net = mobilenetv2();
lgraph = layerGraph(net);
```

Update Network Input Size

Update the network input size to meet the training data requirements. For example, assume the training data are 300-by-300 RGB images. Set the input size.

```
imageInputSize = [300 300 3];
```

Next, create a new image input layer with the same name as the original layer.

```
imgLayer = imageInputLayer(imageInputSize,"Name","input_1");
```

Replace the old image input layer with the new image input layer.

```
lgraph = replaceLayer(lgraph,"input_1",imgLayer);
```

Select Feature Extraction Layers

SSD predict object locations using multiple feature maps. Typically, you choose feature extraction layers with different output sizes to leverage the benefit of multi-scale features. You can use the `analyzeNetwork` function or the Deep Network Designer app to determine the output sizes of layers within a network. Note that selecting an optimal set feature extraction layers requires empirical evaluation.

For brevity, this example illustrates the use one feature extraction layer. Set the feature extraction layer to "block_12_add".

```
featureExtractionLayer = "block_12_add";
```

Remove Layers After Feature Extraction Layer

Next, remove the layers after the feature extraction layer. You can do so by importing the network into the Deep Network Designer app, manually removing the layers, and exporting the modified the network to your workspace.

For this example, load the modified network, which has been added to this example as a supporting file.

```
modified = load("mobilenetv2Block12Add.mat");  
lgraph = modified.mobilenetv2Block12Add;
```

Attach AnchorBoxLayer

Specify the anchor boxes and number of object classes and use anchorBoxLayer to create an anchor box layer.

```
numClasses = 5;  
  
anchorBoxes = [  
    16 16  
    32 16  
];  
  
anchorBox = anchorBoxLayer(anchorBoxes, "Name", "anchors");
```

Attach the anchor box layer to the feature extraction layer.

```
lgraph = addLayers(lgraph, anchorBox);  
lgraph = connectLayers(lgraph, "block_12_add", "anchors");
```

Create SSD Classification Branch

Create a convolution layer where the number of convolution filters equals the numAnchors times the numClasses + 1. The additional class represents the background class.

```
numAnchors = size(anchorBoxes, 1);  
numClassesPlusBackground = numClasses + 1;  
numClsFilters = numAnchors * numClassesPlusBackground;  
filterSize = 3;  
conv = convolution2dLayer(filterSize, numClsFilters, ...  
    "Name", "convClassification", ...  
    "Padding", "same");
```

Add and connect the convolution layer to the anchor box layer.

```
lgraph = addLayers(lgraph, conv);  
lgraph = connectLayers(lgraph, "anchors", "convClassification");
```

Create SSD Regression Branch

Create a convolution layer where the number of convolution filters equals the four times number of anchor boxes.

```
numRegFilters = 4 * numAnchors;  
conv = convolution2dLayer(filterSize, numRegFilters, ...  
    "Name", "convRegression", ...  
    "Padding", "same");
```

Add and connect the convolution layer to the anchor box layer.

```
lgraph = addLayers(lgraph, conv);  
lgraph = connectLayers(lgraph, "anchors", "convRegression");
```


Merge Classification Features

Create an `ssdMergeLayer` initialized with the number of classes and the number of feature extraction layers.

```
numFeatureExtractionLayers = numel(featureExtractionLayer);
mergeClassification = ssdMergeLayer(numClassesPlusBackground,numFeatureExtractionLayers,...
    "Name","mergeClassification");
```

Add and connect the SSD merge layer to the `convClassification` layer.

```
lgraph = addLayers(lgraph,mergeClassification);
lgraph = connectLayers(lgraph,"convClassification","mergeClassification/in1");
```

Merge Regression Features

Create an `ssdMergeLayer` initialized with the number of coordinate offsets used to refine anchor box positions and the number of feature extraction layers.

```
numCoordinates = 4;
mergeRegression = ssdMergeLayer(numCoordinates,numFeatureExtractionLayers,...
    "Name","mergeRegression");
```

Add and connect the SSD merge layer to the `convRegression` layer.

```
lgraph = addLayers(lgraph,mergeRegression);
lgraph = connectLayers(lgraph,"convRegression","mergeRegression/in1");
```

Complete SSD Detection Network

To complete the classification branch, create and attach a softmax layer and a focal loss layer.

```
clsLayers = [
    softmaxLayer("Name","softmax")
    focalLossLayer("Name","focalLoss")
];
```

```
lgraph = addLayers(lgraph,clsLayers);
lgraph = connectLayers(lgraph,"mergeClassification","softmax");
```

To complete the regression branch, create and attach a box regression layer.

```
reg = rcnnBoxRegressionLayer("Name","boxRegression");

lgraph = addLayers(lgraph,reg);
lgraph = connectLayers(lgraph,"mergeRegression","boxRegression");
```

Use `analyzeNetwork` to check the network.

```
analyzeNetwork(lgraph)
```

The SSD network is complete and can be trained using the `trainSSDObjectDetector` function.

Train YOLO v2 Network for Vehicle Detection

Load the training data for vehicle detection into the workspace.

```
data = load('vehicleTrainingData.mat');
trainingData = data.vehicleTrainingData;
```

Specify the directory in which training samples are stored. Add full path to the file names in training data.

```
dataDir = fullfile(toolboxdir('vision'),'visiondata');
trainingData.imageFilename = fullfile(dataDir,trainingData.imageFilename);
```

Randomly shuffle data for training.

```
rng(0);
shuffledIdx = randperm(height(trainingData));
trainingData = trainingData(shuffledIdx,:);
```

Create an imageDatastore using the files from the table.

```
imds = imageDatastore(trainingData.imageFilename);
```

Create a boxLabelDatastore using the label columns from the table.

```
blds = boxLabelDatastore(trainingData(:,2:end));
```

Combine the datastores.

```
ds = combine(imds, blds);
```

Load a preinitialized YOLO v2 object detection network.

```
net = load('yolov2VehicleDetector.mat');
lgraph = net.lgraph
```

```
lgraph =
  LayerGraph with properties:
    Layers: [25x1 nnet.cnn.layer.Layer]
    Connections: [24x2 table]
    InputNames: {'input'}
    OutputNames: {'yolov2OutputLayer'}
```

Inspect the layers in the YOLO v2 network and their properties. You can also create the YOLO v2 network by following the steps given in “Create YOLO v2 Object Detection Network” on page 3-180.

```
lgraph.Layers
```

```
ans =
  25x1 Layer array with layers:

    1 'input'           Image Input           128x128x3 images
    2 'conv_1'         Convolution           16 3x3 convolutions with stride [1 1]
    3 'BN1'            Batch Normalization   Batch normalization
    4 'relu_1'         ReLU                  ReLU
    5 'maxpool1'       Max Pooling           2x2 max pooling with stride [2 2] and
```

6	'conv_2'	Convolution	32 3x3 convolutions with stride [1 1]
7	'BN2'	Batch Normalization	Batch normalization
8	'relu_2'	ReLU	ReLU
9	'maxpool2'	Max Pooling	2x2 max pooling with stride [2 2] and
10	'conv_3'	Convolution	64 3x3 convolutions with stride [1 1]
11	'BN3'	Batch Normalization	Batch normalization
12	'relu_3'	ReLU	ReLU
13	'maxpool3'	Max Pooling	2x2 max pooling with stride [2 2] and
14	'conv_4'	Convolution	128 3x3 convolutions with stride [1 1]
15	'BN4'	Batch Normalization	Batch normalization
16	'relu_4'	ReLU	ReLU
17	'yolov2Conv1'	Convolution	128 3x3 convolutions with stride [1 1]
18	'yolov2Batch1'	Batch Normalization	Batch normalization
19	'yolov2Relu1'	ReLU	ReLU
20	'yolov2Conv2'	Convolution	128 3x3 convolutions with stride [1 1]
21	'yolov2Batch2'	Batch Normalization	Batch normalization
22	'yolov2Relu2'	ReLU	ReLU
23	'yolov2ClassConv'	Convolution	24 1x1 convolutions with stride [1 1]
24	'yolov2Transform'	YOLO v2 Transform Layer.	YOLO v2 Transform Layer with 4 anchors
25	'yolov2OutputLayer'	YOLO v2 Output	YOLO v2 Output with 4 anchors.

Configure the network training options.

```
options = trainingOptions('sgdm',...
    'InitialLearnRate',0.001,...
    'Verbose',true,...
    'MiniBatchSize',16,...
    'MaxEpochs',30,...
    'Shuffle','never',...
    'VerboseFrequency',30,...
    'CheckpointPath',tempdir);
```

Train the YOLO v2 network.

```
[detector,info] = trainYOLOv2ObjectDetector(ds,lgraph,options);
```

```
*****
Training a YOLO v2 Object Detector for the following object classes:
```

```
* vehicle
```

Training on single CPU.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch RMSE	Mini-batch Loss	Base Learning Rate
1	1	00:00:01	7.13	50.8	0.0010
2	30	00:00:14	1.35	1.8	0.0010
4	60	00:00:27	1.13	1.3	0.0010
5	90	00:00:39	0.64	0.4	0.0010
7	120	00:00:51	0.65	0.4	0.0010
9	150	00:01:04	0.72	0.5	0.0010
10	180	00:01:16	0.52	0.3	0.0010
12	210	00:01:28	0.45	0.2	0.0010
14	240	00:01:41	0.61	0.4	0.0010
15	270	00:01:52	0.43	0.2	0.0010
17	300	00:02:05	0.42	0.2	0.0010
19	330	00:02:17	0.52	0.3	0.0010

20	360	00:02:29	0.43	0.2	0.0010
22	390	00:02:42	0.43	0.2	0.0010
24	420	00:02:54	0.59	0.4	0.0010
25	450	00:03:06	0.61	0.4	0.0010
27	480	00:03:18	0.65	0.4	0.0010
29	510	00:03:31	0.48	0.2	0.0010
30	540	00:03:42	0.34	0.1	0.0010

Detector training complete.

Inspect the properties of the detector.

```
detector
```

```
detector =
```

```
  yolov2objectDetector with properties:
```

```
    ModelName: 'vehicle'
```

```
    Network: [1x1 DAGNetwork]
```

```
  TrainingImageSize: [128 128]
```

```
    AnchorBoxes: [4x2 double]
```

```
    ClassNames: vehicle
```

You can verify the training accuracy by inspecting the training loss for each iteration.

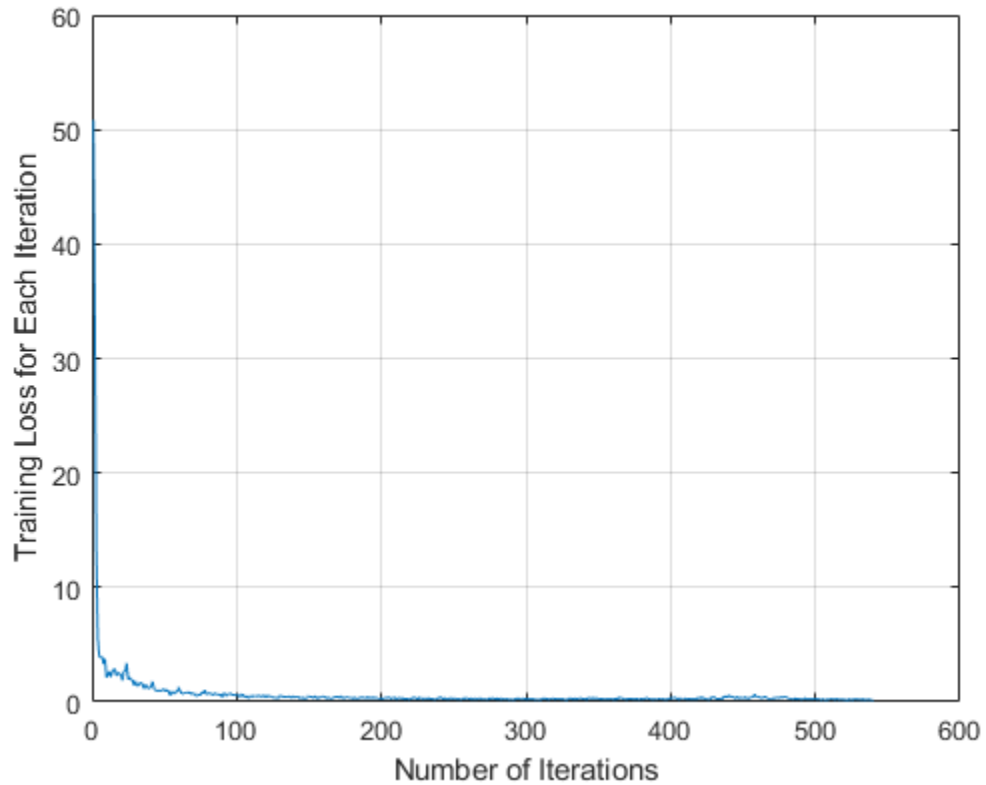
```
figure
```

```
plot(info.TrainingLoss)
```

```
grid on
```

```
xlabel('Number of Iterations')
```

```
ylabel('Training Loss for Each Iteration')
```



Read a test image into the workspace.

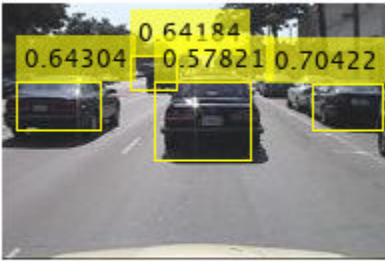
```
img = imread('detectcars.png');
```

Run the trained YOLO v2 object detector on the test image for vehicle detection.

```
[bboxes,scores] = detect(detector,img);
```

Display the detection results.

```
if(~isempty(bboxes))  
    img = insertObjectAnnotation(img,'rectangle',bboxes,scores);  
end  
figure  
imshow(img)
```



Object Detection Using YOLO v2 Deep Learning

This example shows how to train a you only look once (YOLO) v2 object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a YOLO v2 vehicle detector using the `trainYOLOv2ObjectDetector` function. For more information, see “Object Detection using Deep Learning”.

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('yolov2ResNet50VehicleExample_19b.mat','file')
    disp('Downloading pretrained detector (98 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/yolov2ResNet50VehicleExample_19b.mat';
    websave('yolov2ResNet50VehicleExample_19b.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small vehicle dataset that contains 295 images. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the YOLO v2 training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip('vehicleDatasetImages.zip');
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

```
% Display first few rows of the data set.
vehicleDataset(1:4,:)
```

```
ans=4x2 table
```

imageFilename	vehicle
{'vehicleImages/image_00001.jpg'}	{1x4 double}
{'vehicleImages/image_00002.jpg'}	{1x4 double}
{'vehicleImages/image_00003.jpg'}	{1x4 double}
{'vehicleImages/image_00004.jpg'}	{1x4 double}

```
% Add the fullpath to the local vehicle data folder.
vehicleDataset.imageFilename = fullfile(pwd,vehicleDataset.imageFilename);
```

Split the dataset into training, validation, and test sets. Select 60% of the data for training, 10% for validation, and the rest for testing the trained detector.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
```

```
idx = floor(0.6 * length(shuffledIndices) );  
  
trainingIdx = 1:idx;  
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);  
  
validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );  
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);  
  
testIdx = validationIdx(end)+1 : length(shuffledIndices);  
testDataTbl = vehicleDataset(shuffledIndices(testIdx),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingDataTbl{:, 'imageFilename'});  
blsTrain = boxLabelDatastore(trainingDataTbl(:, 'vehicle'));  
  
imdsValidation = imageDatastore(validationDataTbl{:, 'imageFilename'});  
blsValidation = boxLabelDatastore(validationDataTbl(:, 'vehicle'));  
  
imdsTest = imageDatastore(testDataTbl{:, 'imageFilename'});  
blsTest = boxLabelDatastore(testDataTbl(:, 'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,blsTrain);  
validationData = combine(imdsValidation,blsValidation);  
testData = combine(imdsTest,blsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);  
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I, 'Rectangle', bbox);  
annotatedImage = imresize(annotatedImage,2);  
figure  
imshow(annotatedImage)
```




Create a YOLO v2 Object Detection Network

A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. The feature extraction network is typically a pretrained CNN (for details, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox)). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18 can also be used depending on application requirements. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific for YOLO v2.

Use the `yoLov2Layers` function to create a YOLO v2 object detection network automatically given a pretrained ResNet-50 feature extraction network. `yoLov2Layers` requires you to specify several inputs that parameterize a YOLO v2 network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size and the number of classes. When choosing the network input size, consider the minimum size required by the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [224 224 3], which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Define the number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes. Resize the training data to the input image size of the network using the supporting function `preprocessData`.

```
trainingDataForEstimation = transform(trainingData,@(data)preprocessData(data,inputSize));  
numAnchors = 7;  
[anchorBoxes, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchorBoxes = 7×2
```

```
    145    126  
     91     86  
    161    132  
     41     34  
     67     64  
    136    111  
     33     23
```

```
meanIoU = 0.8651
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” on page 3-146 (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” on page 14-21.

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select `'activation_40_relu'` as the feature extraction layer to replace the layers after `'activation_40_relu'` with the detection subnetwork. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis.

```
featureLayer = 'activation_40_relu';
```

Create the YOLO v2 object detection network.

```
lgraph = yolov2Layers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer);
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the YOLO v2 network architecture, use Deep Network Designer to design the YOLO v2 detection network manually. For more information, see “Design a YOLO v2 Detection Network” on page 14-27.

Data Augmentation

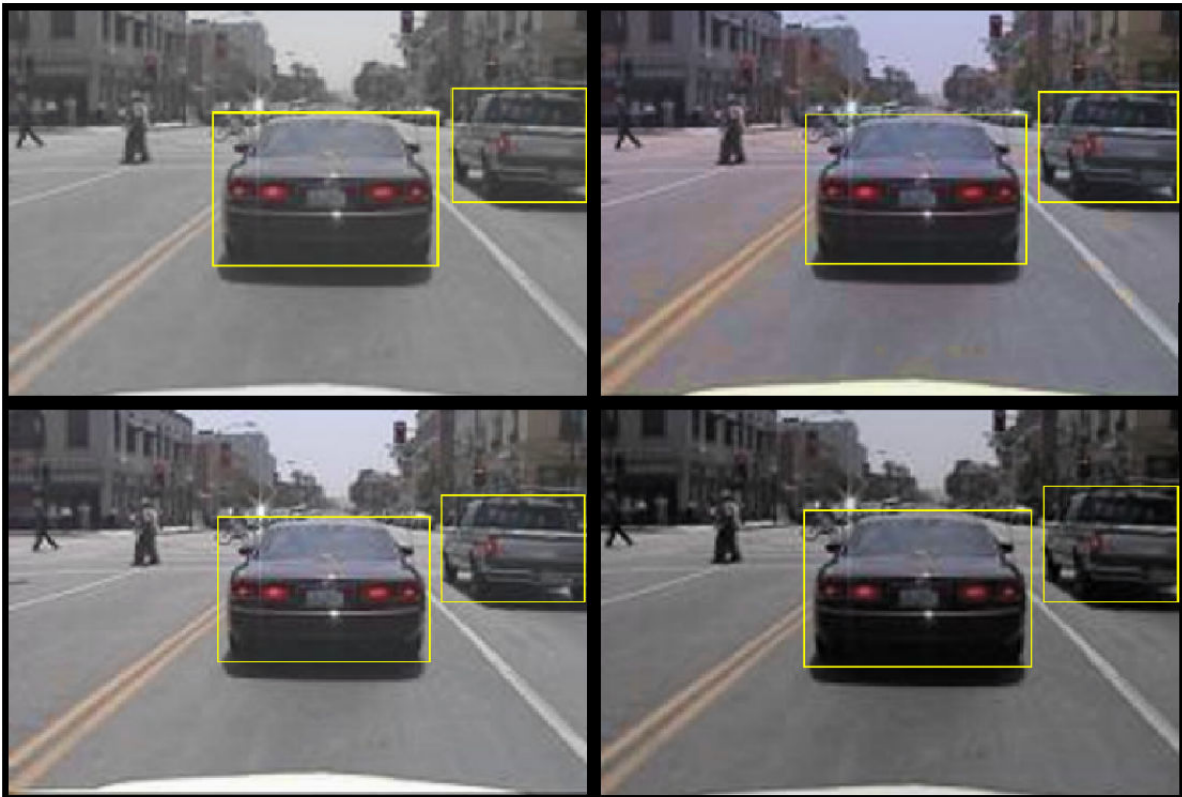
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
% Visualize the augmented images.
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1}, 'Rectangle', data{2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData, 'BorderSize', 10)
```



Preprocess Training Data

Preprocess the augmented training data, and the validation data to prepare for training.

```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))  
preprocessedValidationData = transform(validationData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image and bounding boxes.

```
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I,'Rectangle',bbox);  
annotatedImage = imresize(annotatedImage,2);  
figure  
imshow(annotatedImage)
```



Train YOLO v2 Object Detector

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',16, ...
    'InitialLearnRate',1e-3, ...
    'MaxEpochs',20, ...
    'CheckpointPath',tempdir, ...
    'ValidationData',preprocessedValidationData);
```

Use `trainYOLOv2ObjectDetector` function to train YOLO v2 object detector if `doTraining` is true. Otherwise, load the pretrained network.

```
if doTraining
    % Train the YOLO v2 detector.
    [detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,options);
else
    % Load pretrained detector for the example.
    pretrained = load('yoloV2ResNet50VehicleExample_19b.mat');
    detector = pretrained.detector;
end
```

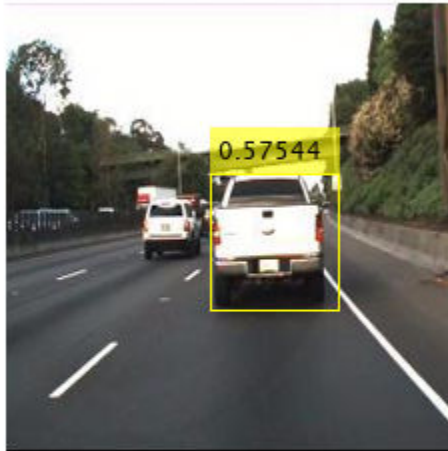
This example was verified on an NVIDIA™ Titan X GPU with 12 GB of memory. If your GPU has less memory, you may run out of memory. If this happens, lower the `'MiniBatchSize'` using the `trainingOptions` function. Training this network took approximately 7 minutes using this setup. Training time varies depending on the hardware you use.

As a quick test, run the detector on a test image. Make sure you resize the image to the same size as the training images.

```
I = imread('highway.png');
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```



Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Test data should be representative of the original data and be left unmodified for unbiased evaluation.

```
preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

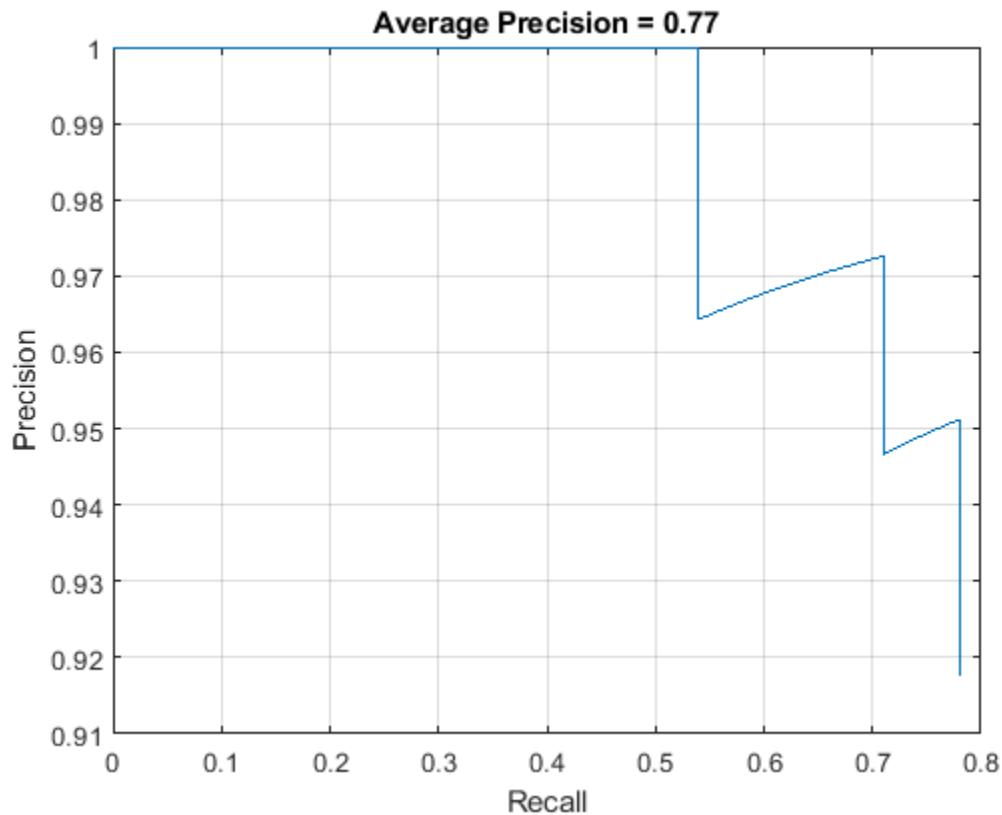
```
detectionResults = detect(detector, preprocessedTestData);
```

Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults, preprocessedTestData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure  
plot(recall,precision)  
xlabel('Recall')  
ylabel('Precision')  
grid on  
title(sprintf('Average Precision = %.2f',ap))
```



Code Generation

Once the detector is trained and evaluated, you can generate code for the `yoloV2ObjectDetector` using GPU Coder™. See “Code Generation for Object Detection by Using YOLO v2” (GPU Coder) example for more details.

Supporting Functions

```
function B = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.
B = cell(size(A));

I = A{1};
sz = size(I);
if numel(sz)==3 && sz(3) == 3
    I = jitterColorHSV(I,...
        'Contrast',0.2,...
        'Hue',0,...
        'Saturation',0.1,...
        'Brightness',0.2);
end

% Randomly flip and scale image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');
```

```
B{1} = imwarp(I,tform,'OutputView',rout);

% Apply same transform to boxes.
[B{2},indices] = bboxwarp(A{2},tform,rout,'OverlapThreshold',0.25);
B{3} = A{3}(indices);

% Return original data only when all boxes are removed by warping.
if isempty(indices)
    B = A;
end
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to the targetSize.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

Import Pretrained ONNX YOLO v2 Object Detector

This example shows how to import a pretrained ONNX™ (Open Neural Network Exchange) you only look once (YOLO) v2 [1] on page 3-0 object detection network and use it to detect objects. After you import the network, you can deploy it to embedded platforms using GPU Coder™ or retrain it on custom data using transfer learning with `trainYOLOv2ObjectDetector`.

Download ONNX YOLO v2 Network

Download files related to the pretrained Tiny YOLO v2 network [2] on page 3-0 , [3] on page 3-0 .

```
pretrainedURL = 'https://onnxzoo.blob.core.windows.net/models/opset_8/tiny_yolov2/tiny_yolov2.ta
pretrainedNetZip = 'yolov2Tmp.tar.gz';
if ~exist(pretrainedNetZip, 'file')
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetZip, pretrainedURL);
end
```

```
Downloading pretrained network (58 MB)...
```

Extract YOLO v2 Network

Unzip and untar the downloaded file to extract the Tiny YOLO v2 network. Load the 'model.onnx' model, which is an ONNX YOLO v2 network pretrained on the PASCAL VOC data set. The network can detect objects from 20 different classes [4] on page 3-0 .

```
pretrainedNetTar = gunzip(pretrainedNetZip);
onnxfiles = untar(pretrainedNetTar{1});
pretrainedNet = onnxfiles{1,2};
```

Import ONNX YOLO v2 Layers

Use the `importONNXLayers` function to import the downloaded network.

```
lgraph = importONNXLayers(pretrainedNet, 'ImportWeights', true);
```

```
Warning: Imported layers have no output layer because ONNX files do not specify the network's out
```

In this example you add an output layer to the imported layers, so you can ignore this warning. The `Add YOLO v2 Transform and Output layers` on page 3-0 section shows how to add YOLO v2 output layer along with YOLO v2 Transform layer to the imported layers.

The network in this example contains no unsupported layers. Note that if the network you want to import has unsupported layers, the function imports them as placeholder layers. Before you can use your imported network, you must replace these layers. For more information on replacing placeholder layers, see `findPlaceholderLayers` (Deep Learning Toolbox).

Define YOLO v2 Anchor Boxes

YOLO v2 uses predefined anchor boxes to predict object location. The anchor boxes used in the imported network are defined in the Tiny YOLO v2 network configuration file [5] on page 3-0 . The ONNX anchors are defined with respect to the output size of the final convolution layer, which is 13-by-13. To use the anchors with `yolov2ObjectDetector`, resize the anchor boxes to the network input size, which is 416-by-416. The anchor boxes for `yolov2ObjectDetector` must be specified in the form [height, width].

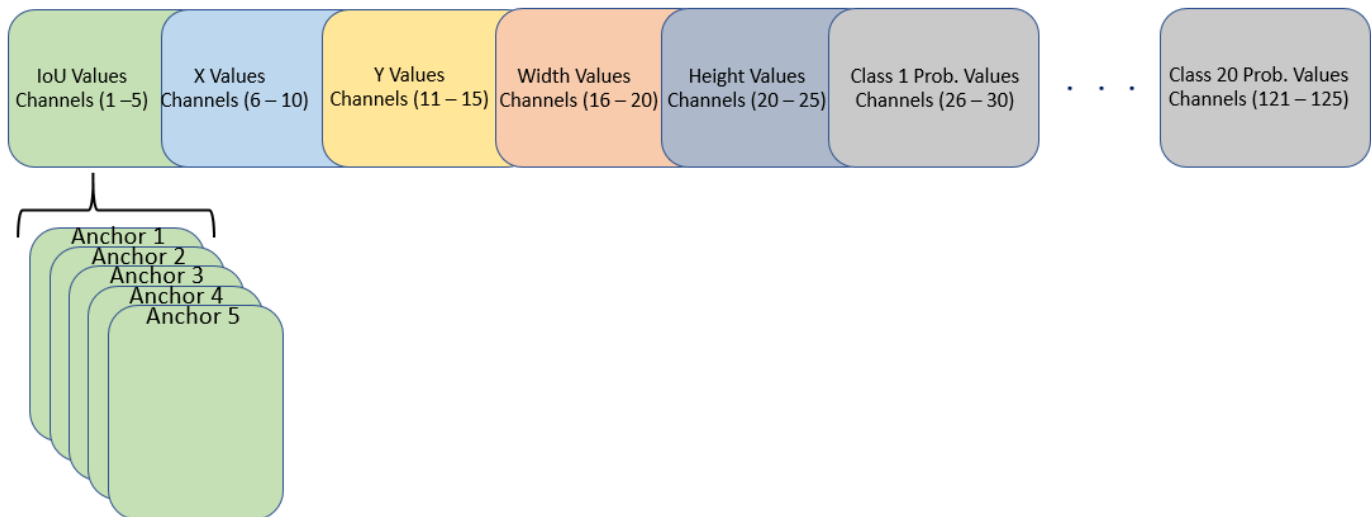
```
onnxAnchors = [1.08,1.19; 3.42,4.41; 6.63,11.38; 9.42,5.11; 16.62,10.52];

inputSize = lgraph.Layers(1,1).InputSize(1:2);
lastActivationSize = [13,13];
upScaleFactor = inputSize./lastActivationSize;
anchorBoxesTmp = round(upScaleFactor.* onnxAnchors);
anchorBoxes = [anchorBoxesTmp(:,2),anchorBoxesTmp(:,1)];
```

Reorder Detection Layer Weights

For efficient processing, you must reorder the weights and biases of the last convolution layer in the imported network to obtain the activations in the arrangement that `yoloV2ObjectDetector` requires. `yoloV2ObjectDetector` expects the 125 channels of the feature map of the last convolution layer in the following arrangement:

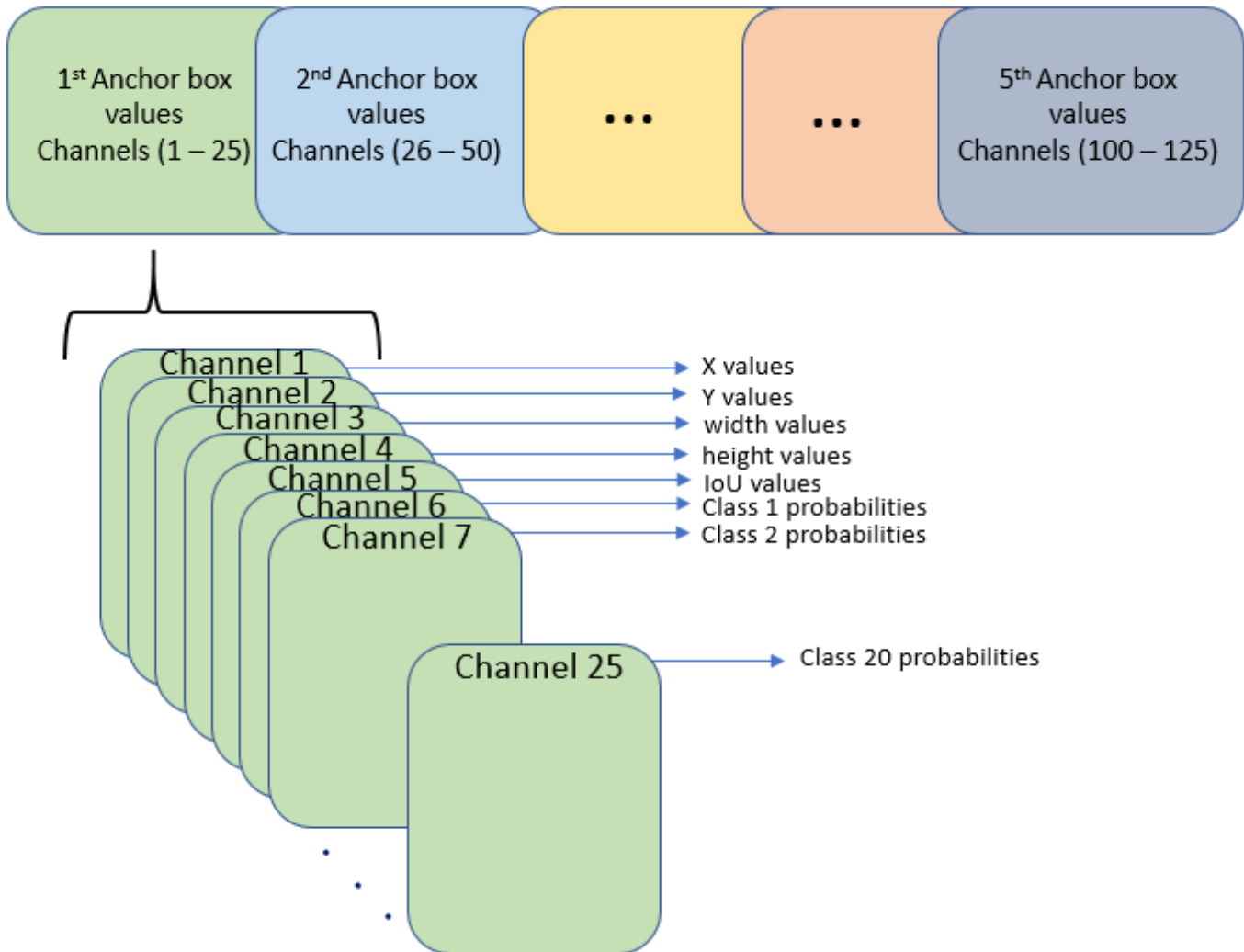
- Channels 1 to 5 - IoU values for five anchors
- Channels 6 to 10 - X values for five anchors
- Channels 11 to 15 - Y values for five anchors
- Channels 16 to 20 - Width values for five anchors
- Channels 21 to 25 - Height values for five anchors
- Channels 26 to 30 - Class 1 probability values for five anchors
- Channels 31 to 35 - Class 2 probability values for five anchors
- Channels 121 to 125 - Class 20 probability values for five anchors



However, in the last convolution layer, which is of size 13-by-13, the activations are arranged differently. Each of the 25 channels in the feature map corresponds to:

- Channel 1 - X values
- Channel 2 - Y values
- Channel 3 - Width values
- Channel 4 - Height values
- Channel 5 - IoU values

- Channel 6 - Class 1 probability values
- Channel 7 - Class 2 probability values
- Channel 25 - Class 20 probability values



Use the supporting function `rearrangeONNXWeights`, listed at the end of this example, to reorder the weights and biases of the last convolution layer in the imported network and obtain the activations in the format required by `yoloV2ObjectDetector`.

```
weights = lgraph.Layers(end,1).Weights;
bias = lgraph.Layers(end,1).Bias;
layerName = lgraph.Layers(end,1).Name;

numAnchorBoxes = size(onnxAnchors,1);
[modWeights,modBias] = rearrangeONNXWeights(weights,bias,numAnchorBoxes);
```

Replace the weights and biases of the last convolution layer in the imported network with the new convolution layer using the reordered weights and biases.

```
filterSize = size(modWeights,[1 2]);
numFilters = size(modWeights,4);
modConvolution8 = convolution2dLayer(filterSize,numFilters,...
    'Name',layerName,'Bias',modBias,'Weights',modWeights);
lgraph = replaceLayer(lgraph,'convolution8',modConvolution8);
```

Add YOLO v2 Transform and Output Layers

A YOLO v2 detection network requires the YOLO v2 transform and YOLO v2 output layers. Create both of these layers, stack them in series, and attach the YOLO v2 transform layer to the last convolution layer.

```
classNames = tinyYOLOv2Classes;

layersToAdd = [
    yolov2TransformLayer(numAnchorBoxes,'Name','yolov2Transform');
    yolov2OutputLayer(anchorBoxes,'Classes',classNames,'Name','yolov2Output');
];

lgraph = addLayers(lgraph, layersToAdd);
lgraph = connectLayers(lgraph,layerName,'yolov2Transform');
```

The `ElementwiseAffineLayer` in the imported network duplicates the preprocessing step performed by `yolov2ObjectDetector`. Hence, remove the `ElementwiseAffineLayer` from the imported network.

```
yoloScaleLayerIdx = find(...
    arrayfun( @(x)isa(x,'nnet.onnx.layer.ElementwiseAffineLayer'), ...
    lgraph.Layers));

if ~isempty(yoloScaleLayerIdx)
    for i = 1:size(yoloScaleLayerIdx,1)
        layerNames {i} = lgraph.Layers(yoloScaleLayerIdx(i,1),1).Name;
    end
    lgraph = removeLayers(lgraph,layerNames);
    lgraph = connectLayers(lgraph,'Input_image','convolution');
end
```

Create YOLO v2 Object Detector

Assemble the layer graph using the `assembleNetwork` function and create a YOLO v2 object detector using the `yolov2ObjectDetector` function.

```
net = assembleNetwork(lgraph)

net =
    DAGNetwork with properties:

        Layers: [34x1 nnet.cnn.layer.Layer]
    Connections: [33x2 table]
    InputNames: {'Input_image'}
    OutputNames: {'yolov2Output'}

yolov2Detector = yolov2ObjectDetector(net)

yolov2Detector =
    yolov2ObjectDetector with properties:
```

```

    ModelName: 'importedNetwork'
      Network: [1x1 DAGNetwork]
TrainingImageSize: [416 416]
  AnchorBoxes: [5x2 double]
  ClassNames: [aeroplane  bicycle  bird  boat  bottle  bus  car  cat  cl

```

Detect Objects Using Imported YOLO v2 Detector

Use the imported detector to detect objects in a test image. Display the results.

```

I = imread('car1.jpg');
% Convert image to BGR format.
I = cat(3,I(:,:,3),I(:,:,2),I(:,:,1));
[bboxes, scores, labels] = detect(yolov2Detector, I);
detectedImg = insertObjectAnnotation(I, 'rectangle', bboxes, scores);
figure
imshow(detectedImg);

```



Supporting Functions

```

function [modWeights,modBias] = rearrangeONNXWeights(weights,bias,numAnchorBoxes)
%rearrangeONNXWeights rearranges the weights and biases of an imported YOLO
%v2 network as required by yolov2ObjectDetector. numAnchorBoxes is a scalar
%value containing the number of anchors that are used to reorder the weights and
%biases. This function performs the following operations:
% * Extract the weights and biases related to IoU, boxes, and classes.

```

```
% * Reorder the extracted weights and biases as expected by yolov2objectDetector.
% * Combine and reshape them back to the original dimensions.

weightsSize = size(weights);
biasSize = size(bias);
sizeofPredictions = biasSize(3)/numAnchorBoxes;

% Reshape the weights with regard to the size of the predictions and anchors.
reshapedWeights = reshape(weights,prod(weightsSize(1:3)),sizeofPredictions,numAnchorBoxes);

% Extract the weights related to IoU, boxes, and classes.
weightsIou = reshapedWeights(:,5,:);
weightsBoxes = reshapedWeights(:,1:4,:);
weightsClasses = reshapedWeights(:,6:end,:);

% Combine the weights of the extracted parameters as required by
% yolov2objectDetector.
reorderedWeights = cat(2,weightsIou,weightsBoxes,weightsClasses);
permutedWeights = permute(reorderedWeights,[1 3 2]);

% Reshape the new weights to the original size.
modWeights = reshape(permutedWeights,weightsSize);

% Reshape the biases with regard to the size of the predictions and anchors.
reshapedBias = reshape(bias,sizeofPredictions,numAnchorBoxes);

% Extract the biases related to IoU, boxes, and classes.
biasIou = reshapedBias(5,:);
biasBoxes = reshapedBias(1:4,:);
biasClasses = reshapedBias(6:end,:);

% Combine the biases of the extracted parameters as required by yolov2objectDetector.
reorderedBias = cat(1,biasIou,biasBoxes,biasClasses);
permutedBias = permute(reorderedBias,[2 1]);

% Reshape the new biases to the original size.
modBias = reshape(permutedBias,biasSize);
end

function classes = tinyYOLOv2Classes()
% Return the class names corresponding to the pretrained ONNX tiny YOLO v2
% network.
%
% The tiny YOLO v2 network is pretrained on the Pascal VOC data set,
% which contains images from 20 different classes [4].

classes = [ ...
    " aeroplane", "bicycle", "bird", "boat", "bottle", "bus", "car",...
    "cat", "chair", "cow", "diningtable", "dog", "horse", "motorbike",...
    "person", "pottedplant", "sheep", "sofa", "train", "tvmonitor"];
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 6517–25. Honolulu, HI: IEEE, 2017. <https://doi.org/10.1109/CVPR.2017.690>.

[2] "Tiny YOLO v2 Model." https://github.com/onnx/models/tree/master/vision/object_detection_segmentation/tiny-yolov2

[3] "Tiny YOLO v2 Model License." <https://github.com/onnx/onnx/blob/master/LICENSE>.

[4] Everingham, Mark, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. "The Pascal Visual Object Classes (VOC) Challenge." *International Journal of Computer Vision* 88, no. 2 (June 2010): 303–38. <https://doi.org/10.1007/s11263-009-0275-4>.

[5] "yolov2-tiny-voc.cfg" <https://github.com/pjreddie/darknet/blob/master/cfg/yolov2-tiny-voc.cfg>.

Export YOLO v2 Object Detector to ONNX

This example shows how to export a YOLO v2 object detection network to ONNX™ (Open Neural Network Exchange) model format. After exporting the YOLO v2 network, you can import the network into other deep learning frameworks for inference. This example also presents the workflow that you can follow to perform inference using the imported ONNX model.

Export YOLO v2 Network

Export the detection network to ONNX and gather the metadata required to generate object detection results.

First, load a pretrained YOLO v2 object detector into the workspace.

```
input = load('yolov2VehicleDetector.mat');  
net = input.detector.Network;
```

Next, obtain the YOLO v2 detector metadata to use for inference. The detector metadata includes the network input image size, anchor boxes, and activation size of last convolution layer.

Read the network input image size from the input YOLO v2 network.

```
inputImageSize = net.Layers(1,1).InputSize;
```

Read the anchor boxes used for training from the input detector.

```
anchorBoxes = input.detector.AnchorBoxes;
```

Get the activation size of the last convolution layer in the input network by using the `analyzeNetwork` function.

```
analyzeNetwork(net);
```


Deep Learning Network Analyzer

net

Analysis date: 08-Dec-2019 18:52:06

25 layers

0 warnings

0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnables
	Batch normalization with ...			Scale 1x1x128
16	relu_4 ReLU	ReLU	16x16x128	-
17	yolov2Conv1 128 3x3x128 convolution...	Convolution	16x16x128	Weights 3x3x128x128 Bias 1x1x128
18	yolov2Batch1 Batch normalization with ...	Batch Normalization	16x16x128	Offset 1x1x128 Scale 1x1x128
19	yolov2Relu1 ReLU	ReLU	16x16x128	-
20	yolov2Conv2 128 3x3x128 convolution...	Convolution	16x16x128	Weights 3x3x128x128 Bias 1x1x128
21	yolov2Batch2 Batch normalization with ...	Batch Normalization	16x16x128	Offset 1x1x128 Scale 1x1x128
22	yolov2Relu2 ReLU	ReLU	16x16x128	-
23	yolov2ClassConv 24 1x1x128 convolutions ...	Convolution	16x16x24	Weights 1x1x128x24 Bias 1x1x24
24	yolov2Transform YOLO v2 Transform Laye...	YOLO v2 Transform...	16x16x24	-
25	yolov2OutputLayer YOLO v2 Output with 4 a...	YOLO v2 Output	-	-

```
finalActivationSize = [16 16 24];
```

Export to ONNX Model Format

Export the YOLO v2 object detection network as an ONNX format file by using the `exportONNXNetwork` (Deep Learning Toolbox) function. Specify the file name as `yolov2.onnx`. The function saves the exported ONNX file to the current working folder.

```
filename = 'yolov2.onnx';
exportONNXNetwork(net, filename);
```

The `exportONNXNetwork` function maps the `yolov2TransformLayer` and `yolov2OutputLayer` in the input YOLO v2 network to the basic ONNX operator and identity operator, respectively. After you export the network, you can import the `yolov2.onnx` file into any deep learning framework that supports ONNX import.

Using the `exportONNXNetwork`, requires Deep Learning Toolbox™ and the Deep Learning Toolbox Converter for ONNX Model Format support package. If this support package is not installed, then the function provides a download link.

Object Detection Using Exported YOLO v2 Network

When exporting is complete, you can import the ONNX model into any deep learning framework and use the following workflow to perform object detection. Along with the ONNX network, this workflow

also requires the YOLO v2 detector metadata `inputImageSize`, `anchorBoxes`, and `finalActivationSize` obtained from the MATLAB workspace. The following code is a MATLAB implementation of the workflow that you must translate into the equivalent code for the framework of your choice.

Preprocess Input Image

Preprocess the image to use for inference. The image must be an RGB image and must be resized to the network input image size, and its pixel values must lie in the interval [0 1].

```
I = imread('highway.png');  
resizedI = imresize(I,inputImageSize(1:2));  
rescaledI = rescale(resizedI);
```

Pass Input and Run ONNX Model

Run the ONNX model in the deep learning framework of your choice with the preprocessed image as input to the imported ONNX model.

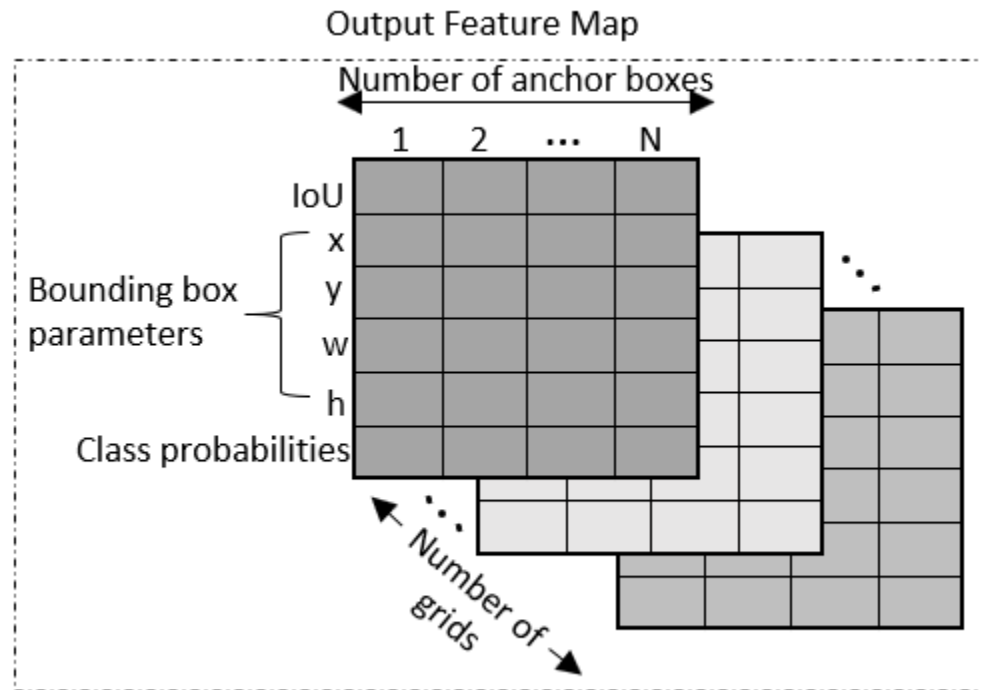
Extract Predictions from Output of ONNX Model

The model predicts the following:

- Intersection over union (IoU) with ground truth boxes
- x , y , w , and h bounding box parameters for each anchor box
- Class probabilities for each anchor box

The output of the ONNX model is a feature map that contains the predictions and is of size `predictionsPerAnchor-by-numAnchors-by-numGrids`.

- `numAnchors` is the number of anchor boxes.
- `numGrids` is the number of grids calculated as the product of the height and width of the last convolution layer.
- `predictionsPerAnchor` is the output predictions in the form `[IoU;x;y;w;h;class probabilities]`.



- The first row in the feature map contains IoU predictions for each anchor box.
- The second and third rows in the feature map contain predictions for the centroid coordinates (x,y) of each anchor box.
- The fourth and fifth rows in the feature map contain the predictions for the width and height of each anchor box.
- The sixth row in the feature map contains the predictions for class probabilities of each anchor box.

Compute Final Detections

To compute final detections for the preprocessed test image, you must:

- Rescale the bounding box parameters with respect to the size of the input layer of the network.
- Compute object confidence scores from the predictions.
- Obtain predictions with high object confidence scores.
- Perform nonmaximum suppression.

As an implementation guide, use the code for `yolov2PostProcess` on page 3-0 function in Postprocessing Functions on page 3-0 .

```
[bboxes,scores,labels] = yolov2PostProcess(featureMap,inputImageSize,finalActivationsSize,anchors)
```

Display Detection Results

```
Idisp = insertObjectAnnotation(resizedI, 'rectangle', bboxes, scores);
figure
imshow(Idisp)
```

References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6517-25. Honolulu, HI: IEEE, 2017. <https://doi.org/10.1109/CVPR.2017.690>.

Postprocessing Functions

```
function [bboxes,scores,labels] = yolov2PostProcess(featureMap,inputImageSize,finalActivationsS

% Extract prediction values from the feature map.
iouPred = featureMap(1,:,:);
xyPred = featureMap(2:3,:,:);
whPred = featureMap(4:5,:,:);
probPred = featureMap(6,:,:);

% Rescale the bounding box parameters.
bBoxes = rescaleBbox(xyPred,whPred,anchorBoxes,finalActivationsSize,inputImageSize);

% Rearrange the feature map as a two-dimensional matrix for efficient processing.
predVal = [bBoxes;iouPred;probPred];
predVal = reshape(predVal,size(predVal,1),[]);

% Compute object confidence scores from the rearranged prediction values.
[confScore,idx] = computeObjectScore(predVal);

% Obtain predictions with high object confidence scores.
[bboxPred,scorePred,classPred] = selectMaximumPredictions(confScore,idx,predVal);

% To get the final detections, perform nonmaximum suppression with an overlap threshold of 0.5.
[bboxes,scores,labels] = selectStrongestBboxMulticlass(bboxPred, scorePred, classPred, 'RatioType

end

function bBoxes = rescaleBbox(xyPred,whPred,anchorBoxes,finalActivationsSize,inputImageSize)

% To rescale the bounding box parameters, compute the scaling factor by using the network parameter
scaleY = inputImageSize(1)/finalActivationsSize(1);
scaleX = inputImageSize(2)/finalActivationsSize(2);
scaleFactor = [scaleY scaleX];

bBoxes = zeros(size(xyPred,1)+size(whPred,1),size(anchors,1),size(xyPred,3),'like',xyPred);
for rowIdx=0:finalActivationsSize(1,1)-1
    for colIdx=0:finalActivationsSize(1,2)-1
        ind = rowIdx*finalActivationsSize(1,2)+colIdx+1;
        for anchorIdx = 1 : size(anchorBoxes,1)

            % Compute the center with respect to image.
            cx = (xyPred(1,anchorIdx,ind)+colIdx)* scaleFactor(1,2);
            cy = (xyPred(2,anchorIdx,ind)+rowIdx)* scaleFactor(1,1);

            % Compute the width and height with respect to the image.
            bw = whPred(1,anchorIdx,ind)* anchorBoxes(anchorIdx,1);
            bh = whPred(2,anchorIdx,ind)* anchorBoxes(anchorIdx,2);

            bBoxes(1,anchorIdx,ind) = (cx-bw/2);
            bBoxes(2,anchorIdx,ind) = (cy-bh/2);
            bBoxes(3,anchorIdx,ind) = w;
```

```
        bBoxes(4,anchorIdx,ind) = h;
    end
end
end

function [confScore,idx] = computeObjectScore(predVal)
iouPred = predVal(5,:);
probPred = predVal(6:end,:);
[imax,idx] = max(probPred,[],1);
confScore = iouPred.*imax;
end

function [bboxPred,scorePred,classPred] = selectMaximumPredictions(confScore,idx,predVal)
% Specify the threshold for confidence scores.
confScoreId = confScore >= 0.5;
% Obtain the confidence scores greater than or equal to 0.5.
scorePred = confScore(:,confScoreId);
% Obtain the class IDs for predictions with confidence scores greater than
% or equal to 0.5.
classPred = idx(:,confScoreId);
% Obtain the bounding box parameters for predictions with confidence scores
% greater than or equal to 0.5.
bboxesXYWH = predVal(1:4,:);
bboxPred = bboxesXYWH(:,confScoreId);
end
```

Estimate Anchor Boxes From Training Data

Anchor boxes are important parameters of deep learning object detectors such as Faster R-CNN and YOLO v2. The shape, scale, and number of anchor boxes impact the efficiency and accuracy of the detectors.

For more information, see “Anchor Boxes for Object Detection” on page 14-21.

Load Training Data

Load the vehicle dataset, which contains 295 images and associated box labels.

```
data = load('vehicleTrainingData.mat');  
vehicleDataset = data.vehicleTrainingData;
```

Add the full path to the local vehicle data folder.

```
dataDir = fullfile(toolboxdir('vision'),'visiondata');  
vehicleDataset.imageFilename = fullfile(dataDir,vehicleDataset.imageFilename);
```

Display the data set summary.

```
summary(vehicleDataset)
```

Variables:

```
imageFilename: 295x1 cell array of character vectors  
vehicle: 295x1 cell
```

Visualize Ground Truth Box Distribution

Visualize the labeled boxes to better understand the range of object sizes present in the data set.

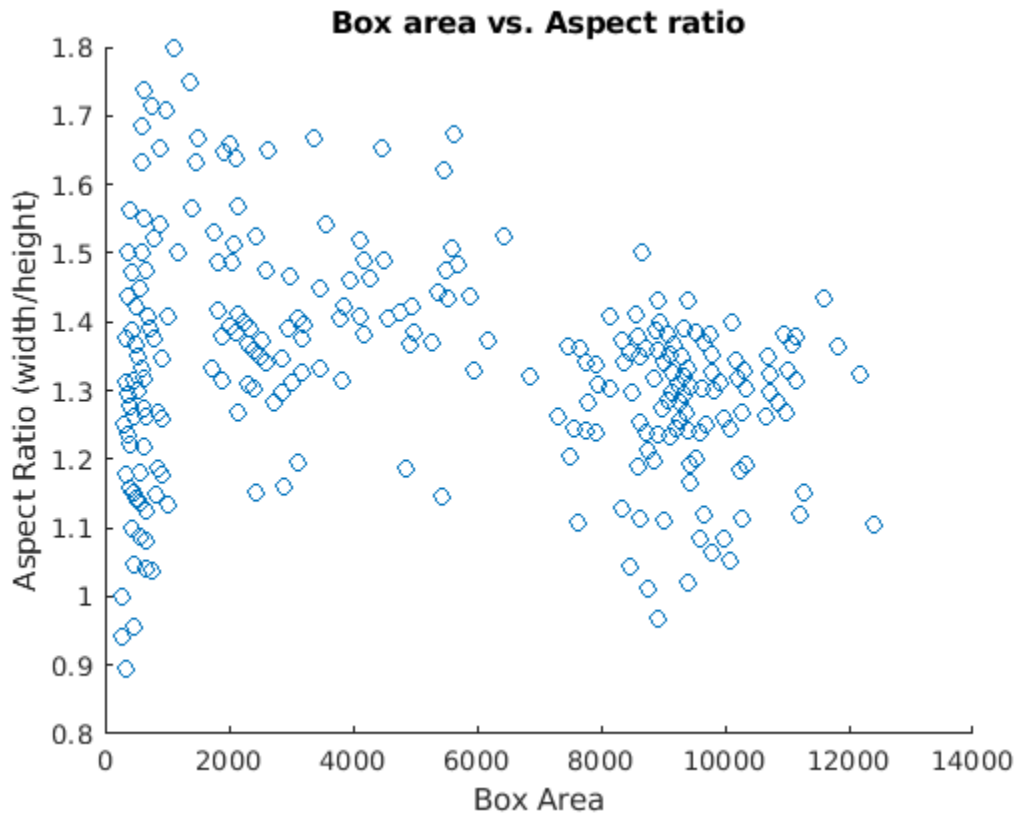
Combine all the ground truth boxes into one array.

```
allBoxes = vertcat(vehicleDataset.vehicle{:});
```

Plot the box area versus the box aspect ratio.

```
aspectRatio = allBoxes(:,3) ./ allBoxes(:,4);  
area = prod(allBoxes(:,3:4),2);
```

```
figure  
scatter(area,aspectRatio)  
xlabel("Box Area")  
ylabel("Aspect Ratio (width/height)");  
title("Box Area vs. Aspect Ratio")
```



The plot shows a few groups of objects that are of similar size and shape. However, because the groups are spread out, manually choosing anchor boxes is difficult. A better way to estimate anchor boxes is to use a clustering algorithm that can group similar boxes together using a meaningful metric.

Estimate Anchor Boxes

Estimate anchor boxes from training data using the `estimateAnchorBoxes` function, which uses the intersection-over-union (IoU) distance metric.


A distance metric based on IoU is invariant to the size of boxes, unlike the Euclidean distance metric, which produces larger errors as the box sizes increase [1]. In addition, using an IoU distance metric leads to boxes of similar aspect ratios and sizes being clustered together, which results in anchor box estimates that fit the data.

Create a `boxLabelDatastore` using the ground truth boxes in the vehicle data set. If the preprocessing step for training an object detector involves resizing of the images, use `transform` and `bboxresize` to resize the bounding boxes in the `boxLabelDatastore` before estimating the anchor boxes.

```
trainingData = boxLabelDatastore(vehicleDataset(:,2:end));
```

Select the number of anchors and estimate the anchor boxes using `estimateAnchorBoxes` function.

```

numAnchors = 5  ;
[anchorBoxes,meanIoU] = estimateAnchorBoxes(trainingData,numAnchors);
anchorBoxes

anchorBoxes = 5x2

    21    27
    87   116
    67    92
    43    61
    86   105

```

Choosing the number of anchors is another training hyperparameter that requires careful selection using empirical analysis. One quality measure for judging the estimated anchor boxes is the mean IoU of the boxes in each cluster. The `estimateAnchorBoxes` function uses a k -means clustering algorithm with the IoU distance metric to calculate the overlap using the equation, $1 - \text{bboxOverlapRatio}(\text{allBoxes}, \text{boxInCluster})$.

```
meanIoU
```

```
meanIoU = 0.8411
```

The mean IoU value greater than 0.5 ensures that the anchor boxes overlap well with the boxes in the training data. Increasing the number of anchors can improve the mean IoU measure. However, using more anchor boxes in an object detector can also increase the computation cost and lead to overfitting, which results in poor detector performance.

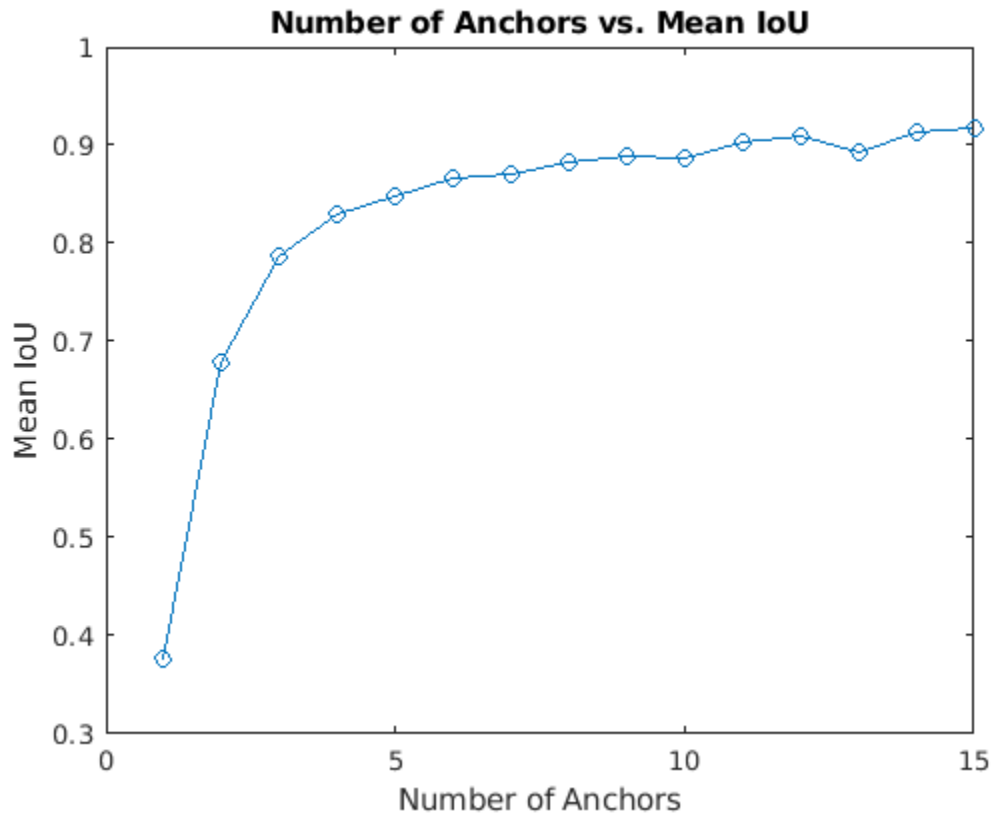
Sweep over a range of values and plot the mean IoU versus number of anchor boxes to measure the trade-off between number of anchors and mean IoU.

```

maxNumAnchors = 15;
meanIoU = zeros([maxNumAnchors,1]);
anchorBoxes = cell(maxNumAnchors, 1);
for k = 1:maxNumAnchors
    % Estimate anchors and mean IoU.
    [anchorBoxes{k},meanIoU(k)] = estimateAnchorBoxes(trainingData,k);
end

figure
plot(1:maxNumAnchors,meanIoU,'-o')
ylabel("Mean IoU")
xlabel("Number of Anchors")
title("Number of Anchors vs. Mean IoU")

```

Using two anchor boxes results in a mean IoU value greater than 0.65, and using more than 7 anchor boxes yields only marginal improvement in mean IoU value. Given these results, the next step is to train and evaluate multiple object detectors using values between 2 and 6. This empirical analysis helps determine the number of anchor boxes required to satisfy application performance requirements, such as detection speed, or accuracy.

References

Redmon, Joseph, and Ali Farhadi. "YOLO9000: better, faster, stronger." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 7263-7271. 2017.

- [1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6517-25. Honolulu, HI: IEEE, 2017. <https://doi.org/10.1109/CVPR.2017.690>.

Object Detection Using YOLO v3 Deep Learning

This example shows how to train a YOLO v3 on page 3-0 object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several techniques for object detection exist, including Faster R-CNN, you only look once (YOLO) v2, and single shot detector (SSD). This example shows how to train a YOLO v3 object detector. YOLO v3 improves upon YOLO v2 by adding detection at multiple scales to help detect smaller objects. Moreover, the loss function used for training is separated into mean squared error for bounding box regression and binary cross-entropy for object classification to help improve detection accuracy.

Download Pretrained Network

Download a pretrained network using the helper function `downloadPretrainedYOLOv3Detector` to avoid having to wait for training to complete. If you want to train the network, set the `doTraining` variable to `true`.

```
doTraining = false;

if ~doTraining
    net = downloadPretrainedYOLOv3Detector();
end
```

Load Data

This example uses a small labeled data set that contains 295 images. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the YOLO v3 training procedure, but in practice, more labeled images are needed to train a robust network.

Unzip the vehicle images and load the vehicle ground truth data.

```
unzip('vehicleDatasetImages.zip');
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;

% Add the full path to the local vehicle data folder.
vehicleDataset.imageFilename = fullfile(pwd, vehicleDataset.imageFilename);
```

Note: In case of multiple classes, the data can also be organized as three columns where the first column contains the image file names with paths, the second column contains the bounding boxes and the third column must be a cell vector that contains the label names corresponding to each bounding box. For more information on how to arrange the bounding boxes and labels, see `boxLabelDatastore`.

All the bounding boxes must be in the form `[x y width height]`. This vector specifies the upper left corner and the size of the bounding box in pixels.

Split the data set into a training set for training the network, and a test set for evaluating the network. Use 90% of the data for training set and the rest for the test set.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices));
trainingDataTbl = vehicleDataset(shuffledIndices(1:idx), :);
testDataTbl = vehicleDataset(shuffledIndices(idx+1:end), :);
```

Create an image datastore for loading the images.

```
imdsTrain = imageDatastore(trainingDataTbl.imageFilename);
imdsTest = imageDatastore(testDataTbl.imageFilename);
```

Create a datastore for the ground truth bounding boxes.

```
bldsTrain = boxLabelDatastore(trainingDataTbl(:, 2:end));
bldsTest = boxLabelDatastore(testDataTbl(:, 2:end));
```

Combine the image and box label datastores.

```
trainingData = combine(imdsTrain, bldsTrain);
testData = combine(imdsTest, bldsTest);
```

Use `validateInputData` to detect invalid images, bounding boxes or labels i.e.,

- Samples with invalid image format or containing NaNs
- Bounding boxes containing zeros/NaNs/Infs/empty
- Missing/non-categorical labels.

The values of the bounding boxes should be finite, positive, non-fractional, non-NaN and should be within the image boundary with a positive height and width. Any invalid samples must either be discarded or fixed for proper training.

```
validateInputData(trainingData);
validateInputData(testData);
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

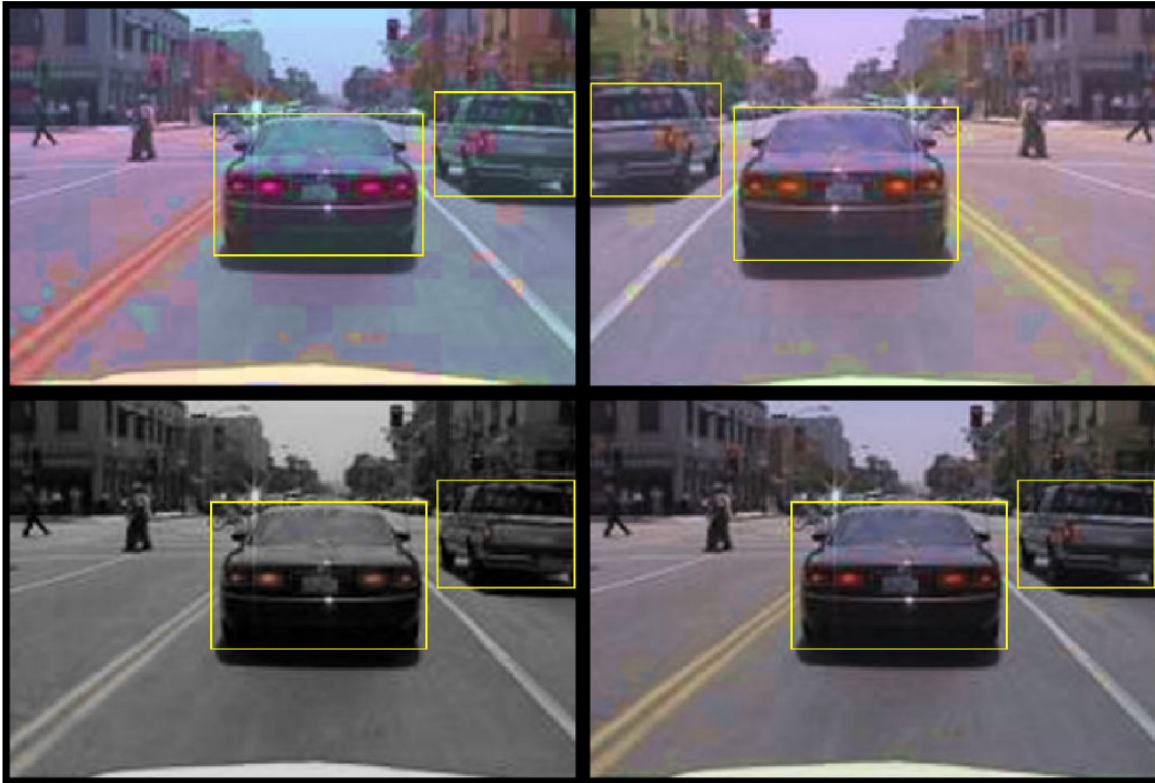
Use `transform` function to apply custom data augmentations to the training data. The `augmentData` helper function, listed at the end of the example, applies the following augmentations to the input data.

- Color jitter augmentation in HSV space
- Random horizontal flip
- Random scaling by 10 percent

```
augmentedTrainingData = transform(trainingData, @augmentData);
```

Read the same image four times and display the augmented training data.

```
% Visualize the augmented images.
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1,1}, 'Rectangle', data{1,2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData, 'BorderSize', 10)
```



Preprocess Training Data

Specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [227 227 3].

```
networkInputSize = [227 227 3];
```

Preprocess the augmented training data to prepare for training. The `preprocessData` helper function, listed at the end of the example, applies the following preprocessing operations to the input data.

- Resize the images to the network input size
- Scale the image pixels in the range [0 1].

```
preprocessedTrainingData = transform(augmentedTrainingData, @(data)preprocessData(data, networkInputSize));
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image with the bounding boxes.

```
I = data{1,1};
bbox = data{1,2};
annotatedImage = insertShape(I, 'Rectangle', bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Reset the datastore.

```
reset(preprocessedTrainingData);
```

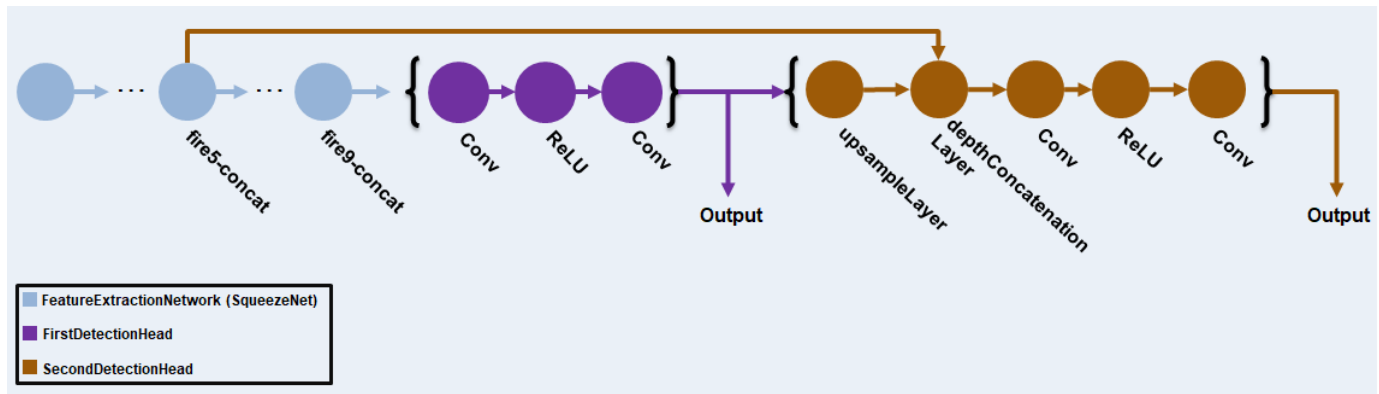
Define YOLO v3 Network

The YOLO v3 network in this example is based on SqueezeNet, and uses the feature extraction network in SqueezeNet with the addition of two detection heads at the end. The second detection head is twice the size of the first detection head, so it is better able to detect small objects. Note that you can specify any number of detection heads of different sizes based on the size of the objects that you want to detect. The YOLO v3 network uses anchor boxes estimated using training data to have better initial priors corresponding to the type of data set and to help the network learn to predict the

boxes accurately. For information about anchor boxes, see “Anchor Boxes for Object Detection” on page 14-21.

The YOLO v3 network in this example is illustrated in the following diagram.

You can use Deep Network Designer (Deep Learning Toolbox) to create the network shown in the diagram.



First, use `transform` to preprocess the training data for computing the anchor boxes, as the training images used in this example are bigger than 227-by-227 and vary in size. Specify the number of anchors as 6 to achieve a good tradeoff between number of anchors and mean IoU. Use the `estimateAnchorBoxes` function to estimate the anchor boxes. For details on estimating anchor boxes, see “Estimate Anchor Boxes From Training Data” on page 3-146. In case of using a pretrained YOLOv3 object detector, the anchor boxes calculated on that particular training dataset need to be specified. Note that the estimation process is not deterministic. To prevent the estimated anchor boxes from changing while tuning other hyperparameters set the random seed prior to estimation using `rng`.

```
rng(0)
trainingDataForEstimation = transform(trainingData, @(data)preprocessData(data, networkInputSize)
numAnchors = 6;
[anchorBoxes, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

`anchorBoxes = 6x2`

```
    41    34
   163   130
    98    93
   144   125
    33    24
    69    66
```

`meanIoU = 0.8507`

Specify `anchorBoxMasks` to select anchor boxes to use in both the detection heads. `anchorBoxMasks` is a cell array of [Mx1], where M denotes the number of detection heads. Each detection head consists of a [1xN] array of row index of anchors in `anchorBoxes`, where N is the number of anchor boxes to use. Select anchor boxes for each detection head based on size—use larger anchor boxes at lower scale and smaller anchor boxes at higher scale. To do so, sort the anchor boxes with the larger anchor boxes first and assign the first three to the first detection head and the next three to the second detection head.

```

area = anchorBoxes(:, 1).*anchorBoxes(:, 2);
[~, idx] = sort(area, 'descend');
anchorBoxes = anchorBoxes(idx, :);
anchorBoxMasks = {[1,2,3]
                  [4,5,6]
                  };

```

Load the SqueezeNet network pretrained on Imagenet data set. You can also choose to load a different pretrained network such as MobileNet-v2 or ResNet-18. YOLO v3 performs better and trains faster when you use a pretrained network.

Next, create the feature extraction network. Choosing the optimal feature extraction layer requires trial and error, and you can use `analyzeNetwork` to find the names of potential feature extraction layers within a network. For this example, use the `squeezenetFeatureExtractor` helper function, listed at the end of this example, to remove the layers after the feature extraction layer `'fire9-concat'`. The layers after this layer are specific to classification tasks and do not help with object detection.

```

baseNetwork = squeezenet;
lgraph = squeezenetFeatureExtractor(baseNetwork, networkInputSize);

```

Specify the names of the object classes, number of object classes to detect, and number of prediction elements per anchor box. The number of predictions per anchor box is set to 5 plus the number of object classes. "5" denoted the 4 bounding box attributes and 1 object confidence. If you use a pretrained YOLOv3 network, specify the class names in the same order they are specified for training the network.

```

classNames = {'vehicle'};
numClasses = size(classNames, 2);
numPredictorsPerAnchor = 5 + numClasses;

```

Add the detection heads to the feature extraction network. Each detection head predicts the bounding box coordinates (x, y, width, height), object confidence, and class probabilities for the respective anchor box masks. Therefore, for each detection head, the number of output filters in the last convolution layer is the number of anchor box mask times the number of prediction elements per anchor box. Use the supporting functions `addFirstDetectionHead` and `addSecondDetectionHead` to add the detection heads to the feature extraction network.

```

lgraph = addFirstDetectionHead(lgraph, anchorBoxMasks{1}, numPredictorsPerAnchor);
lgraph = addSecondDetectionHead(lgraph, anchorBoxMasks{2}, numPredictorsPerAnchor);

```

Finally, connect the detection heads by connecting the first detection head to the feature extraction layer and the second detection head to the output of the first detection head. In addition, merge the upsampled features in the second detection head with features from the `'fire5-concat'` layer to get more meaningful semantic information in the second detection head.

```

lgraph = connectLayers(lgraph, 'fire9-concat', 'conv1Detection1');
lgraph = connectLayers(lgraph, 'relu1Detection1', 'upsample1Detection2');
lgraph = connectLayers(lgraph, 'fire5-concat', 'depthConcat1Detection2/in2');

```

The detection heads comprise the output layer of the network. To extract output features, specify the names of detection heads using an array of form [Mx1]. M is the number of detection heads. Specify the names of detection heads in the order in which it occurs in the network.

```

networkOutputs = ["conv2Detection1"
                  "conv2Detection2"
                  ];

```

Alternatively, instead of the network created above using SqueezeNet, other pretrained YOLOv3 architectures trained using larger datasets like MS-COCO can be used to transfer learn the detector on custom object detection task. Transfer learning can be realized either by changing the value of number of filters of the last convolution layer or by creating new detection heads as described above, where in the latter case refer to the `squeezenetFeatureExtractor` for extracting the relevant layers. Transfer learning workflow is recommended if the class of the custom object detection is present either as one of the class or subclass of classes trained in the pretrained network.

Specify Training Options

Specify these training options.

- Set the number of epochs to be 70.
- Set the mini batch size as 8. Stable training can be possible with higher learning rates when higher mini batch size is used. Although, this should be set depending on the available memory.
- Set the learning rate to 0.001.
- Set the warmup period as 1000 iterations. This parameter denotes the number of iterations to increase the learning rate exponentially based on the formula $\text{learningRate} \times \left(\frac{\text{iteration}}{\text{warmupPeriod}}\right)^4$. It helps in stabilizing the gradients at higher learning rates.
- Set the L2 regularization factor to 0.0005.
- Specify the penalty threshold as 0.5. Detections that overlap less than 0.5 with the ground truth are penalized.
- Initialize the velocity of gradient as []. This is used by SGDM to store the velocity of gradients.

```
numEpochs = 70;  
miniBatchSize = 8;  
learningRate = 0.001;  
warmupPeriod = 1000;  
l2Regularization = 0.0005;  
penaltyThreshold = 0.5;  
velocity = [];
```

Train Model

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

Use the `minibatchqueue` function to split the preprocessed training data into batches with the supporting function `createBatchData` which returns the batched images and bounding boxes combined with the respective class IDs. For faster extraction of the batch data for training, `dispatchInBackground` should be set to "true" which ensures the usage of parallel pool.

`minibatchqueue` automatically detects the availability of a GPU. If you do not have a GPU, or do not want to use one for training, set the `OutputEnvironment` parameter to "cpu".

```
if canUseParallelPool  
    dispatchInBackground = true;  
else  
    dispatchInBackground = false;  
end  
  
mbqTrain = minibatchqueue(preprocessedTrainingData, 2, ...
```



```

"MiniBatchSize", miniBatchSize,...
"MiniBatchFcn", @(images, boxes, labels) createBatchData(images, boxes, labels, className
"MiniBatchFormat", ["SSCB", ""],...
"DispatchInBackground", dispatchInBackground,...
"OutputCast", ["", "double"]);

```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object. Then create the training progress plotter using supporting function `configureTrainingProgressPlotter`.

Finally, specify the custom training loop. For each iteration:

- Read data from the `minibatchqueue`. If it doesn't have any more data, reset the `minibatchqueue` and shuffle.
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function. The function `modelGradients`, listed as a supporting function, returns the gradients of the loss with respect to the learnable parameters in `net`, the corresponding mini-batch loss, and the state of the current batch.
- Apply a weight decay factor to the gradients to regularization for more robust training.
- Determine the learning rate based on the iterations using the `piecewiseLearningRateWithWarmup` supporting function.
- Update the network parameters using the `sgdupdate` function.
- Update the state parameters of `net` with the moving average.
- Display the learning rate, total loss, and the individual losses (box loss, object loss and class loss) for every iteration. These can be used to interpret how the respective losses are changing in each iteration. For example, a sudden spike in the box loss after few iterations implies that there are Inf or NaNs in the predictions.
- Update the training progress plot.

The training can also be terminated if the loss has saturated for few epochs.

```

if doTraining
    % Convert layer graph to dlnetwork.
    net = dlnetwork(lgraph);

    % Create subplots for the learning rate and mini-batch loss.
    fig = figure;
    [lossPlotter, learningRatePlotter] = configureTrainingProgressPlotter(fig);

    iteration = 0;
    % Custom training loop.
    for epoch = 1:numEpochs

        reset(mbqTrain);
        shuffle(mbqTrain);

        while(hasdata(mbqTrain))
            iteration = iteration + 1;

            [XTrain, YTrain] = next(mbqTrain);

            % Evaluate the model gradients and loss using dlfeval and the
            % modelGradients function.
            [gradients, state, lossInfo] = dlfeval(@modelGradients, net, XTrain, YTrain, anchorB

```

```

% Apply L2 regularization.
gradients = dlupdate(@(g,w) g + l2Regularization*w, gradients, net.Learnables);

% Determine the current learning rate value.
currentLR = piecewiseLearningRateWithWarmup(iteration, epoch, learningRate, warmupPe

% Update the network learnable parameters using the SGDM optimizer.
[net, velocity] = sgdmupdate(net, gradients, velocity, currentLR);

% Update the state parameters of dlnetwork.
net.State = state;

% Display progress.
displayLossInfo(epoch, iteration, currentLR, lossInfo);

% Update training plot with new points.
updatePlots(lossPlotter, learningRatePlotter, iteration, currentLR, lossInfo.totalLo

end
end
end

```

Evaluate Model

Computer Vision System Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). In this example, the average precision metric is used. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Following these steps to evaluate the trained `dlnetwork` object `net` on test data.

- Specify the confidence threshold as 0.5 to keep only detections with confidence scores above this value.
- Specify the overlap threshold as 0.5 to remove overlapping detections.
- Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Test data must be representative of the original data and be left unmodified for unbiased evaluation.
- Collect the detection results by running the detector on `preprocessedTestData`. Use the supporting function `yolov3Detect` to get the bounding boxes, object confidence scores, and class labels.
- Call `evaluateDetectionPrecision` with predicted results and `preprocessedTestData` as arguments.

```

confidenceThreshold = 0.5;
overlapThreshold = 0.5;

% Create the test datastore.
preprocessedTestData = transform(testData, @(data)preprocessData(data, networkInputSize));

% Create a table to hold the bounding boxes, scores, and labels returned by
% the detector.
numImages = size(testDataTbl, 1);
results = table('Size', [0 3], ...
    'VariableTypes', {'cell', 'cell', 'cell'}, ...

```

```

    'VariableNames', {'Boxes', 'Scores', 'Labels'});

mbqTest = minibatchqueue(preprocessedTestData, 1, ...
    "MiniBatchSize", miniBatchSize, ...
    "MiniBatchFormat", "SSCB");

% Run detector on images in the test set and collect results.
while hasdata(mbqTest)
    % Read the datastore and get the image.
    XTest = next(mbqTest);

    % Run the detector.
    [bboxes, scores, labels] = yolov3Detect(net, XTest, networkOutputs, anchorBoxes, anchorBoxM

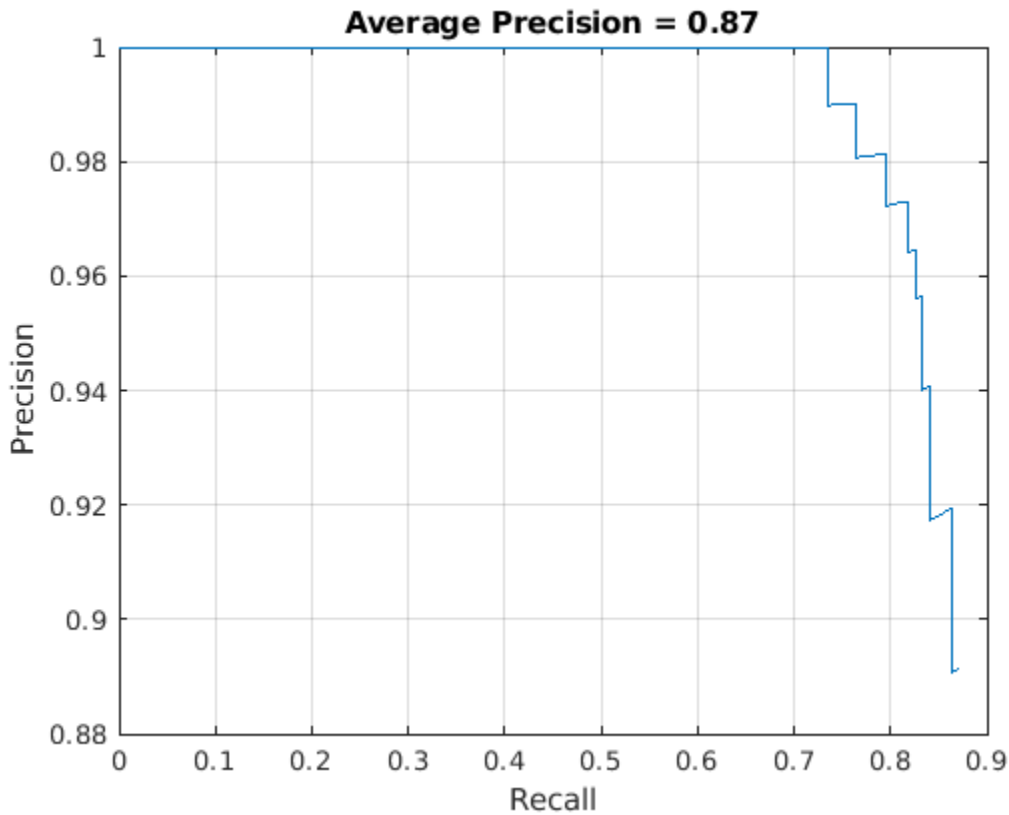
    % Collect the results.
    tbl = table(bboxes, scores, labels, 'VariableNames', {'Boxes', 'Scores', 'Labels'});
    results = [results; tbl];
end

% Evaluate the object detector using Average Precision metric.
[ap, recall, precision] = evaluateDetectionPrecision(results, preprocessedTestData);

The precision-recall (PR) curve shows how precise a detector is at varying levels of recall. Ideally, the
precision is 1 at all recall levels.

% Plot precision-recall curve.
figure
plot(recall, precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))

```



Detect Objects Using YOLO v3

Use the network for object detection.

- Read an image.
- Convert the image to a `dIarray` and use a GPU if one is available..
- Use the supporting function `yoloV3Detect` to get the predicted bounding boxes, confidence scores, and class labels.
- Display the image with bounding boxes and confidence scores.

```
% Read the datastore.
reset(preprocessedTestData)
data = read(preprocessedTestData);

% Get the image.
I = data{1};

% Convert to dIarray.
XTest = dIarray(I, 'SSCB');

executionEnvironment = "auto";

% If GPU is available, then convert data to gpuArray.
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    XTest = gpuArray(XTest);
end
```

```
[bboxes, scores, labels] = yolov3Detect(net, XTest, networkOutputs, anchorBoxes, anchorBoxMasks,
% Clear the persistent variables used in the yolov3Detect function to avoid retaining their values
clear yolov3Detect

% Display the detections on image.
if ~isempty(scores{1})
    I = insertObjectAnnotation(I, 'rectangle', bboxes{1}, scores{1});
end
figure
imshow(I)
```



Supporting Functions

Model Gradients Function

The function `modelGradients` takes as input the `dlnetwork` object `net`, a mini-batch of input data `XTrain` with corresponding ground truth boxes `YTrain`, anchor boxes, anchor box mask, the specified penalty threshold, and the network output names as input arguments and returns the gradients of the loss with respect to the learnable parameters in `net`, the corresponding mini-batch loss, and the state of the current batch.

The model gradients function computes the total loss and gradients by performing these operations.

- Generate predictions from the input batch of images using the supporting function `yolov3Forward`.
- Collect predictions on the CPU for postprocessing.
- Convert the predictions from the YOLO v3 grid cell coordinates to bounding box coordinates to allow easy comparison with the ground truth data by using the supporting functions `generateTiledAnchors` and `applyAnchorBoxOffsets`.

- Generate targets for loss computation by using the converted predictions and ground truth data. These targets are generated for bounding box positions (x, y, width, height), object confidence, and class probabilities. See the supporting function `generateTargets`.
- Calculates the mean squared error of the predicted bounding box coordinates with target boxes. See the supporting function `bboxOffsetLoss`.
- Determines the binary cross-entropy of the predicted object confidence score with target object confidence score. See the supporting function `objectnessLoss`.
- Determines the binary cross-entropy of the predicted class of object with the target. See the supporting function `classConfidenceLoss`.
- Computes the total loss as the sum of all losses.
- Computes the gradients of learnables with respect to the total loss.

```
function [gradients, state, info] = modelGradients(net, XTrain, YTrain, anchors, mask, penaltyThresh,
inputImageSize = size(XTrain,1:2);
```

```
% Gather the ground truths in the CPU for post processing
YTrain = gather(extractdata(YTrain));
```

```
% Extract the predictions from the network.
[YPredCell, state] = yolov3Forward(net,XTrain,networkOutputs,mask);
```

```
% Gather the activations in the CPU for post processing and extract dlarray data.
gatheredPredictions = cellfun(@ gather, YPredCell(:,1:6),'UniformOutput',false);
gatheredPredictions = cellfun(@ extractdata, gatheredPredictions, 'UniformOutput', false);
```

```
% Convert predictions from grid cell coordinates to box coordinates.
tiledAnchors = generateTiledAnchors(gatheredPredictions(:,2:5),anchors,mask);
gatheredPredictions(:,2:5) = applyAnchorBoxOffsets(tiledAnchors, gatheredPredictions(:,2:5), inputImageSize);
```

```
% Generate target for predictions from the ground truth data.
[boxTarget, objectnessTarget, classTarget, objectMaskTarget, boxErrorScale] = generateTargets(gatheredPredictions, YTrain, anchors, mask, penaltyThresh, inputImageSize);
```

```
% Compute the loss.
boxLoss = bboxOffsetLoss(YPredCell(:,[2 3 7 8]),boxTarget,objectMaskTarget,boxErrorScale);
objLoss = objectnessLoss(YPredCell(:,1),objectnessTarget,objectMaskTarget);
clsLoss = classConfidenceLoss(YPredCell(:,6),classTarget,objectMaskTarget);
totalLoss = boxLoss + objLoss + clsLoss;
```

```
info.boxLoss = boxLoss;
info.objLoss = objLoss;
info.clsLoss = clsLoss;
info.totalLoss = totalLoss;
```

```
% Compute gradients of learnables with regard to loss.
gradients = dlgradient(totalLoss, net.Learnables);
end
```

```
function [YPredCell, state] = yolov3Forward(net, XTrain, networkOutputs, anchorBoxMask)
% Predict the output of network and extract the confidence score, x, y,
% width, height, and class.
YPredictions = cell(size(networkOutputs));
[YPredictions{:}, state] = forward(net, XTrain, 'Outputs', networkOutputs);
YPredCell = extractPredictions(YPredictions, anchorBoxMask);
```

```
% Append predicted width and height to the end as they are required
```

```

% for computing the loss.
YPredCell(:,7:8) = YPredCell(:,4:5);

% Apply sigmoid and exponential activation.
YPredCell(:,1:6) = applyActivations(YPredCell(:,1:6));
end

function boxLoss = bboxOffsetLoss(boxPredCell, boxDeltaTarget, boxMaskTarget, boxErrorScaleTarget)
% Mean squared error for bounding box position.
lossX = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,1),boxDeltaTarget(:,1),boxMaskTarget(:,1)),1));
lossY = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,2),boxDeltaTarget(:,2),boxMaskTarget(:,2)),1));
lossW = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,3),boxDeltaTarget(:,3),boxMaskTarget(:,3)),1));
lossH = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,4),boxDeltaTarget(:,4),boxMaskTarget(:,4)),1));
boxLoss = lossX+lossY+lossW+lossH;
end

function objLoss = objectnessLoss(objectnessPredCell, objectnessDeltaTarget, boxMaskTarget)
% Binary cross-entropy loss for objectness score.
objLoss = sum(cellfun(@(a,b,c) crossentropy(a.*c,b.*c,'TargetCategories','independent'),objectnessPredCell,objectnessDeltaTarget,boxMaskTarget),1);
end

function clsLoss = classConfidenceLoss(classPredCell, classTarget, boxMaskTarget)
% Binary cross-entropy loss for class confidence score.
clsLoss = sum(cellfun(@(a,b,c) crossentropy(a.*c,b.*c,'TargetCategories','independent'),classPredCell,classTarget,boxMaskTarget),1);
end

```

Augmentation and Data Processing Functions

```

function data = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.

data = cell(size(A));
for ii = 1:size(A,1)
    I = A{ii,1};
    bboxes = A{ii,2};
    labels = A{ii,3};
    sz = size(I);

    if numel(sz) == 3 && sz(3) == 3
        I = jitterColorHSV(I,...
            'Contrast',0.0,...
            'Hue',0.1,...
            'Saturation',0.2,...
            'Brightness',0.2);
    end

    % Randomly flip image.
    tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
    rout = affineOutputView(sz,tform,'BoundsStyle','centerOutput');
    I = imwarp(I,tform,'OutputView',rout);

    % Apply same transform to boxes.
    [bboxes,indices] = bboxwarp(bboxes,tform,rout,'OverlapThreshold',0.25);
    labels = labels(indices);

    % Return original data only when all boxes are removed by warping.
end

```

```

    if isempty(indices)
        data(ii,:) = A(ii,:);
    else
        data(ii,:) = {I, bboxes, labels};
    end
end
end

```

```

function data = preprocessData(data, targetSize)
% Resize the images and scale the pixels to between 0 and 1. Also scale the
% corresponding bounding boxes.

```

```

for ii = 1:size(data,1)
    I = data{ii,1};
    imgSize = size(I);

    % Convert an input image with single channel to 3 channels.
    if numel(imgSize) < 3
        I = repmat(I,1,1,3);
    end
    bboxes = data{ii,2};

    I = im2single(imresize(I,targetSize(1:2)));
    scale = targetSize(1:2)./imgSize(1:2);
    bboxes = bboxresize(bboxes,scale);

    data(ii, 1:2) = {I, bboxes};
end
end

```

```

function [XTrain, YTrain] = createBatchData(data, groundTruthBoxes, groundTruthClasses, className)
% Returns images combined along the batch dimension in XTrain and
% normalized bounding boxes concatenated with classIDs in YTrain

```

```

% Concatenate images along the batch dimension.
XTrain = cat(4, data{:},1);

% Get class IDs from the class names.
classNames = repmat({categorical(className)}, size(groundTruthClasses));
[~, classIndices] = cellfun(@(a,b)ismember(a,b), groundTruthClasses, classNames, 'UniformOutput'

% Append the label indexes and training image size to scaled bounding boxes
% and create a single cell array of responses.
combinedResponses = cellfun(@(bbox, classid)[bbox, classid], groundTruthBoxes, classIndices, 'UniformOutput'
len = max( cellfun(@(x)size(x,1), combinedResponses ) );
paddedBBoxes = cellfun( @(v) padarray(v,[len-size(v,1),0],0,'post'), combinedResponses, 'UniformOutput'
YTrain = cat(4, paddedBBoxes{:},1);
end

```

Network Creation Functions

```

function lgraph = squeezeNetFeatureExtractor(net, imageInputSize)
% The squeezeNetFeatureExtractor function removes the layers after 'fire9-concat'
% in SqueezeNet and also removes any data normalization used by the image input layer.

% Convert to layerGraph.
lgraph = layerGraph(net);

```



```

lgraph = removeLayers(lgraph, {'drop9' 'conv10' 'relu_conv10' 'pool10' 'prob' 'ClassificationLayer'});
inputLayer = imageInputLayer(imageInputSize, 'Normalization', 'none', 'Name', 'data');
lgraph = replaceLayer(lgraph, 'data', inputLayer);
end

function lgraph = addFirstDetectionHead(lgraph, anchorBoxMasks, numPredictorsPerAnchor)
% The addFirstDetectionHead function adds the first detection head.

numAnchorsScale1 = size(anchorBoxMasks, 2);
% Compute the number of filters for last convolution layer.
numFilters = numAnchorsScale1*numPredictorsPerAnchor;
firstDetectionSubNetwork = [
    convolution2dLayer(3,256, 'Padding', 'same', 'Name', 'conv1Detection1', 'WeightsInitializer', 'he');
    reluLayer('Name', 'relu1Detection1');
    convolution2dLayer(1,numFilters, 'Padding', 'same', 'Name', 'conv2Detection1', 'WeightsInitializer');
];
lgraph = addLayers(lgraph, firstDetectionSubNetwork);
end

function lgraph = addSecondDetectionHead(lgraph, anchorBoxMasks, numPredictorsPerAnchor)
% The addSecondDetectionHead function adds the second detection head.

numAnchorsScale2 = size(anchorBoxMasks, 2);
% Compute the number of filters for the last convolution layer.
numFilters = numAnchorsScale2*numPredictorsPerAnchor;

secondDetectionSubNetwork = [
    upsampleLayer(2, 'upsample1Detection2');
    depthConcatenationLayer(2, 'Name', 'depthConcat1Detection2');
    convolution2dLayer(3,128, 'Padding', 'same', 'Name', 'conv1Detection2', 'WeightsInitializer', 'he');
    reluLayer('Name', 'relu1Detection2');
    convolution2dLayer(1,numFilters, 'Padding', 'same', 'Name', 'conv2Detection2', 'WeightsInitializer');
];
lgraph = addLayers(lgraph, secondDetectionSubNetwork);
end

```

Learning Rate Schedule Function

```

function currentLR = piecewiseLearningRateWithWarmup(iteration, epoch, learningRate, warmupPeriod)
% The piecewiseLearningRateWithWarmup function computes the current
% learning rate based on the iteration number.
persistent warmUpEpoch;

if iteration <= warmupPeriod
    % Increase the learning rate for number of iterations in warmup period.
    currentLR = learningRate * ((iteration/warmupPeriod)^4);
    warmUpEpoch = epoch;
elseif iteration >= warmupPeriod && epoch < warmUpEpoch+floor(0.6*(numEpochs-warmUpEpoch))
    % After warm up period, keep the learning rate constant if the remaining number of epochs is
    currentLR = learningRate;

elseif epoch >= warmUpEpoch + floor(0.6*(numEpochs-warmUpEpoch)) && epoch < warmUpEpoch+floor(0.9*(numEpochs-warmUpEpoch))
    % If the remaining number of epochs is more than 60 percent but less
    % than 90 percent multiply the learning rate by 0.1.
    currentLR = learningRate*0.1;

else

```

```

    % If remaining epochs are more than 90 percent multiply the learning
    % rate by 0.01.
    currentLR = learningRate*0.01;
end
end

```

Predict Functions

```

function [bboxes,scores,labels] = yolov3Detect(net, XTest, networkOutputs, anchors, anchorBoxMask)
% The yolov3Detect function detects the bounding boxes, scores, and labels in an image.

```

```

imageSize = size(XTest, [1,2]);

```

```

% Find the input image layer and get the network input size. To retain 'networkInputSize' in memory
% recalculating it, declare it as persistent.
persistent networkInputSize

```

```

if isempty(networkInputSize)
    networkInputIdx = arrayfun( @(x)isa(x,'nnet.cnn.layer.ImageInputLayer'), net.Layers);
    networkInputSize = net.Layers(networkInputIdx).InputSize;
end

```

```

% Predict and filter the detections based on confidence threshold.
predictions = yolov3Predict(net,XTest,networkOutputs,anchorBoxMask);
predictions = cellfun(@ gather, predictions, 'UniformOutput', false);
predictions = cellfun(@ extractdata, predictions, 'UniformOutput', false);
tiledAnchors = generateTiledAnchors(predictions(:,2:5),anchors,anchorBoxMask);
predictions(:,2:5) = applyAnchorBoxOffsets(tiledAnchors, predictions(:,2:5), networkInputSize);

```

```

numMiniBatch = size(XTest, 4);

```

```

bboxes = cell(numMiniBatch, 1);
scores = cell(numMiniBatch, 1);
labels = cell(numMiniBatch, 1);

```

```

for ii = 1:numMiniBatch
    fmap = cellfun(@(x) x(:,:,,ii), predictions, 'UniformOutput', false);
    [bboxes{ii}, scores{ii}, labels{ii}] = ...
        generateYOLOv3Detections(fmap, confidenceThreshold, overlapThreshold, imageSize, classes);
end

```

```

end

```

```

function YPredCell = yolov3Predict(net,XTrain,networkOutputs,anchorBoxMask)
% Predict the output of network and extract the confidence, x, y,
% width, height, and class.
YPredictions = cell(size(networkOutputs));
[YPredictions{:}] = predict(net, XTrain);
YPredCell = extractPredictions(YPredictions, anchorBoxMask);

```

```

% Apply activation to the predicted cell array.
YPredCell = applyActivations(YPredCell);
end

```

Utility Functions

```

function YPredCell = applyActivations(YPredCell)
YPredCell(:,1:3) = cellfun(@ sigmoid, YPredCell(:,1:3), 'UniformOutput', false);
YPredCell(:,4:5) = cellfun(@ exp, YPredCell(:,4:5), 'UniformOutput', false);
YPredCell(:,6) = cellfun(@ sigmoid, YPredCell(:,6), 'UniformOutput', false);
end

function predictions = extractPredictions(YPredictions, anchorBoxMask)
predictions = cell(size(YPredictions, 1),6);
for ii = 1:size(YPredictions, 1)
    % Get the required info on feature size.
    numChannelsPred = size(YPredictions{ii},3);
    numAnchors = size(anchorBoxMask{ii},2);
    numPredElemsPerAnchors = numChannelsPred/numAnchors;
    allIds = (1:numChannelsPred);

    stride = numPredElemsPerAnchors;
    endIdx = numChannelsPred;

    % X positions.
    startIdx = 1;
    predictions{ii,2} = YPredictions{ii}(:, :, startIdx:stride:endIdx, :);
    xIds = startIdx:stride:endIdx;

    % Y positions.
    startIdx = 2;
    predictions{ii,3} = YPredictions{ii}(:, :, startIdx:stride:endIdx, :);
    yIds = startIdx:stride:endIdx;

    % Width.
    startIdx = 3;
    predictions{ii,4} = YPredictions{ii}(:, :, startIdx:stride:endIdx, :);
    wIds = startIdx:stride:endIdx;

    % Height.
    startIdx = 4;
    predictions{ii,5} = YPredictions{ii}(:, :, startIdx:stride:endIdx, :);
    hIds = startIdx:stride:endIdx;

    % Confidence scores.
    startIdx = 5;
    predictions{ii,1} = YPredictions{ii}(:, :, startIdx:stride:endIdx, :);
    confIds = startIdx:stride:endIdx;

    % Accumulate all the non-class indexes
    nonClassIds = [xIds yIds wIds hIds confIds];

    % Class probabilities.
    % Get the indexes which do not belong to the nonClassIds
    classIdx = setdiff(allIds, nonClassIds);
    predictions{ii,6} = YPredictions{ii}(:, :, classIdx, :);
end
end

function tiledAnchors = generateTiledAnchors(YPredCell, anchorBoxes, anchorBoxMask)
% Generate tiled anchor offset.
tiledAnchors = cell(size(YPredCell));

```

```

for i=1:size(YPredCell,1)
    anchors = anchorBoxes(anchorBoxMask{i}, :);
    [h,w,~,n] = size(YPredCell{i,1});
    [tiledAnchors{i,2}, tiledAnchors{i,1}] = ndgrid(0:h-1,0:w-1,1:size(anchors,1),1:n);
    [~,~,tiledAnchors{i,3}] = ndgrid(0:h-1,0:w-1,anchors(:,2),1:n);
    [~,~,tiledAnchors{i,4}] = ndgrid(0:h-1,0:w-1,anchors(:,1),1:n);
end
end

function tiledAnchors = applyAnchorBoxOffsets(tiledAnchors,YPredCell,inputImageSize)
% Convert grid cell coordinates to box coordinates.
for i=1:size(YPredCell,1)
    [h,w,~,~] = size(YPredCell{i,1});
    tiledAnchors{i,1} = (tiledAnchors{i,1}+YPredCell{i,1})./w;
    tiledAnchors{i,2} = (tiledAnchors{i,2}+YPredCell{i,2})./h;
    tiledAnchors{i,3} = (tiledAnchors{i,3}.*YPredCell{i,3})./inputImageSize(2);
    tiledAnchors{i,4} = (tiledAnchors{i,4}.*YPredCell{i,4})./inputImageSize(1);
end
end

function [lossPlotter, learningRatePlotter] = configureTrainingProgressPlotter(f)
% Create the subplots to display the loss and learning rate.
figure(f);
clf
subplot(2,1,1);
ylabel('Learning Rate');
xlabel('Iteration');
learningRatePlotter = animatedline;
subplot(2,1,2);
ylabel('Total Loss');
xlabel('Iteration');
lossPlotter = animatedline;
end

function displayLossInfo(epoch, iteration, currentLR, lossInfo)
% Display loss information for each iteration.
disp("Epoch : " + epoch + " | Iteration : " + iteration + " | Learning Rate : " + currentLR + ..
    " | Total Loss : " + double(gather(extractdata(lossInfo.totalLoss))) + ...
    " | Box Loss : " + double(gather(extractdata(lossInfo.boxLoss))) + ...
    " | Object Loss : " + double(gather(extractdata(lossInfo.objLoss))) + ...
    " | Class Loss : " + double(gather(extractdata(lossInfo.clsLoss))));
end

function updatePlots(lossPlotter, learningRatePlotter, iteration, currentLR, totalLoss)
% Update loss and learning rate plots.
addpoints(lossPlotter, iteration, double(extractdata(gather(totalLoss))));
addpoints(learningRatePlotter, iteration, currentLR);
drawnow
end

function net = downloadPretrainedYOLOv3Detector()
% Download a pretrained yolov3 detector.
if ~exist('yolov3SqueezeNetVehicleExample_20b.mat', 'file')
    if ~exist('yolov3SqueezeNetVehicleExample_20b.zip', 'file')
        disp('Downloading pretrained detector (8.9 MB)...');
        pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/yolov3SqueezeNetVehi
        websave('yolov3SqueezeNetVehicleExample_20b.zip', pretrainedURL);
    end
end

```

```
        unzip('yolov3SqueezeNetVehicleExample_20b.zip');  
end  
pretrained = load("yolov3SqueezeNetVehicleExample_20b.mat");  
net = pretrained.net;  
end
```

References

1. Redmon, Joseph, and Ali Farhadi. "YOLOv3: An Incremental Improvement." Preprint, submitted April 8, 2018. <https://arxiv.org/abs/1804.02767>.

Object Detection Using YOLO v2 Deep Learning

This example shows how to train a you only look once (YOLO) v2 object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a YOLO v2 vehicle detector using the `trainYOLOv2ObjectDetector` function. For more information, see “Object Detection using Deep Learning”.

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('yolov2ResNet50VehicleExample_19b.mat','file')
    disp('Downloading pretrained detector (98 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/yolov2ResNet50VehicleExample_19b.mat';
    websave('yolov2ResNet50VehicleExample_19b.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small vehicle dataset that contains 295 images. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the YOLO v2 training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip('vehicleDatasetImages.zip');
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

```
% Display first few rows of the data set.
vehicleDataset(1:4,:)
```

```
ans=4x2 table
```

imageFilename	vehicle
'vehicleImages/image_00001.jpg'	{1x4 double}
'vehicleImages/image_00002.jpg'	{1x4 double}
'vehicleImages/image_00003.jpg'	{1x4 double}
'vehicleImages/image_00004.jpg'	{1x4 double}

```
% Add the fullpath to the local vehicle data folder.
vehicleDataset.imageFilename = fullfile(pwd,vehicleDataset.imageFilename);
```

Split the dataset into training, validation, and test sets. Select 60% of the data for training, 10% for validation, and the rest for testing the trained detector.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
```

```

idx = floor(0.6 * length(shuffledIndices) );

trainingIdx = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);

validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);

testIdx = validationIdx(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIdx),:);

```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

```

imdsTrain = imageDatastore(trainingDataTbl{:, 'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:, 'vehicle'));

imdsValidation = imageDatastore(validationDataTbl{:, 'imageFilename'});
bldsValidation = boxLabelDatastore(validationDataTbl(:, 'vehicle'));

imdsTest = imageDatastore(testDataTbl{:, 'imageFilename'});
bldsTest = boxLabelDatastore(testDataTbl(:, 'vehicle'));

```

Combine image and box label datastores.

```

trainingData = combine(imdsTrain,bldsTrain);
validationData = combine(imdsValidation,bldsValidation);
testData = combine(imdsTest,bldsTest);

```

Display one of the training images and box labels.

```

data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I, 'Rectangle', bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)

```



Create a YOLO v2 Object Detection Network

A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. The feature extraction network is typically a pretrained CNN (for details, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox)). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18 can also be used depending on application requirements. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific for YOLO v2.

Use the `yoLov2Layers` function to create a YOLO v2 object detection network automatically given a pretrained ResNet-50 feature extraction network. `yoLov2Layers` requires you to specify several inputs that parameterize a YOLO v2 network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size and the number of classes. When choosing the network input size, consider the minimum size required by the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of `[224 224 3]`, which is the minimum size required to run the network.


```
inputSize = [224 224 3];
```

Define the number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes. Resize the training data to the input image size of the network using the supporting function `preprocessData`.

```
trainingDataForEstimation = transform(trainingData,@(data)preprocessData(data,inputSize));
numAnchors = 7;
[anchorBoxes, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchorBoxes = 7×2
```

```
    145    126
     91     86
    161    132
     41     34
     67     64
    136    111
     33     23
```

```
meanIoU = 0.8651
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” on page 3-146 (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” on page 14-21.

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select `'activation_40_relu'` as the feature extraction layer to replace the layers after `'activation_40_relu'` with the detection subnetwork. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis.

```
featureLayer = 'activation_40_relu';
```

Create the YOLO v2 object detection network.

```
lgraph = yolov2Layers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer);
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the YOLO v2 network architecture, use Deep Network Designer to design the YOLO v2 detection network manually. For more information, see “Design a YOLO v2 Detection Network” on page 14-27.

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
% Visualize the augmented images.
```

```
augmentedData = cell(4,1);
```

```
for k = 1:4
```

```
    data = read(augmentedTrainingData);
```

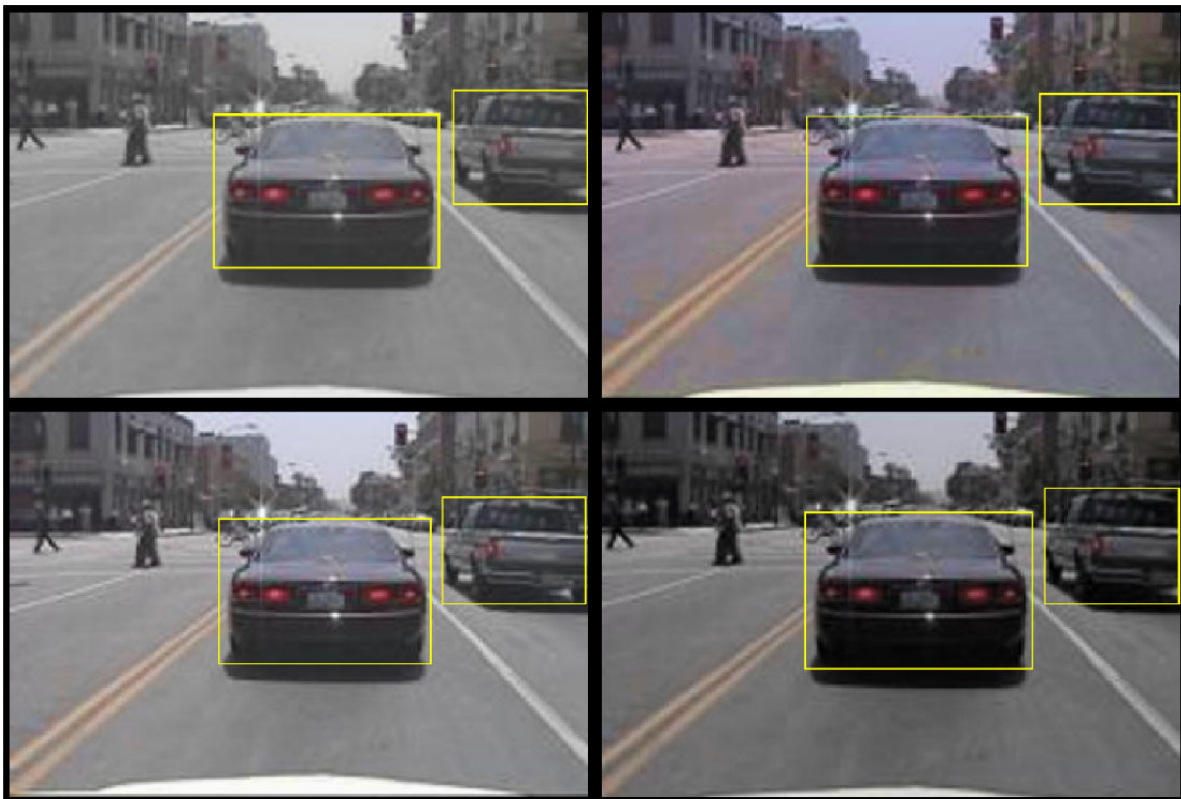
```
    augmentedData{k} = insertShape(data{1}, 'Rectangle', data{2});
```

```
    reset(augmentedTrainingData);
```

```
end
```

```
figure
```

```
montage(augmentedData, 'BorderSize', 10)
```



Preprocess Training Data

Preprocess the augmented training data, and the validation data to prepare for training.

```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))  
preprocessedValidationData = transform(validationData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image and bounding boxes.

```
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I,'Rectangle',bbox);  
annotatedImage = imresize(annotatedImage,2);  
figure  
imshow(annotatedImage)
```



Train YOLO v2 Object Detector

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',16, ...
    'InitialLearnRate',1e-3, ...
    'MaxEpochs',20,...
    'CheckpointPath',tempdir, ...
    'ValidationData',preprocessedValidationData);
```

Use `trainYOLOv2ObjectDetector` function to train YOLO v2 object detector if `doTraining` is true. Otherwise, load the pretrained network.

```
if doTraining
    % Train the YOLO v2 detector.
    [detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,options);
else
    % Load pretrained detector for the example.
    pretrained = load('yoloV2ResNet50VehicleExample_19b.mat');
    detector = pretrained.detector;
end
```

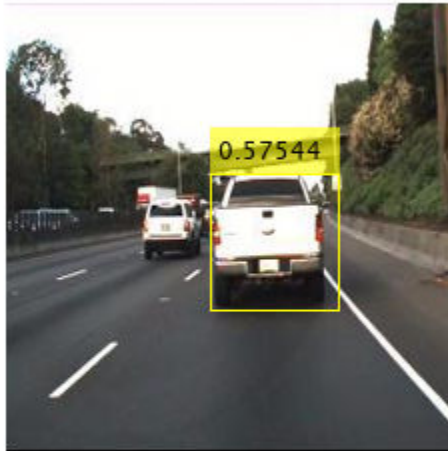
This example was verified on an NVIDIA™ Titan X GPU with 12 GB of memory. If your GPU has less memory, you may run out of memory. If this happens, lower the `'MiniBatchSize'` using the `trainingOptions` function. Training this network took approximately 7 minutes using this setup. Training time varies depending on the hardware you use.

As a quick test, run the detector on a test image. Make sure you resize the image to the same size as the training images.

```
I = imread('highway.png');
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```



Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Test data should be representative of the original data and be left unmodified for unbiased evaluation.

```
preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

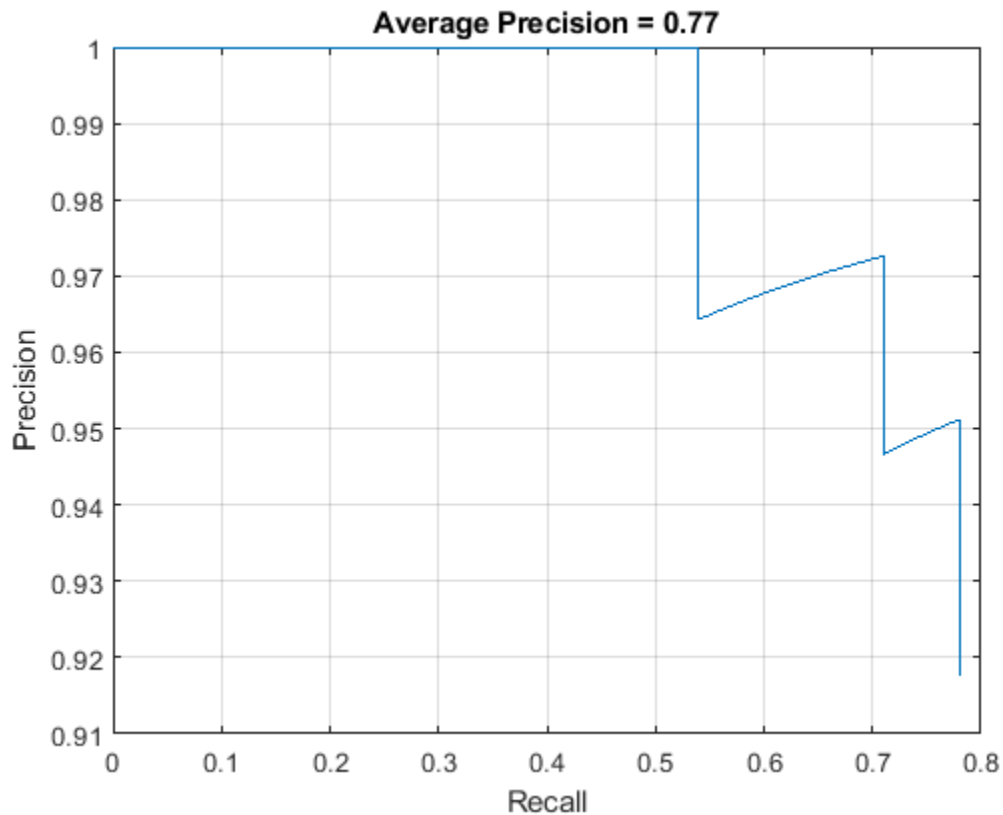
```
detectionResults = detect(detector, preprocessedTestData);
```

Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults, preprocessedTestData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```



Code Generation

Once the detector is trained and evaluated, you can generate code for the `yoloV2ObjectDetector` using GPU Coder™. See “Code Generation for Object Detection by Using YOLO v2” (GPU Coder) example for more details.

Supporting Functions

```
function B = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.
B = cell(size(A));

I = A{1};
sz = size(I);
if numel(sz)==3 && sz(3) == 3
    I = jitterColorHSV(I,...
        'Contrast',0.2,...
        'Hue',0,...
        'Saturation',0.1,...
        'Brightness',0.2);
end

% Randomly flip and scale image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');
```

```
B{1} = imwarp(I,tform,'OutputView',rout);

% Apply same transform to boxes.
[B{2},indices] = bboxwarp(A{2},tform,rout,'OverlapThreshold',0.25);
B{3} = A{3}(indices);

% Return original data only when all boxes are removed by warping.
if isempty(indices)
    B = A;
end
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to the targetSize.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

Create YOLO v2 Object Detection Network

This example shows how to modify a pretrained MobileNet v2 network to create a YOLO v2 object detection network.

The procedure to convert a pretrained network into a YOLO v2 network is similar to the transfer learning procedure for image classification:

- 1 Load the pretrained network.
- 2 Select a layer from the pretrained network to use for feature extraction.
- 3 Remove all layers after the feature extraction layer.
- 4 Add new layers to support the object detection task.

Load Pretrained Network

Load a pretrained MobileNet v2 network using `mobilenetv2`. This requires the Deep Learning Toolbox Model for MobileNet v2 Network™ support package. If this support package is not installed, then the function provides a download link. After you load the network, convert the network into a `layerGraph` object so that you can manipulate the layers.

```
net = mobilenetv2();
lgraph = layerGraph(net);
```

Update Network Input Size

Update the network input size to meet the training data requirements. For example, assume the training data are 300-by-300 RGB images. Set the input size.

```
imageInputSize = [300 300 3];
```

Next, create a new image input layer with the same name as the original layer.

```
imgLayer = imageInputLayer(imageInputSize, "Name", "input_1")
```

```
imgLayer =
  ImageInputLayer with properties:
      Name: 'input_1'
      InputSize: [300 300 3]

  Hyperparameters
      DataAugmentation: 'none'
      Normalization: 'zerocenter'
      NormalizationDimension: 'auto'
      Mean: []
```

Replace the old image input layer with the new image input layer.

```
lgraph = replaceLayer(lgraph, "input_1", imgLayer);
```

Select Feature Extraction Layer

A YOLO v2 feature extraction layer is most effective when the output feature width and height are between 8 and 16 times smaller than the input image. This amount of downsampling is a trade-off between spatial resolution and output-feature quality. You can use the `analyzeNetwork` function or

the Deep Network Designer app to determine the output sizes of layers within a network. Note that selecting an optimal feature extraction layer requires empirical evaluation.

Set the feature extraction layer to "block_12_add". The output size of this layer is about 16 times smaller than the input image size of 300-by-300.

```
featureExtractionLayer = "block_12_add";
```

Remove Layers After Feature Extraction Layer

Next, remove the layers after the feature extraction layer. You can do so by importing the network into the Deep Network Designer app, manually removing the layers, and exporting the modified the network to your workspace.

For this example, load the modified network, which has been added to this example as a supporting file.

```
modified = load("mobilenetv2Block12Add.mat");
lgraph = modified.mobilenetv2Block12Add;
```

Create YOLO v2 Detection Sub-Network

The detection subnetwork consists of groups of serially connected convolution, ReLU, and batch normalization layers. These layers are followed by a `yolov2TransformLayer` and a `yolov2OutputLayer`.

First, create two groups of serially connected convolution, ReLU, and batch normalization layers. Set the convolution layer filter size to 3-by-3 and the number of filters to match the number of channels in the feature extraction layer output. Specify "same" padding in the convolution layer to preserve the input size.

```
filterSize = [3 3];
numFilters = 96;
```

```
detectionLayers = [
    convolution2dLayer(filterSize,numFilters,"Name","yolov2Conv1","Padding", "same", "WeightsIni
    batchNormalizationLayer("Name","yolov2Batch1")
    reluLayer("Name","yolov2Relu1")
    convolution2dLayer(filterSize,numFilters,"Name","yolov2Conv2","Padding", "same", "WeightsIni
    batchNormalizationLayer("Name","yolov2Batch2")
    reluLayer("Name","yolov2Relu2")
]
```

```
detectionLayers =
    6x1 Layer array with layers:
```

1	'yolov2Conv1'	Convolution	96 3x3 convolutions with stride [1 1] and padding
2	'yolov2Batch1'	Batch Normalization	Batch normalization
3	'yolov2Relu1'	ReLU	ReLU
4	'yolov2Conv2'	Convolution	96 3x3 convolutions with stride [1 1] and padding
5	'yolov2Batch2'	Batch Normalization	Batch normalization
6	'yolov2Relu2'	ReLU	ReLU

Next, create the final portion of the detection subnetwork, which has a convolution layer followed by a `yolov2TransformLayer` and a `yolov2OutputLayer`. The output of convolution layer predicts the following for each anchor box:

- 1 The object class probabilities.
- 2 The x and y location offset.
- 3 The width and height offset.

Specify the anchor boxes and number of classes and compute the number of filters for the convolution layer.

```
numClasses = 5;

anchorBoxes = [
    16 16
    32 16
];

numAnchors = size(anchorBoxes,1);
numPredictionsPerAnchor = 5;
numFiltersInLastConvLayer = numAnchors*(numClasses+numPredictionsPerAnchor);
```

Add the convolution2dLayer, yolov2TransformLayer, and yolov2OutputLayer to the detection subnetwork.

```
detectionLayers = [
    detectionLayers
    convolution2dLayer(1,numFiltersInLastConvLayer,"Name","yolov2ClassConv",...
        "WeightsInitializer", @(sz)randn(sz)*0.01)
    yolov2TransformLayer(numAnchors,"Name","yolov2Transform")
    yolov2OutputLayer(anchorBoxes,"Name","yolov2OutputLayer")
];
```

detectionLayers =
9x1 Layer array with layers:

1	'yolov2Conv1'	Convolution	96 3x3 convolutions with stride [1 1]
2	'yolov2Batch1'	Batch Normalization	Batch normalization
3	'yolov2Relu1'	ReLU	ReLU
4	'yolov2Conv2'	Convolution	96 3x3 convolutions with stride [1 1]
5	'yolov2Batch2'	Batch Normalization	Batch normalization
6	'yolov2Relu2'	ReLU	ReLU
7	'yolov2ClassConv'	Convolution	20 1x1 convolutions with stride [1 1]
8	'yolov2Transform'	YOLO v2 Transform Layer.	YOLO v2 Transform Layer with 2 anchors
9	'yolov2OutputLayer'	YOLO v2 Output	YOLO v2 Output with 2 anchors.

Complete YOLO v2 Detection Network

Attach the detection subnetwork to the feature extraction network.

```
lgraph = addLayers(lgraph,detectionLayers);
lgraph = connectLayers(lgraph,featureExtractionLayer,"yolov2Conv1");
```

Use analyzeNetwork(lgraph) to check the network and then train a YOLO v2 object detector using the trainYOLOv2ObjectDetector function.

Train Object Detector Using R-CNN Deep Learning

This example shows how to train an object detector using deep learning and R-CNN (Regions with Convolutional Neural Networks).

Overview

This example shows how to train an R-CNN object detector for detecting stop signs. R-CNN is an object detection framework, which uses a convolutional neural network (CNN) to classify image regions within an image [1]. Instead of classifying every region using a sliding window, the R-CNN detector only processes those regions that are likely to contain an object. This greatly reduces the computational cost incurred when running a CNN.

To illustrate how to train an R-CNN stop sign detector, this example follows the transfer learning workflow that is commonly used in deep learning applications. In transfer learning, a network trained on a large collection of images, such as ImageNet [2], is used as the starting point to solve a new classification or detection task. The advantage of using this approach is that the pretrained network has already learned a rich set of image features that are applicable to a wide range of images. This learning is transferable to the new task by fine-tuning the network. A network is fine-tuned by making small adjustments to the weights such that the feature representations learned for the original task are slightly adjusted to support the new task.

The advantage of transfer learning is that the number of images required for training and the training time are reduced. To illustrate these advantages, this example trains a stop sign detector using the transfer learning workflow. First a CNN is pretrained using the CIFAR-10 data set, which has 50,000 training images. Then this pretrained CNN is fine-tuned for stop sign detection using just 41 training images. Without pretraining the CNN, training the stop sign detector would require many more images.

Note: This example requires Computer Vision Toolbox™, Image Processing Toolbox™, Deep Learning Toolbox™, and Statistics and Machine Learning Toolbox™.

Using a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for running this example. Use of a GPU requires the Parallel Computing Toolbox™.

Download CIFAR-10 Image Data

Download the CIFAR-10 data set [3]. This dataset contains 50,000 training images that will be used to train a CNN.

Download CIFAR-10 data to a temporary directory

```
cifar10Data = tempdir;
url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';
helperCIFAR10Data.download(url,cifar10Data);
```

Load the CIFAR-10 training and test data.

```
[trainingImages,trainingLabels,testImages,testLabels] = helperCIFAR10Data.load(cifar10Data);
```

Each image is a 32x32 RGB image and there are 50,000 training samples.

```
size(trainingImages)
```

```
ans = 1x4
      32      32      3      50000
```

CIFAR-10 has 10 image categories. List the image categories:

```
numImageCategories = 10;
categories(trainingLabels)
```

```
ans = 10x1 cell
      {'airplane' }
      {'automobile'}
      {'bird' }
      {'cat' }
      {'deer' }
      {'dog' }
      {'frog' }
      {'horse' }
      {'ship' }
      {'truck' }
```

You can display a few of the training images using the following code.

```
figure
thumbnails = trainingImages(:, :, :, 1:100);
montage(thumbnails)
```

Create A Convolutional Neural Network (CNN)

A CNN is composed of a series of layers, where each layer defines a specific computation. The Deep Learning Toolbox™ provides functionality to easily design a CNN layer-by-layer. In this example, the following layers are used to create a CNN:

- `imageInputLayer` (Deep Learning Toolbox) - Image input layer
- `convolution2dLayer` (Deep Learning Toolbox) - 2D convolution layer for Convolutional Neural Networks
- `reluLayer` (Deep Learning Toolbox) - Rectified linear unit (ReLU) layer
- `maxPooling2dLayer` (Deep Learning Toolbox) - Max pooling layer
- `fullyConnectedLayer` (Deep Learning Toolbox) - Fully connected layer
- `softmaxLayer` (Deep Learning Toolbox) - Softmax layer
- `classificationLayer` (Deep Learning Toolbox) - Classification output layer for a neural network

The network defined here is similar to the one described in [4] and starts with an `imageInputLayer`. The input layer defines the type and size of data the CNN can process. In this example, the CNN is used to process CIFAR-10 images, which are 32x32 RGB images:

```
% Create the image input layer for 32x32x3 CIFAR-10 images.
[height,width,numChannels, ~] = size(trainingImages);

imageSize = [height width numChannels];
inputLayer = imageInputLayer(imageSize)
```

```

inputLayer =
  ImageInputLayer with properties:
      Name: ''
      InputSize: [32 32 3]
  Hyperparameters
      DataAugmentation: 'none'
      Normalization: 'zerocenter'
  NormalizationDimension: 'auto'
      Mean: []

```

Next, define the middle layers of the network. The middle layers are made up of repeated blocks of convolutional, ReLU (rectified linear units), and pooling layers. These 3 layers form the core building blocks of convolutional neural networks. The convolutional layers define sets of filter weights, which are updated during network training. The ReLU layer adds non-linearity to the network, which allow the network to approximate non-linear functions that map image pixels to the semantic content of the image. The pooling layers downsample data as it flows through the network. In a network with lots of layers, pooling layers should be used sparingly to avoid downsampling the data too early in the network.

```

% Convolutional layer parameters
filterSize = [5 5];
numFilters = 32;

middleLayers = [

% The first convolutional layer has a bank of 32 5x5x3 filters. A
% symmetric padding of 2 pixels is added to ensure that image borders
% are included in the processing. This is important to avoid
% information at the borders being washed away too early in the
% network.
convolution2dLayer(filterSize,numFilters,'Padding',2)

% Note that the third dimension of the filter can be omitted because it
% is automatically deduced based on the connectivity of the network. In
% this case because this layer follows the image layer, the third
% dimension must be 3 to match the number of channels in the input
% image.

% Next add the ReLU layer:
reluLayer()

% Follow it with a max pooling layer that has a 3x3 spatial pooling area
% and a stride of 2 pixels. This down-samples the data dimensions from
% 32x32 to 15x15.
maxPooling2dLayer(3,'Stride',2)

% Repeat the 3 core layers to complete the middle of the network.
convolution2dLayer(filterSize,numFilters,'Padding',2)
reluLayer()
maxPooling2dLayer(3, 'Stride',2)

convolution2dLayer(filterSize,2 * numFilters,'Padding',2)
reluLayer()
maxPooling2dLayer(3,'Stride',2)

]

```

```

middleLayers =
    9x1 Layer array with layers:

     1  ''  Convolution    32 5x5 convolutions with stride [1 1] and padding [2 2 2 2]
     2  ''  ReLU          ReLU
     3  ''  Max Pooling   3x3 max pooling with stride [2 2] and padding [0 0 0 0]
     4  ''  Convolution   32 5x5 convolutions with stride [1 1] and padding [2 2 2 2]
     5  ''  ReLU          ReLU
     6  ''  Max Pooling   3x3 max pooling with stride [2 2] and padding [0 0 0 0]
     7  ''  Convolution   64 5x5 convolutions with stride [1 1] and padding [2 2 2 2]
     8  ''  ReLU          ReLU
     9  ''  Max Pooling   3x3 max pooling with stride [2 2] and padding [0 0 0 0]
    
```

A deeper network may be created by repeating these 3 basic layers. However, the number of pooling layers should be reduced to avoid downsampling the data prematurely. Downsampling early in the network discards image information that is useful for learning.

The final layers of a CNN are typically composed of fully connected layers and a softmax loss layer.

```

finalLayers = [
    % Add a fully connected layer with 64 output neurons. The output size of
    % this layer will be an array with a length of 64.
    fullyConnectedLayer(64)

    % Add an ReLU non-linearity.
    reluLayer

    % Add the last fully connected layer. At this point, the network must
    % produce 10 signals that can be used to measure whether the input image
    % belongs to one category or another. This measurement is made using the
    % subsequent loss layers.
    fullyConnectedLayer(numImageCategories)

    % Add the softmax loss layer and classification layer. The final layers use
    % the output of the fully connected layer to compute the categorical
    % probability distribution over the image classes. During the training
    % process, all the network weights are tuned to minimize the loss over this
    % categorical distribution.
    softmaxLayer
    classificationLayer
]
    
```

```

finalLayers =
    5x1 Layer array with layers:

     1  ''  Fully Connected    64 fully connected layer
     2  ''  ReLU              ReLU
     3  ''  Fully Connected    10 fully connected layer
     4  ''  Softmax            softmax
     5  ''  Classification Output  crossentropyex
    
```

Combine the input, middle, and final layers.

```

layers = [
    inputLayer
    middleLayers
    finalLayers
]
    
```

```

layers =
  15x1 Layer array with layers:

   1  ''  Image Input           32x32x3 images with 'zerocenter' normalization
   2  ''  Convolution          32 5x5 convolutions with stride [1 1] and padding [2 2]
   3  ''  ReLU                  ReLU
   4  ''  Max Pooling          3x3 max pooling with stride [2 2] and padding [0 0 0 0]
   5  ''  Convolution          32 5x5 convolutions with stride [1 1] and padding [2 2]
   6  ''  ReLU                  ReLU
   7  ''  Max Pooling          3x3 max pooling with stride [2 2] and padding [0 0 0 0]
   8  ''  Convolution          64 5x5 convolutions with stride [1 1] and padding [2 2]
   9  ''  ReLU                  ReLU
  10  ''  Max Pooling          3x3 max pooling with stride [2 2] and padding [0 0 0 0]
  11  ''  Fully Connected      64 fully connected layer
  12  ''  ReLU                  ReLU
  13  ''  Fully Connected      10 fully connected layer
  14  ''  Softmax              softmax
  15  ''  Classification Output crossentropyex

```

Initialize the first convolutional layer weights using normally distributed random numbers with standard deviation of 0.0001. This helps improve the convergence of training.

```
layers(2).Weights = 0.0001 * randn([filterSize numChannels numFilters]);
```

Train CNN Using CIFAR-10 Data

Now that the network architecture is defined, it can be trained using the CIFAR-10 training data. First, set up the network training algorithm using the `trainingOptions` (Deep Learning Toolbox) function. The network training algorithm uses Stochastic Gradient Descent with Momentum (SGDM) with an initial learning rate of 0.001. During training, the initial learning rate is reduced every 8 epochs (1 epoch is defined as one complete pass through the entire training data set). The training algorithm is run for 40 epochs.

Note that the training algorithm uses a mini-batch size of 128 images. If using a GPU for training, this size may need to be lowered due to memory constraints on the GPU.

```

% Set the network training options
opts = trainingOptions('sgdm', ...
    'Momentum', 0.9, ...
    'InitialLearnRate', 0.001, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 8, ...
    'L2Regularization', 0.004, ...
    'MaxEpochs', 40, ...
    'MiniBatchSize', 128, ...
    'Verbose', true);

```

Train the network using the `trainNetwork` (Deep Learning Toolbox) function. This is a computationally intensive process that takes 20-30 minutes to complete. To save time while running this example, a pretrained network is loaded from disk. If you wish to train the network yourself, set the `doTraining` variable shown below to true.

Note that a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

```

% A trained network is loaded from disk to save time when running the
% example. Set this flag to true to train the network.

```

```
doTraining = false;

if doTraining
    % Train a network.
    cifar10Net = trainNetwork(trainingImages, trainingLabels, layers, opts);
else
    % Load pre-trained detector for the example.
    load('rcnnStopSigns.mat', 'cifar10Net')
end
```

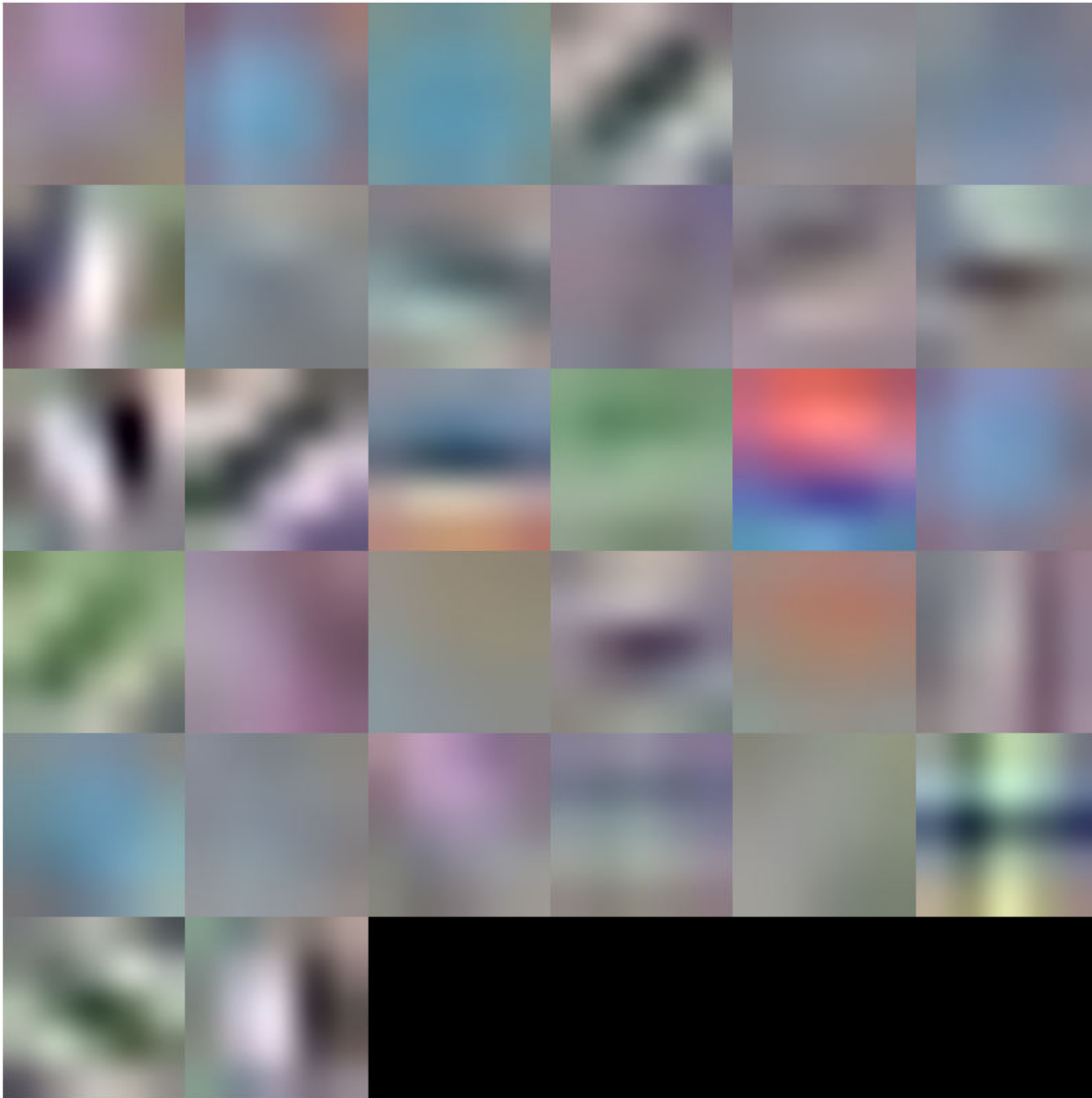
Validate CIFAR-10 Network Training

After the network is trained, it should be validated to ensure that training was successful. First, a quick visualization of the first convolutional layer's filter weights can help identify any immediate issues with training.

```
% Extract the first convolutional layer weights
w = cifar10Net.Layers(2).Weights;

% rescale the weights to the range [0, 1] for better visualization
w = rescale(w);

figure
montage(w)
```

The first layer weights should have some well defined structure. If the weights still look random, then that is an indication that the network may require additional training. In this case, as shown above, the first layer filters have learned edge-like features from the CIFAR-10 training data.

To completely validate the training results, use the CIFAR-10 test data to measure the classification accuracy of the network. A low accuracy score indicates additional training or additional training data is required. The goal of this example is not necessarily to achieve 100% accuracy on the test set, but to sufficiently train a network for use in training an object detector.

```
% Run the network on the test set.  
YTest = classify(cifar10Net, testImages);
```

```
% Calculate the accuracy.
accuracy = sum(YTest == testLabels)/numel(testLabels)

accuracy = 0.7456
```

Further training will improve the accuracy, but that is not necessary for the purpose of training the R-CNN object detector.

Load Training Data

Now that the network is working well for the CIFAR-10 classification task, the transfer learning approach can be used to fine-tune the network for stop sign detection.

Start by loading the ground truth data for stop signs.

```
% Load the ground truth data
data = load('stopSignsAndCars.mat', 'stopSignsAndCars');
stopSignsAndCars = data.stopSignsAndCars;

% Update the path to the image files to match the local file system
visiondata = fullfile(toolboxdir('vision'),'visiondata');
stopSignsAndCars.imageFilename = fullfile(visiondata, stopSignsAndCars.imageFilename);

% Display a summary of the ground truth data
summary(stopSignsAndCars)

Variables:
    imageFilename: 41x1 cell array of character vectors
         stopSign: 41x1 cell
         carRear: 41x1 cell
         carFront: 41x1 cell
```

The training data is contained within a table that contains the image filename and ROI labels for stop signs, car fronts, and rears. Each ROI label is a bounding box around objects of interest within an image. For training the stop sign detector, only the stop sign ROI labels are needed. The ROI labels for car front and rear must be removed:

```
% Only keep the image file names and the stop sign ROI labels
stopSigns = stopSignsAndCars(:, {'imageFilename','stopSign'});

% Display one training image and the ground truth bounding boxes
I = imread(stopSigns.imageFilename{1});
I = insertObjectAnnotation(I, 'Rectangle', stopSigns.stopSign{1}, 'stop sign', 'LineWidth', 8);

figure
imshow(I)
```



Note that there are only 41 training images within this data set. Training an R-CNN object detector from scratch using only 41 images is not practical and would not produce a reliable stop sign detector. Because the stop sign detector is trained by fine-tuning a network that has been pre-trained on a larger dataset (CIFAR-10 has 50,000 training images), using a much smaller dataset is feasible.

Train R-CNN Stop Sign Detector

Finally, train the R-CNN object detector using `trainRCNNObjectDetector`. The input to this function is the ground truth table which contains labeled stop sign images, the pre-trained CIFAR-10 network, and the training options. The training function automatically modifies the original CIFAR-10 network, which classified images into 10 categories, into a network that can classify images into 2 classes: stop signs and a generic background class.

During training, the input network weights are fine-tuned using image patches extracted from the ground truth data. The 'PositiveOverlapRange' and 'NegativeOverlapRange' parameters control which image patches are used for training. Positive training samples are those that overlap with the ground truth boxes by 0.5 to 1.0, as measured by the bounding box intersection over union metric. Negative training samples are those that overlap by 0 to 0.3. The best values for these parameters should be chosen by testing the trained detector on a validation set.

For R-CNN training, **the use of a parallel pool of MATLAB workers is highly recommended to reduce training time.** `trainRCNNObjectDetector` automatically creates and uses a parallel pool based on your parallel preference settings. Ensure that the use of the parallel pool is enabled prior to training.

To save time while running this example, a pretrained network is loaded from disk. If you wish to train the network yourself, set the `doTraining` variable shown below to true.

Note that a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

```
% A trained detector is loaded from disk to save time when running the
% example. Set this flag to true to train the detector.
```

```
doTraining = false;

if doTraining

    % Set training options
    options = trainingOptions('sgdm', ...
        'MiniBatchSize', 128, ...
        'InitialLearnRate', 1e-3, ...
        'LearnRateSchedule', 'piecewise', ...
        'LearnRateDropFactor', 0.1, ...
        'LearnRateDropPeriod', 100, ...
        'MaxEpochs', 100, ...
        'Verbose', true);

    % Train an R-CNN object detector. This will take several minutes.
    rcnn = trainRCNNObjectDetector(stopSigns, cifar10Net, options, ...
        'NegativeOverlapRange', [0 0.3], 'PositiveOverlapRange',[0.5 1])
else
    % Load pre-trained network for the example.
    load('rcnnStopSigns.mat','rcnn')
end
```

Test R-CNN Stop Sign Detector

The R-CNN object detector can now be used to detect stop signs in images. Try it out on a test image:

```
% Read test image
testImage = imread('stopSignTest.jpg');

% Detect stop signs
[bboxes,score,label] = detect(rcnn,testImage,'MiniBatchSize',128)

bboxes = 1x4
    419    147    31    20

score = single
    0.9955

label = categorical categorical
    stopSign
```

The R-CNN object `detect` method returns the object bounding boxes, a detection score, and a class label for each detection. The labels are useful when detecting multiple objects, e.g. stop, yield, or speed limit signs. The scores, which range between 0 and 1, indicate the confidence in the detection and can be used to ignore low scoring detections.

```
% Display the detection results
[score, idx] = max(score);

bbox = bboxes(idx, :);
```

```

annotation = sprintf('%s: (Confidence = %f)', label(idx), score);
outputImage = insertObjectAnnotation(testImage, 'rectangle', bbox, annotation);

figure
imshow(outputImage)

```



Debugging Tips

The network used within the R-CNN detector can also be used to process the entire test image. By directly processing the entire image, which is larger than the network's input size, a 2-D heat-map of classification scores can be generated. This is a useful debugging tool because it helps identify items in the image that are confusing the network, and may help provide insight into improving training.

```

% The trained network is stored within the R-CNN detector
rcnn.Network

```

```

ans =
  SeriesNetwork with properties:

    Layers: [15x1 nnet.cnn.layer.Layer]

```

Extract the activations (Deep Learning Toolbox) from the softmax layer, which is the 14th layer in the network. These are the classification scores produced by the network as it scans the image.

```

featureMap = activations(rcnn.Network, testImage, 14);

```

```

% The softmax activations are stored in a 3-D array.
size(featureMap)

```

```
ans = 1×3
      43    78    2
```

The 3rd dimension in featureMap corresponds to the object classes.

```
rcnn.ClassNames
```

```
ans = 2×1 cell
      {'stopSign' }
      {'Background'}
```

The stop sign feature map is stored in the first channel.

```
stopSignMap = featureMap(:, :, 1);
```

The size of the activations output is smaller than the input image due to the downsampling operations in the network. To generate a nicer visualization, resize stopSignMap to the size of the input image. This is a very crude approximation that maps activations to image pixels and should only be used for illustrative purposes.

```
% Resize stopSignMap for visualization
[height, width, ~] = size(testImage);
stopSignMap = imresize(stopSignMap, [height, width]);

% Visualize the feature map superimposed on the test image.
featureMapOnImage = imfuse(testImage, stopSignMap);

figure
imshow(featureMapOnImage)
```



The stop sign in the test image corresponds nicely with the largest peak in the network activations. This helps verify that the CNN used within the R-CNN detector has effectively learned to identify stop signs. Had there been other peaks, this may indicate that the training requires additional negative data to help prevent false positives. If that's the case, then you can increase 'MaxEpochs' in the trainingOptions and re-train.

Summary

This example showed how to train an R-CNN stop sign object detector using a network trained with CIFAR-10 data. Similar steps may be followed to train other object detectors using deep learning.

References

- [1] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Columbus, OH, June 2014, pp. 580-587.
- [2] Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database." *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition*. Miami, FL, June 2009, pp. 248-255.
- [3] Krizhevsky, A., and G. Hinton. "Learning multiple layers of features from tiny images." Master's Thesis. University of Toronto, Toronto, Canada, 2009.
- [4] <https://code.google.com/p/cuda-convnet/>

See Also

`activations` | `classify` | `detect` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `rcnnObjectDetector` | `trainFastRCNNObjectDetector` |

`trainFasterRCNNObjectDetector | trainNetwork | trainRCNNObjectDetector | trainingOptions`

More About

- “Object Detection Using Faster R-CNN Deep Learning” on page 3-197
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Object Detection Using Faster R-CNN Deep Learning

This example shows how to train a Faster R-CNN (regions with convolutional neural networks) object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several deep learning techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a Faster R-CNN vehicle detector using the `trainFasterRCNNObjectDetector` function. For more information, see “Object Detection using Deep Learning”.

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTrainingAndEval` variable to true.

```
doTrainingAndEval = false;
if ~doTrainingAndEval && ~exist('fasterRCNNResNet50EndToEndVehicleExample.mat','file')
    disp('Downloading pretrained detector (118 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/fasterRCNNResNet50EndToE...';
    websave('fasterRCNNResNet50EndToEndVehicleExample.mat',pretrainedURL);
end
```

Load Data Set

This example uses a small labeled dataset that contains 295 images. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the Faster R-CNN training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip('vehicleDatasetImages.zip');
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

Split the dataset into training, validation, and test sets. Select 60% of the data for training, 10% for validation, and the rest for testing the trained detector.

```
rng(0)
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));

trainingIdx = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);

validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);

testIdx = validationIdx(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIdx),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingDataTbl{:, 'imageFilename'});  
blsTrain = boxLabelDatastore(trainingDataTbl{:, 'vehicle'});  
  
imdsValidation = imageDatastore(validationDataTbl{:, 'imageFilename'});  
blsValidation = boxLabelDatastore(validationDataTbl{:, 'vehicle'});  
  
imdsTest = imageDatastore(testDataTbl{:, 'imageFilename'});  
blsTest = boxLabelDatastore(testDataTbl{:, 'vehicle'});
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,blsTrain);  
validationData = combine(imdsValidation,blsValidation);  
testData = combine(imdsTest,blsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);  
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I, 'Rectangle', bbox);  
annotatedImage = imresize(annotatedImage, 2);  
figure  
imshow(annotatedImage)
```



Create Faster R-CNN Detection Network

A Faster R-CNN object detection network is composed of a feature extraction network followed by two subnetworks. The feature extraction network is typically a pretrained CNN, such as ResNet-50 or Inception v3. The first subnetwork following the feature extraction network is a region proposal network (RPN) trained to generate object proposals - areas in the image where objects are likely to exist. The second subnetwork is trained to predict the actual class of each object proposal.

The feature extraction network is typically a pretrained CNN (for details, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox)). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18, depending on your application requirements.

Use `fasterRCNNLayers` to create a Faster R-CNN network automatically given a pretrained feature extraction network. `fasterRCNNLayers` requires you to specify several inputs that parameterize a Faster R-CNN network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of `[224 224 3]`, which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes.

```
preprocessedTrainingData = transform(trainingData, @(data)preprocessData(data,inputSize));
numAnchors = 3;
anchorBoxes = estimateAnchorBoxes(preprocessedTrainingData,numAnchors)
```

```
anchorBoxes = 3×2
```

```
    136    119
     55     48
    157    128
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” on page 3-146 (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” on page 14-21.

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select 'activation_40_relu' as the feature extraction layer. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis. You can use `analyzeNetwork` to find the names of other potential feature extraction layers within a network.

```
featureLayer = 'activation_40_relu';
```

Define the number of classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the Faster R-CNN object detection network.

```
lgraph = fasterRCNNLayers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer);
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the Faster R-CNN network architecture, use Deep Network Designer to design the Faster R-CNN detection network manually. For more information, see “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 14-30.

Data Augmentation

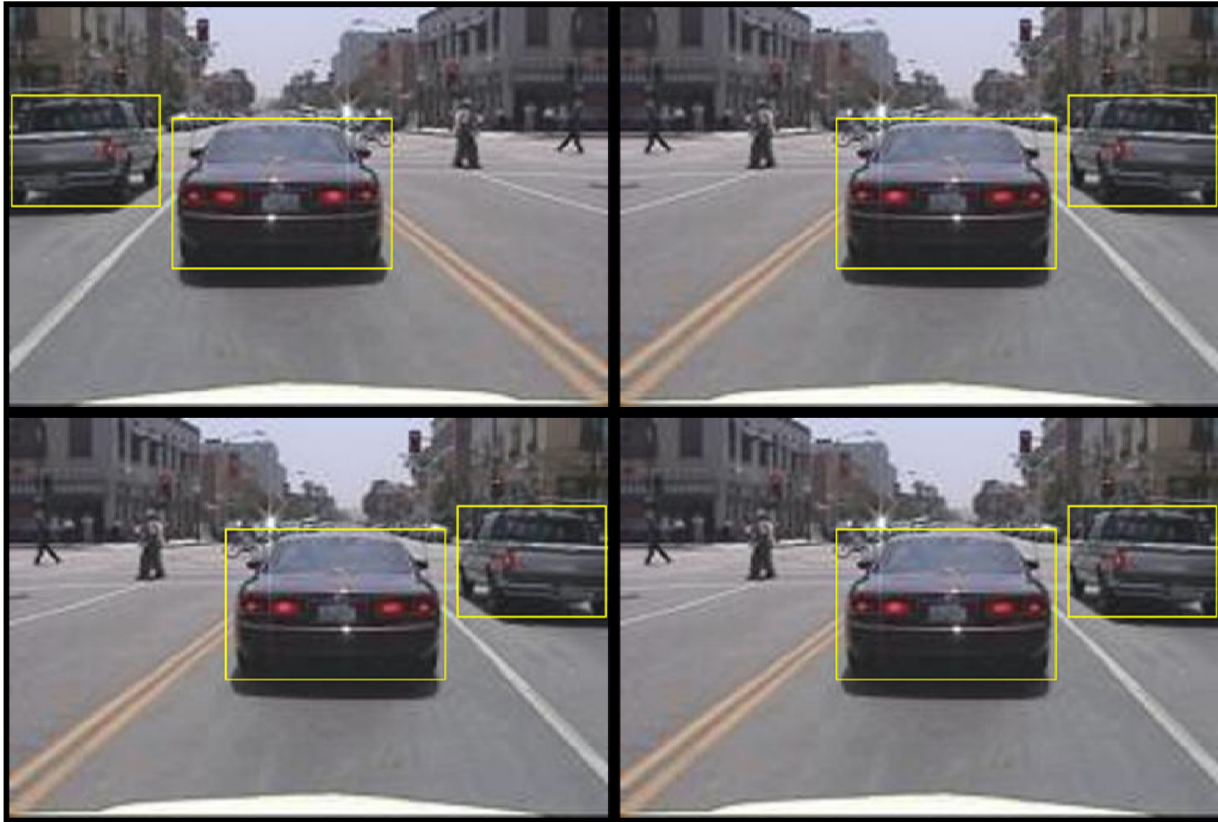
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to test and validation data. Ideally, test and validation data are representative of the original data and are left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
augmentedData = cell(4,1);  
for k = 1:4  
    data = read(augmentedTrainingData);  
    augmentedData{k} = insertShape(data{1}, 'Rectangle', data{2});  
    reset(augmentedTrainingData);  
end  
figure  
montage(augmentedData, 'BorderSize', 10)
```



Preprocess Training Data

Preprocess the augmented training data, and the validation data to prepare for training.

```
trainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));  
validationData = transform(validationData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed data.

```
data = read(trainingData);
```

Display the image and box bounding boxes.

```
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I,'Rectangle',bbox);  
annotatedImage = imresize(annotatedImage,2);  
figure  
imshow(annotatedImage)
```



Train Faster R-CNN

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm',...
    'MaxEpochs',10,...
    'MiniBatchSize',2,...
    'InitialLearnRate',1e-3,...
    'CheckpointPath',tempdir,...
    'ValidationData',validationData);
```

Use `trainFasterRCNNObjectDetector` to train Faster R-CNN object detector if `doTrainingAndEval` is true. Otherwise, load the pretrained network.

```
if doTrainingAndEval
    % Train the Faster R-CNN detector.
```

```

% * Adjust NegativeOverlapRange and PositiveOverlapRange to ensure
% that training samples tightly overlap with ground truth.
[detector, info] = trainFasterRCNNObjectDetector(trainingData,lgraph,options, ...
    'NegativeOverlapRange',[0 0.3], ...
    'PositiveOverlapRange',[0.6 1]);
else
% Load pretrained detector for the example.
pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
detector = pretrained.detector;
end

```

This example was verified on an Nvidia(TM) Titan X GPU with 12 GB of memory. Training the network took approximately 20 minutes. The training time varies depending on the hardware you use.

As a quick check, run the detector on one test image. Make sure you resize the image to the same size as the training images.

```

I = imread(testDataTbl.imageFilename{1});
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);

```

Display the results.

```

I = insertObjectAnnotation(I, 'rectangle', bboxes, scores);
figure
imshow(I)

```



Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the

detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data.

```
testData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

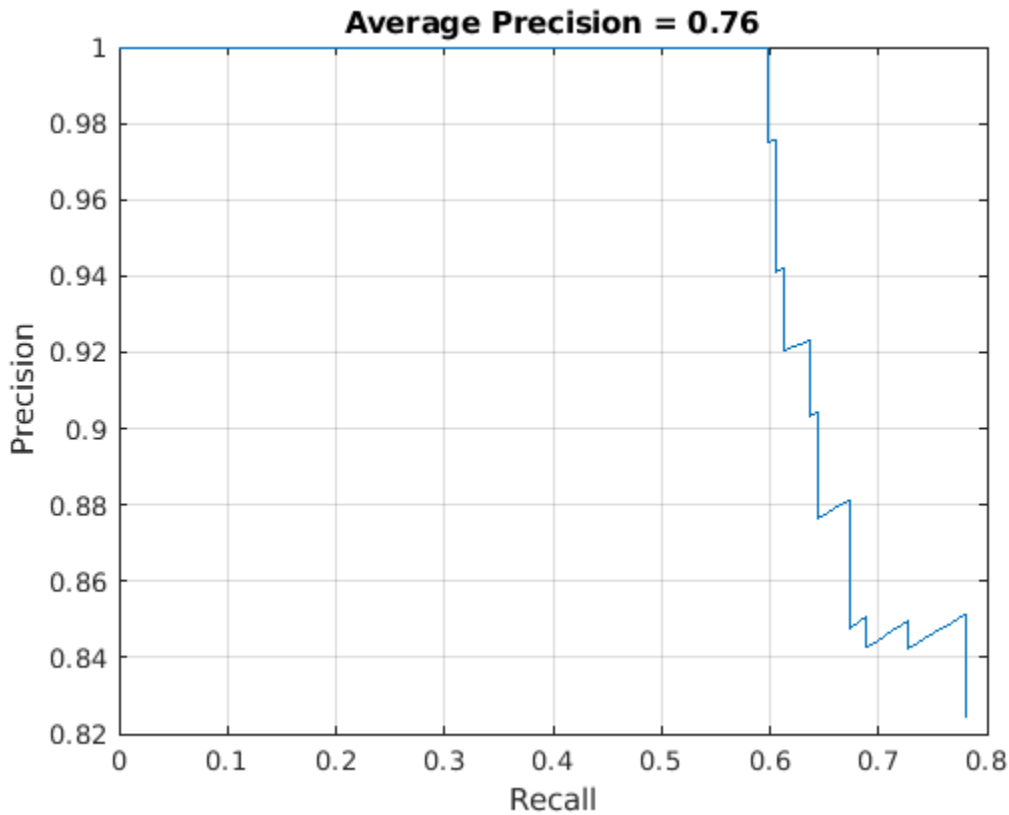
```
if doTrainingAndEval
    detectionResults = detect(detector,testData,'MinibatchSize',4);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detectionResults = pretrained.detectionResults;
end
```

Evaluate the object detector using the average precision metric.

```
[ap, recall, precision] = evaluateDetectionPrecision(detectionResults,testData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```

Supporting Functions

```
function data = augmentData(data)
% Randomly flip images and bounding boxes horizontally.
tform = randomAffine2d('XReflection',true);
rout = affineOutputView(size(data{1}),tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);
data{2} = bboxwarp(data{2},tform,rout);
end
```

```
function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Ren, S., K. He, R. Gershick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *IEEE Transactions of Pattern Analysis and Machine Intelligence*. Vol. 39, Issue 6, June 2017, pp. 1137-1149.

[2] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Columbus, OH, June 2014, pp. 580-587.

[3] Girshick, R. "Fast R-CNN." *Proceedings of the 2015 IEEE International Conference on Computer Vision*. Santiago, Chile, Dec. 2015, pp. 1440-1448.

[4] Zitnick, C. L., and P. Dollar. "Edge Boxes: Locating Object Proposals from Edges." *European Conference on Computer Vision*. Zurich, Switzerland, Sept. 2014, pp. 391-405.

[5] Uijlings, J. R. R., K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. "Selective Search for Object Recognition." *International Journal of Computer Vision*. Vol. 104, Number 2, Sept. 2013, pp. 154-171.

See Also

`detect` | `evaluateDetectionMissRate` | `evaluateDetectionPrecision` |
`fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `insertObjectAnnotation` |
`rcnnObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` |
`trainNetwork` | `trainRCNNObjectDetector` | `trainingOptions`

More About

- "Train Object Detector Using R-CNN Deep Learning" on page 3-183
- "Deep Learning in MATLAB" (Deep Learning Toolbox)

Train Classification Network to Classify Object in 3-D Point Cloud

This example demonstrates the approach outlined in [1 on page 3-0] in which point cloud data is preprocessed into a voxelized encoding and then used directly with a simple 3-D convolutional neural network architecture to perform object classification. In more recent approaches such as [2 on page 3-0], encodings of point cloud data can be more complicated and can be learned encodings that are trained end-to-end along with a network performing a classification/object detection/segmentation task. However, the general pattern of moving from irregular unordered points to a gridded structure that can be fed into convnets remains similar in all of these approaches.

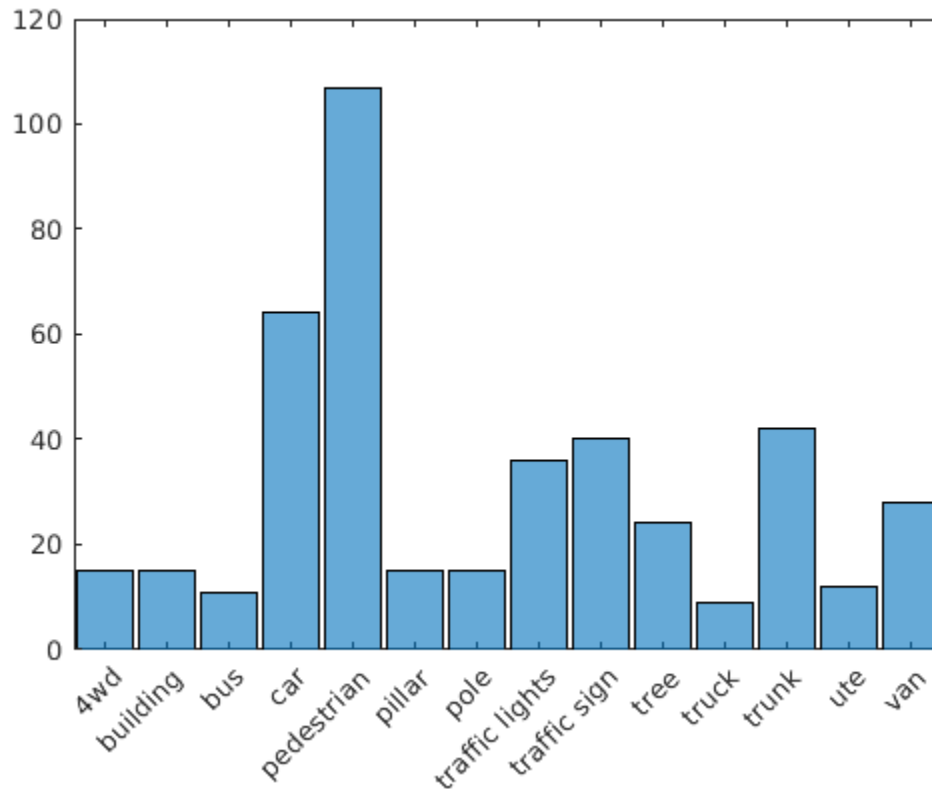
Import and Analyze Data

In this example, we work with the Sydney Urban Objects Dataset. In this example, we use folds 1-3 from the data as the training set and fold 4 as the validation set.

```
dataPath = downloadSydneyUrbanObjects(tempdir);
dsTrain = sydneyUrbanObjectsClassificationDatastore(dataPath,[1 2 3]);
dsVal = sydneyUrbanObjectsClassificationDatastore(dataPath,4);
```

Analyze the training set to understand the labels present in the data and the overall distribution of labels.

```
dsLabels = transform(dsTrain,@(data) data{2});
labels = readall(dsLabels);
figure
histogram(labels)
```



From the histogram, it is apparent that there is a class imbalance issue in the training data in which certain object classes like Car and Pedestrian are much more common than less frequent classes like Ute.

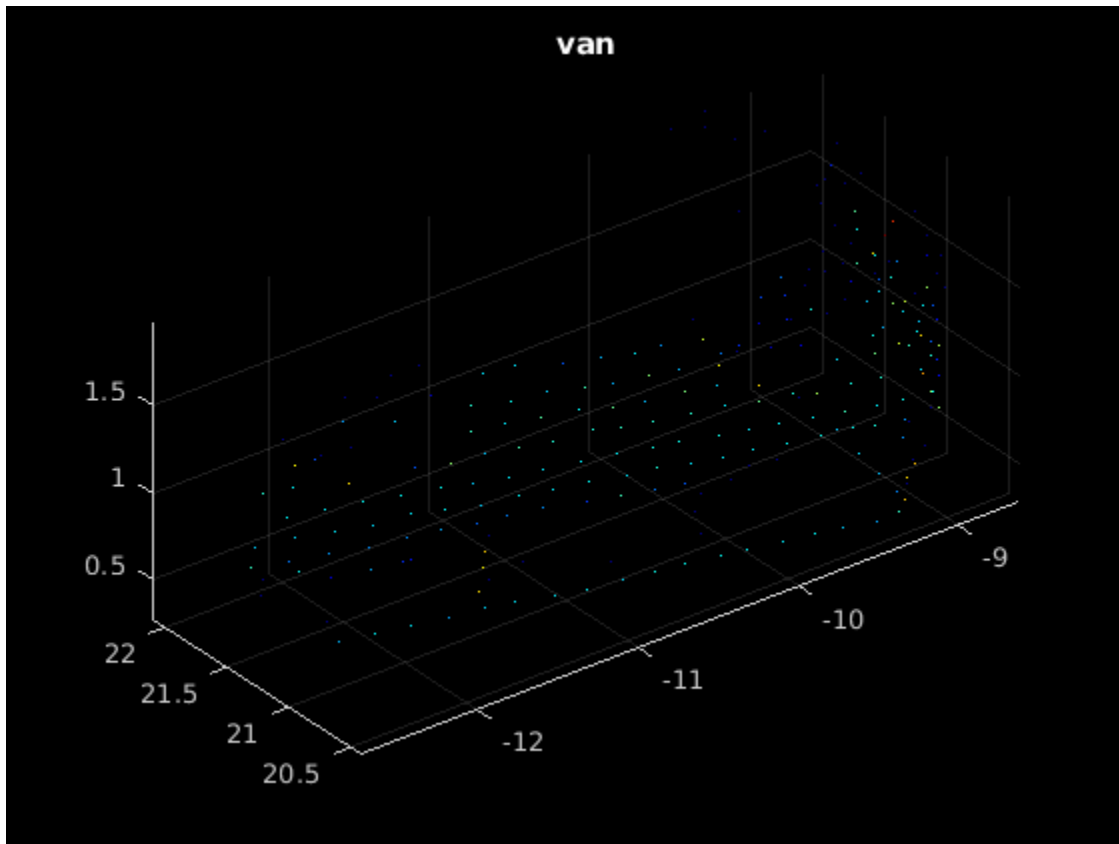
Data augmentation pipeline

To avoid overfitting and add robustness to a classifier, some amount of randomized data augmentation is generally a good idea when training a network. The functions `randomAffine2d` and `pctransform` make it easy to define randomized affine transformations on point cloud data. We additionally add some randomized per-point jitter to each point in every point cloud. The function `augmentPointCloudData` is included in the supporting functions section below.

```
dsTrain = transform(dsTrain,@augmentPointCloudData);
```

Verify that augmentation of point cloud data looks reasonable.

```
dataOut = preview(dsTrain);
figure
pcshow(dataOut{1});
title(dataOut{2});
```

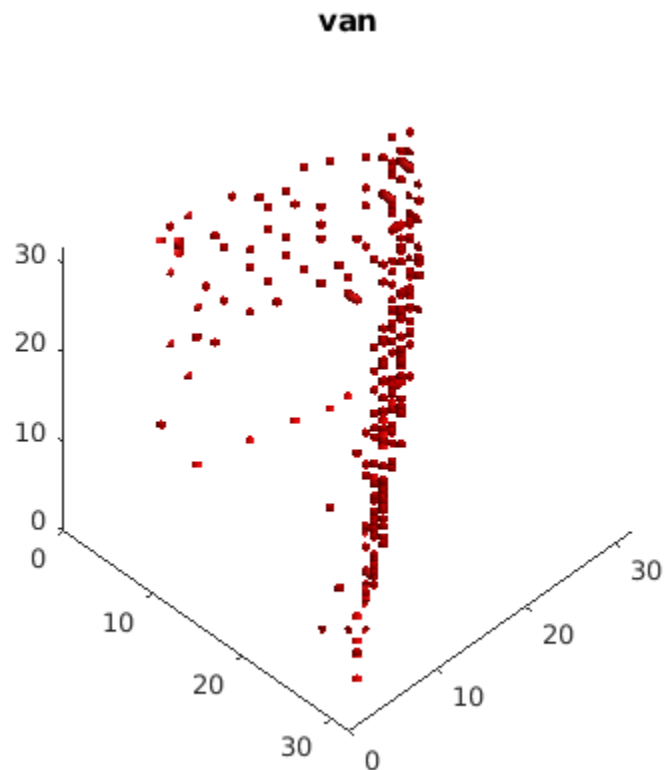


We next add a simple voxelization transform to each input point cloud as discussed in the previous example, to transform our input point cloud into a pseudo-image that can be used with a convolutional neural network. Use a simple occupancy grid.

```
dsTrain = transform(dsTrain,@formOccupancyGrid);
dsVal = transform(dsVal,@formOccupancyGrid);
```

Examine a sample of the final voxelized volume that we will feed into the network to verify that voxelization is working correctly.

```
data = preview(dsTrain);
figure
p = patch(isosurface(data{1},0.5));
p.FaceColor = 'red';
p.EdgeColor = 'none';
daspect([1 1 1])
view(45,45)
camlight;
lighting phong
title(data{2});
```

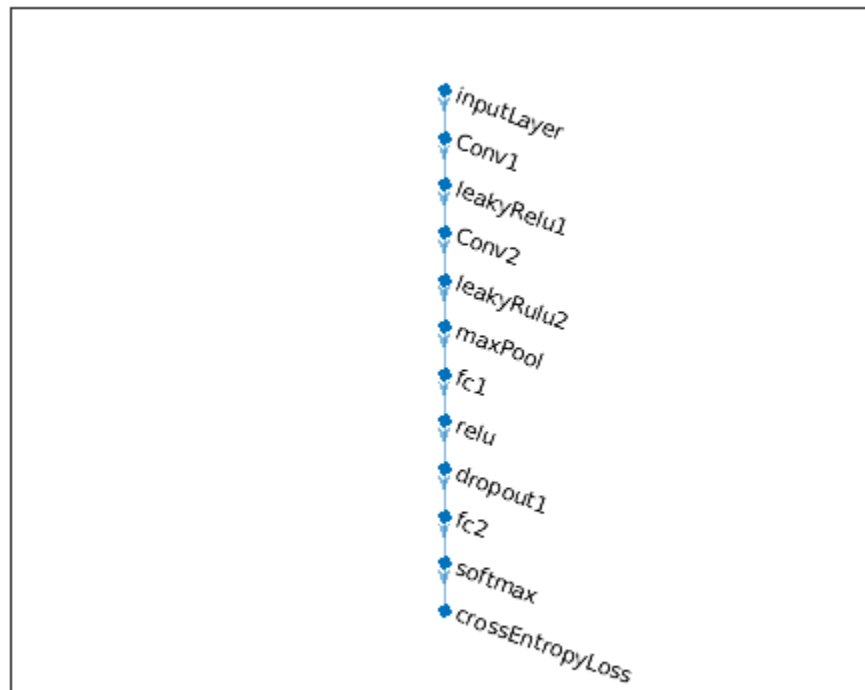


Define network architecture

In this example, we use a simple 3-D classification architecture as described in [1].

```
layers = [image3dInputLayer([32 32 32], 'Name', 'inputLayer', 'Normalization', 'none'), ...
convolution3dLayer(5, 32, 'Stride', 2, 'Name', 'Conv1'), ...
leakyReluLayer(0.1, 'Name', 'leakyRelu1'), ...
convolution3dLayer(3, 32, 'Stride', 1, 'Name', 'Conv2'), ...
leakyReluLayer(0.1, 'Name', 'leakyRulu2'), ...
maxPooling3dLayer(2, 'Stride', 2, 'Name', 'maxPool'), ...
fullyConnectedLayer(128, 'Name', 'fc1'), ...
reluLayer('Name', 'relu'), ...
dropoutLayer(0.5, 'Name', 'dropout1'), ...
fullyConnectedLayer(14, 'Name', 'fc2'), ...
softmaxLayer('Name', 'softmax'), ...
classificationLayer('Name', 'crossEntropyLoss')];

voxnet = layerGraph(layers);
figure
plot(voxnet);
```



Setup training options

Use stochastic gradient descent with momentum with a piecewise adjustment to the learning rate schedule. This example was run on a TitanX GPU, for GPUs with less memory, it may be necessary to reduce the batch size. Though 3D convnets have an advantage of conceptual simplicity, they have the drawback of large amounts of memory usage at training time.

```

miniBatchSize = 32;
dsLength = length(dsTrain.UnderlyingDatastore.Files);
iterationsPerEpoch = floor(dsLength/miniBatchSize);
dropPeriod = floor(8000/iterationsPerEpoch);

options = trainingOptions('sgdm','InitialLearnRate',0.01,'MiniBatchSize',miniBatchSize,...
    'LearnRateSchedule','Piecewise',...
    'LearnRateDropPeriod',dropPeriod,...
    'ValidationData',dsVal,'MaxEpochs',60,...
    'DispatchInBackground',false,...
    'Shuffle','never');
  
```

Train network

```
voxnet = trainNetwork(dsTrain,voxnet,options);
```

Training on single GPU.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Validation Accuracy	Mini-batch Loss	Validat Loss
=====						

1	1	00:00:03	9.38%	20.65%	2.6408	2.6408
4	50	00:00:25	31.25%	29.03%	2.2892	2.2892
8	100	00:00:45	37.50%	37.42%	1.9256	2.0000
12	150	00:01:05	53.12%	47.10%	1.6398	1.6398
16	200	00:01:24	43.75%	55.48%	1.9551	1.9551
20	250	00:01:44	40.62%	61.29%	1.7413	1.7413
24	300	00:02:04	50.00%	60.00%	1.4652	1.4652
27	350	00:02:23	43.75%	64.52%	1.5017	1.5017
31	400	00:02:42	53.12%	69.03%	1.2488	1.2488
35	450	00:03:02	50.00%	69.03%	1.3160	1.3160
39	500	00:03:23	59.38%	69.03%	1.1753	1.1753
43	550	00:03:44	56.25%	65.81%	1.1546	1.1546
47	600	00:04:03	68.75%	65.81%	0.9808	1.0000
50	650	00:04:22	65.62%	69.68%	1.1245	1.1245
54	700	00:04:42	62.50%	65.16%	1.2860	1.2860
58	750	00:05:01	59.38%	68.39%	1.2466	1.2466
60	780	00:05:13	56.25%	64.52%	1.1676	1.1676

Evaluate network

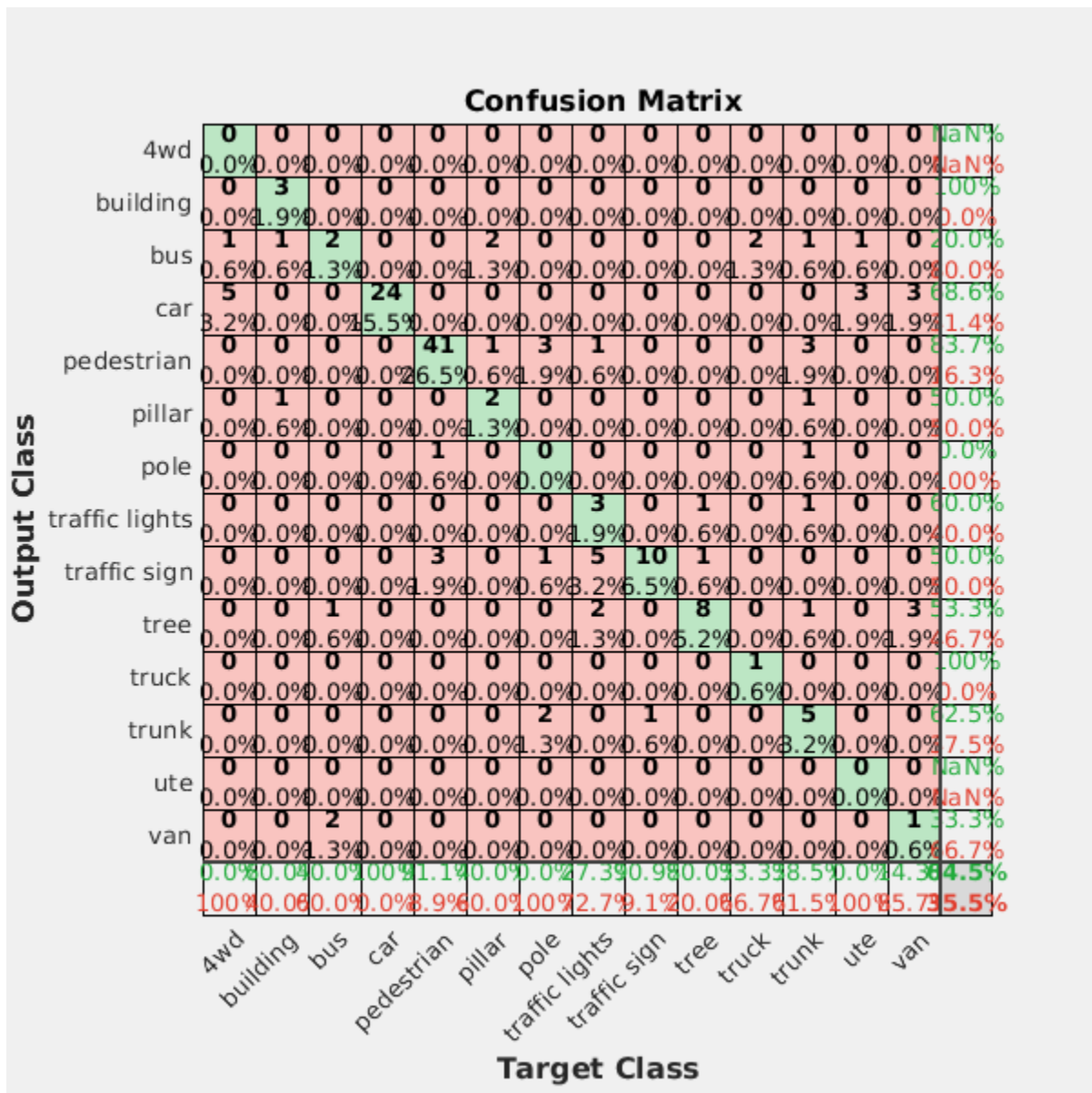
Following the structure of [1 on page 3-0], this example only forms a training and validation set from Sydney Urban Objects. Evaluate the performance of the trained network using the validation, since it was not used to train the network.

```
valLabelSet = transform(dsVal,@(data) data{2});
valLabels = readall(valLabelSet);
outputLabels = classify(voxnet,dsVal);
accuracy = nnz(outputLabels == valLabels) / numel(outputLabels);
disp(accuracy)
```

```
0.6452
```

View the confusion matrix to study the accuracy across the various label categories

```
figure
plotconfusion(valLabels,outputLabels)
```

The label imbalance noted in the training set is an issue in the classification accuracy. The confusion chart illustrates higher precision and recall for pedestrian, the most common class, than for less common classes like van. Since the purpose of this example is to demonstrate a basic classification network training approach with point cloud data, possible next steps that could be taken to improve classification performance such as resampling the training set or achieving better label balance or using a loss function more robust to label imbalance (e.g. weighted cross-entropy) will not be explored.

References

1) *Voxnet: A 3d convolutional neural network for real-time object recognition*, Daniel Maturana, Sebastian Scherer, 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)

2) *PointPillars: Fast Encoders for Object Detection from Point Clouds*, Alex H. Lang, Sourabh Vora, et al, CVPR 2019

3) *Sydney Urban Objects Dataset*, Alastair Quadros, James Underwood, Bertrand Douillard, Sydney Urban Objects

Supporting Functions

```
function datasetPath = downloadSydneyUrbanObjects(dataLoc)

if nargin == 0
    dataLoc = pwd();
end

dataLoc = string(dataLoc);

url = "http://www.acfr.usyd.edu.au/papers/data/";
name = "sydney-urban-objects-dataset.tar.gz";

if ~exist(fullfile(dataLoc,'sydney-urban-objects-dataset'),'dir')
    disp('Downloading Sydney Urban Objects Dataset...');
    untar(fullfile(url,name),dataLoc);
end

datasetPath = dataLoc.append('sydney-urban-objects-dataset');

end

function ds = sydneyUrbanObjectsClassificationDatastore(datapath,folds)
% sydneyUrbanObjectsClassificationDatastore Datastore with point clouds and
% associated categorical labels for Sydney Urban Objects dataset.
%
% ds = sydneyUrbanObjectsDatastore(datapath) constructs a datastore that
% represents point clouds and associated categories for the Sydney Urban
% Objects dataset. The input, datapath, is a string or char array which
% represents the path to the root directory of the Sydney Urban Objects
% Dataset.
%
% ds = sydneyUrbanObjectsDatastore(___,folds) optionally allows
% specification of desired folds that you wish to be included in the
% output ds. For example, [1 2 4] specifies that you want the first,
% second, and fourth folds of the Dataset. Default: [1 2 3 4].

if nargin < 2
    folds = 1:4;
end

datapath = string(datapath);
path = fullfile(datapath,'objects',filesep);

% For now, include all folds in Datastore
foldNames{1} = importdata(fullfile(datapath,'folds','fold0.txt'));
foldNames{2} = importdata(fullfile(datapath,'folds','fold1.txt'));
foldNames{3} = importdata(fullfile(datapath,'folds','fold2.txt'));
foldNames{4} = importdata(fullfile(datapath,'folds','fold3.txt'));
names = foldNames(folds);
names = vertcat(names{:});
```

```

fullfilenames = append(path,names);
ds = fileDatastore(fullfilenames, 'ReadFcn',@extractTrainingData, 'FileExtensions', '.bin');

% Shuffle
ds.Files = ds.Files(randperm(length(ds.Files)));

end

function dataOut = extractTrainingData(fname)

[pointData,intensity] = readbin(fname);

[~,name] = fileparts(fname);
name = string(name);
name = extractBefore(name, '.');
name = replace(name, '_', ' ');

labelNames = ["4wd", "building", "bus", "car", "pedestrian", "pillar", ...
              "pole", "traffic lights", "traffic sign", "tree", "truck", "trunk", "ute", "van"];

label = categorical(name, labelNames);

dataOut = {pointCloud(pointData, 'Intensity', intensity), label};

end

function [pointData,intensity] = readbin(fname)
% readbin Read point and intensity data from Sydney Urban Object binary
% files.

% names = ['t', 'intensity', 'id', ...
%         'x', 'y', 'z', ...
%         'azimuth', 'range', 'pid']
%
% formats = ['int64', 'uint8', 'uint8', ...
%           'float32', 'float32', 'float32', ...
%           'float32', 'float32', 'int32']

fid = fopen(fname, 'r');
c = onCleanup(@() fclose(fid));

fseek(fid,10,-1); % Move to the first X point location 10 bytes from beginning
X = fread(fid,inf, 'single', 30);
fseek(fid,14,-1);
Y = fread(fid,inf, 'single', 30);
fseek(fid,18,-1);
Z = fread(fid,inf, 'single', 30);

fseek(fid,8,-1);
intensity = fread(fid,inf, 'uint8', 33);

pointData = [X,Y,Z];

end

function dataOut = formOccupancyGrid(data)

grid = pccbin(data{1}, [32 32 32]);

```

```
occupancyGrid = zeros(size(grid),'single');
for ii = 1:numel(grid)
    occupancyGrid(ii) = ~isempty(grid{ii});
end
label = data{2};
dataOut = {occupancyGrid,label};

end

function dataOut = augmentPointCloudData(data)

ptCloud = data{1};
label = data{2};

% Apply randomized rotation about Z axis.
tform = randomAffine3d('Rotation',@() deal([0 0 1],360*rand), 'Scale',[0.98,1.02], 'XReflection', t
ptCloud = pctransform(ptCloud,tform);

% Apply jitter to each point in point cloud
amountOfJitter = 0.01;
numPoints = size(ptCloud.Location,1);
D = zeros(size(ptCloud.Location),'like',ptCloud.Location);
D(:,1) = diff(ptCloud.XLimits)*rand(numPoints,1);
D(:,2) = diff(ptCloud.YLimits)*rand(numPoints,1);
D(:,3) = diff(ptCloud.ZLimits)*rand(numPoints,1);
D = amountOfJitter.*D;
ptCloud = pctransform(ptCloud,D);

dataOut = {ptCloud,label};

end
```

Estimate Body Pose Using Deep Learning

This example shows how to estimate the body pose of one or more people using the OpenPose algorithm and a pretrained network.

The goal of body pose estimation is to identify the location of people in an image and the orientation of their body parts. When multiple people are present in a scene, pose estimation can be more difficult because of occlusion, body contact, and proximity of similar body parts.

There are two strategies to estimating body pose. A top-down strategy first identifies individual people using object detection and then estimates the pose of each person. A bottom-up strategy first identifies body parts in an image, such as noses and left elbows, and then assembles individuals based on likely pairings of body parts. The bottom-up strategy is more robust to occlusion and body contact, but the strategy is more difficult to implement. OpenPose is a multi-person human pose estimation algorithm that uses a bottom-up strategy [1 on page 3-0].

To identify body parts in an image, OpenPose uses a pretrained neural network that predicts heatmaps and part affinity fields (PAFs) for body parts in an input image [2 on page 3-0]. Each heatmap shows the probability that a particular type of body part is located at each pixel in the image. The PAFs are vector fields that indicate whether two body parts are connected. For each defined type of body part pairing, such as neck to left shoulder, there are two PAFs that show the x- and y-component of the vector field between instances of the body parts.

To assemble body parts into individual people, the OpenPose algorithm performs a series of post-processing operations. The first operation identifies and localizes body parts using the heatmaps returned by the network. Subsequent operations identify actual connections between body parts, resulting in the individual poses. For more details about the algorithm, see Identify Poses from Heatmaps and PAFs on page 3-0 .

Import the Network

Import a pretrained network from an ONNX file

```
dataDir = fullfile(tempdir, 'OpenPose');
trainedOpenPoseNet_url = 'https://www.mathworks.com/supportfiles/vision/data/trainedOpenPoseNet.r
downloadTrainedOpenPoseNet(trainedOpenPoseNet_url, dataDir)
```

Pretrained OpenPose network already exists.

Download and install the Deep Learning Toolbox™ Converter for ONNX Model Format support package.

If Deep Learning Toolbox Converter™ for ONNX Model Format is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install. If the support package is installed, then the `importONNXLayers` function returns a `LayerGraph` object.

```
modelfile = fullfile(dataDir, 'human-pose-estimation.onnx');
layers = importONNXLayers(modelfile, "ImportWeights", true);
```

Warning: ONNX network has multiple outputs. `importONNXLayers` inserts placeholder layers for the o

Remove the unused output layers.

```
layers = removeLayers(layers, ["Output_node_95" "Output_node_98" "Output_node_147" "Output_node_1
net = dlnetwork(layers);
```

Predict Heatmaps and PAFs of Test Image

Read and display a test image.

```
im = imread("visionteam.jpg");  
imshow(im)
```



The network expects image data of data type `single` in the range `[-0.5, 0.5]`. Shift and rescale the data to this range.

```
netInput = im2single(im)-0.5;
```

The network expects the color channels in the order blue, green, red. Switch the order of the image color channels.

```
netInput = netInput(:,:, [3 2 1]);
```

Store the image data as a `darray`.

```
netInput = darray(netInput, "SSC");
```

Predict the heatmaps, which are output from the 2-D convolutional layer named 'node_147'.

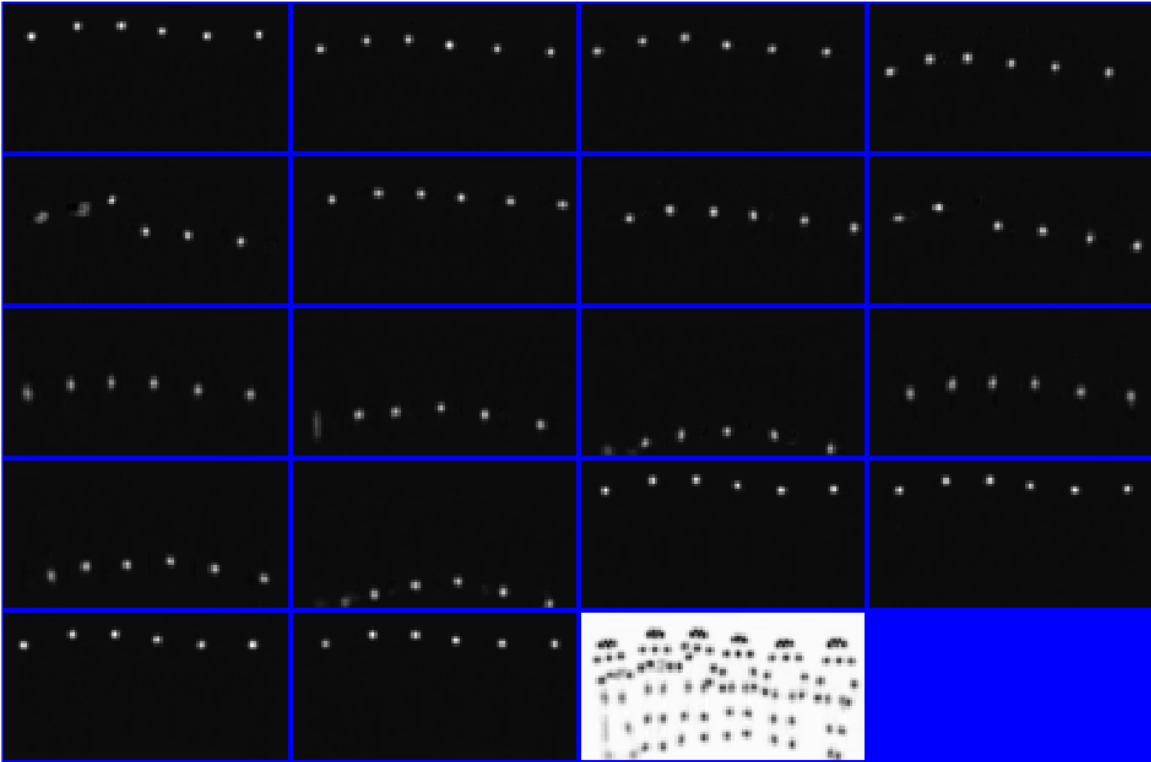
```
heatmaps = predict(net, netInput, "Outputs", "node_147");
```

Get the numeric heatmap data stored in the `darray`. The data has 19 channels. Each channel corresponds to a heatmap for a unique body part, with one additional heatmap for the background.

```
heatmaps = extractdata(heatmaps);
```

Display the heatmaps in a montage, rescaling the data to the range `[0, 1]` expected of images of data type `single`. The scene has six people, and there are six bright spots in each heatmap.

```
montage(rescale(heatmaps), "BackgroundColor", "b", "BorderSize", 3)
```



To visualize the correspondence of bright spots with the bodies, display the first heatmap in falsecolor over the test image.

```
idx = 1;  
hmap = heatmaps(:,:,idx);  
hmap = imresize(hmap,size(im,[1 2]));  
imshowpair(hmap,im);
```



The OpenPose algorithm does not use the background heatmap to determine the location of body parts. Remove the background heatmap.

```
heatmaps = heatmaps(:,:,1:end-1);
```

Predict the PAFs, which are output from the 2-D convolutional layer named 'node_150'.

```
pafs = predict(net,netInput,"Outputs","node_150");
```

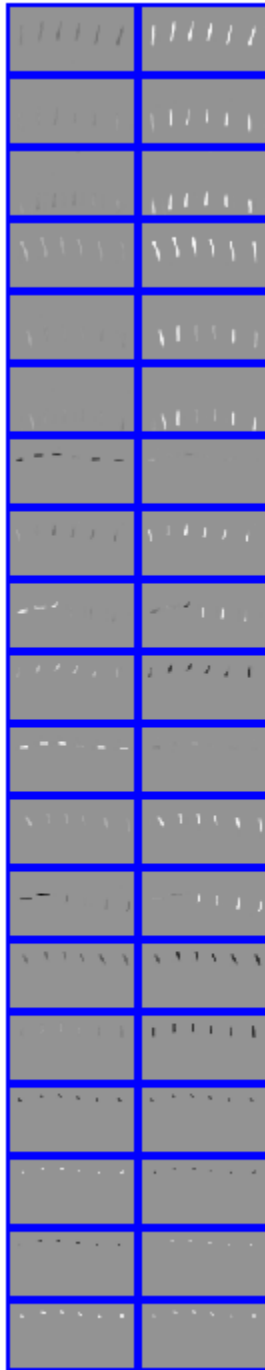
Get the numeric PAF data stored in the `dIarray`. The data has 38 channels. There are two channels for each type of body part pairing, which represent the x- and y-component of the vector field.

```
pafs = extractdata(pafs);
```

Display the PAFs in a montage, rescaling the data to the range [0, 1] expected of images of data type `single`. The two columns show the x- and y-components of the vector field, respectively. The body part pairings are in the order determined by `params.PAF_INDEX` variable.

- Pairs of body parts with a mostly vertical connection have large magnitudes for the y-component pairings and negligible values for the x-component pairings. One example is the right hip to right knee connection, which appears in the second row. Note that the PAFs depend on the actual poses in the image. An image with a body in a different orientation, such as lying down, will not necessarily have a large y-component magnitude for the right hip to right knee connection.
- Pairs of body parts with a mostly horizontal connection have large magnitudes for the x-component pairings and negligible values for the y-component pairings. One example is the neck to left shoulder connection, which appears in the seventh row. Note that the PAF uses
- Pairs of body part at an angle have values for both x- and y-components of the vector field. One example is the neck to left hip, which appears in the first row.

```
montage(rescale(pafs),"Size",[19 2],"BackgroundColor","b","BorderSize",3)
```

To visualize the correspondence of the PAFs with the bodies, display the x- and y-component of the first type of body part pair in falsecolor over the test image.

```
idx = 1;
impair = horzcat(im,im);
pafpair = horzcat(pafs(:,:,2*idx-1),pafs(:,:,2*idx));
pafpair = imresize(pafpair,size(impair,[1 2]));
imshowpair(pafpair,impair);
```



Identify Poses from Heatmaps and PAFs

The post-processing part of the algorithm identifies the individual poses of the people in the image using the heatmaps and PAFs returned by the neural network.

Get parameters of the OpenPose algorithm using the `getBodyPoseParameters` helper function. The function is attached to the example as a supporting file. The function returns a struct with parameters such as the number of body parts and connections between body part types to consider. The parameters also include thresholds that you can adjust to improve the performance of the algorithm.

```
params = getBodyPoseParameters;
```

Identify individual people and their poses by using the `getBodyPoses` helper function. This function is attached to the example as a supporting file. The helper function performs all post-processing steps for pose estimation:

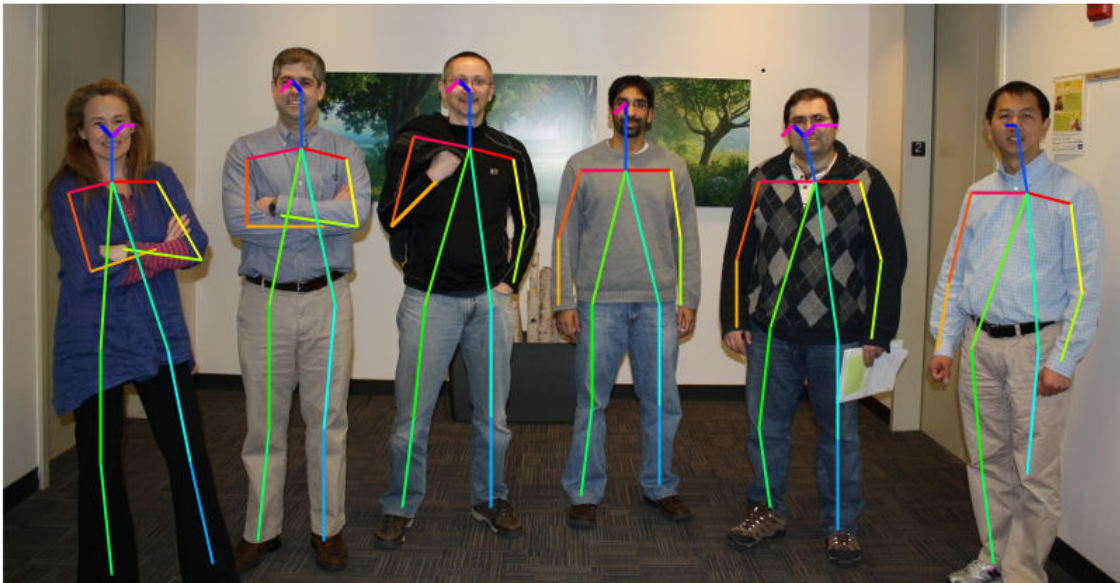
- 1 Detect the precise body part locations from the heatmaps using nonmaximum suppression.
- 2 For each type of body part pairing, generate all possible pairs between detected body parts. For instance, generate all possible pairs between the six necks and the six left shoulders. The result is a bipartite graph.
- 3 Score the pairs by computing the line integral of the straight line connecting the two detected body parts through the PAF vector field. A large score indicates a strong connection between detected body parts.
- 4 Sort the possible pairs by their scores and find the valid pairs. Valid body part pairs are pairs that connect two body parts that belong to the same person. Typically, pairs with the largest score are considered first because they are most likely to be a valid pair. However, the algorithm compensates for occlusion and proximity using additional constraints. For example, the same person cannot have duplicate pairs of body parts, and one body part cannot belong to two different people.
- 5 Knowing which body parts are connected, assemble the body parts into separate poses for each individual person.

The helper function returns a 3-D matrix. The first dimension represents the number of identified people in the image. The second dimension represents the number of body part types. The third dimension indicates the x- and y-coordinates for each body part of each person. If a body part is not detected in the image, then the coordinates for that part are [NaN NaN].

```
poses = getBodyPoses(heatmaps,pafs,params);
```

Display the body poses using the `renderBodyPoses` helper function. This function is attached to the example as a supporting file.

```
renderBodyPoses(im,poses,size(heatmaps,1),size(heatmaps,2),params);
```



References

[1] Cao, Zhe, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. "OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields." *ArXiv:1812.08008 [Cs]*, May 30, 2019. <http://arxiv.org/abs/1812.08008>.

[2] Osokin, Daniil. "Real-Time 2D Multi-Person Pose Estimation on CPU: Lightweight OpenPose." *ArXiv:1811.12004 [Cs]*, November 29, 2018. <http://arxiv.org/abs/1811.12004>.

See Also

```
dlnetwork | importONNXLayers | predict
```

Generate Image from Segmentation Map Using Deep Learning

This example shows how to generate a synthetic image of a scene from a semantic segmentation map using a Pix2PixHD conditional generative adversarial network (CGAN).

Pix2PixHD [1 on page 3-0] consists of two networks that are trained simultaneously to maximize the performance of both.

- 1 The generator is an encoder-decoder style neural network that generates a scene image from a semantic segmentation map. A CGAN network trains the generator to generate a scene image that the discriminator misclassifies as real.
- 2 The discriminator is a fully convolutional neural network that compares a generated scene image and the corresponding real image and attempts to classify them as fake and real, respectively. A CGAN network trains the discriminator to correctly distinguish between generated and real image.

The generator and discriminator networks compete against each other during training. The training converges when neither network can improve further.

Download CamVid Data Set

This example uses the CamVid data set [2 on page 3-0] from the University of Cambridge for training. This data set is a collection of 701 images containing street-level views obtained while driving. The data set provides pixel labels for 32 semantic classes including car, pedestrian, and road.

Download the CamVid data set from these URLs. The download time depends on your internet connection.

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.z
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.
```

```
dataDir = fullfile(tempdir, 'CamVid');
downloadCamVidData(dataDir, imageURL, labelURL);
imgDir = fullfile(dataDir, "images", "701_StillsRaw_full");
labelDir = fullfile(dataDir, 'labels');
```

Preprocess Training Data

Create an `imageDatastore` to store the images in the CamVid data set.

```
imds = imageDatastore(imgDir);
imageSize = [576 768];
```

Define the class names and pixel label IDs of the 32 classes in the CamVid data set using the helper function `defineCamVid32ClassesAndPixelLabelIDs`. Get a standard color map for the CamVid data set using the helper function `camvid32ColorMap`. The helper functions are attached to the example as supporting files.

```
numClasses = 32;
[classes, labelIDs] = defineCamVid32ClassesAndPixelLabelIDs;
cmap = camvid32ColorMap;
```

Create a `pixelLabelDatastore` to store the pixel label images.

```
pxds = pixelLabelDatastore(labelDir, classes, labelIDs);
```

Preview a pixel label image and the corresponding ground truth scene image. Convert the labels from categorical labels to RGB colors by using the `label2rgb` function, then display the pixel label image and ground truth image in a montage.

```
im = preview(imds);
px = preview(pxds);
px = label2rgb(px, cmap);
montage({px, im})
```



Partition the data into training and test sets using the helper function `partitionCamVidForPix2PixHD`. This function is attached to the example as a supporting file. The helper function splits the data into 648 training files and 32 test files.

```
[imdsTrain, imdsTest, pxdsTrain, pxdsTest] = partitionCamVidForPix2PixHD(imds, pxds, classes, labelIDs)
```

Use the `combine` function to combine the pixel label images and ground truth scene images into a single datastore.

```
dsTrain = combine(pxdsTrain, imdsTrain);
```

Augment the training data by using the `transform` function with custom preprocessing operations specified by the helper function `preprocessCamVidForPix2PixHD`. This helper function is attached to the example as a supporting file.

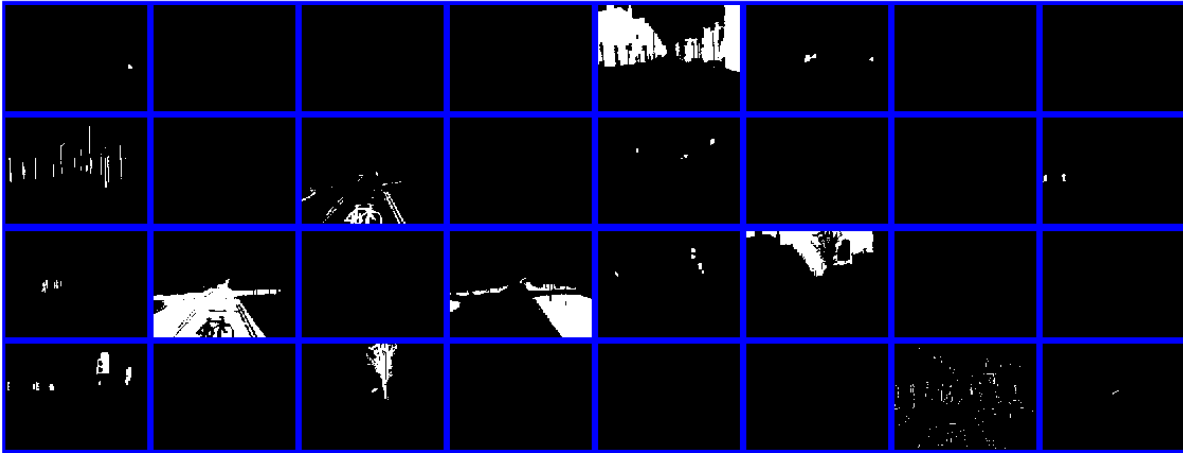
The `preprocessCamVidForPix2PixHD` function performs these operations:

- 1 Scale the ground truth data to the range $[-1, 1]$. This range matches the range of the final `tanhLayer` (Deep Learning Toolbox) in the generator network.
- 2 Resize the image and labels to the output size of the network, 576-by-768 pixels, using bicubic and nearest neighbor downsampling, respectively.
- 3 Convert the single channel segmentation map to a 32-channel one-hot encoded segmentation map using the `onehotencode` (Deep Learning Toolbox) function.
- 4 Randomly flip image and pixel label pairs in the horizontal direction.

```
dsTrain = transform(dsTrain, @(x) preprocessCamVidForPix2PixHD(x, imageSize));
```

Preview the channels of a one-hot encoded segmentation map in a montage. Each channel represents a one-hot map corresponding to pixels of a unique class.

```
map = preview(dsTrain);
montage(map{1}, 'Size', [4 8], 'Bordersize', 5, 'BackgroundColor', 'b')
```



Create Generator Network

Define a generator network that generates a scene image from a depth-wise one-hot encoded segmentation map. This input has same height and width as the original segmentation map and the same number of channels as classes.

```
generatorInputSize = [imageSize numClasses];
```

Create layers of the initial subnetwork. `reflectionPad2dLayer` is a custom layer implemented specifically for this example. This layer is attached to the example as a supporting file.

```
numFiltersFirstConvLayerGenerator = 64;
filterSize = [7 7];

initialLayers = [ ...
    imageInputLayer(generatorInputSize, 'Normalization', 'none', 'Name', 'inputLayer') ...
    reflectionPad2dLayer(3, 'iPad') ...
    convolution2dLayer(filterSize, numFiltersFirstConvLayerGenerator, 'Name', 'iConv') ...
    groupNormalizationLayer('channel-wise', 'Name', 'iGn') ...
    reluLayer('Name', 'iRelu')
];
```

Add layers of the downsampling subnetwork. Use four downsampling convolutional layers. Each downsampling layer has twice the number of filters as the previous convolutional layer.

```
numFilters = numFiltersFirstConvLayerGenerator;
numDownsamplingLayers = 4;
filterSize = [3 3];

downsamplingLayers = [];
for idx = 1:numDownsamplingLayers
    % Compute the number of filters in the next convolutional layer
```

```

numFilters = numFilters*2;

s = int2str(idx);
downsamplingLayers = [
    downsamplingLayers ...
    convolution2dLayer(filterSize,numFilters,"Name",strcat("dConv",s), ...
        "Stride",2,"Padding",1) ...
    groupNormalizationLayer("channel-wise",'Name',strcat("dGn",s)) ...
    reluLayer('Name',strcat('dRelu',s))
];
end

```

Create a layer graph from the initial subnetwork and downsampling subnetwork.

```
generator = layerGraph([initialLayers downsamplingLayers]);
```

Create layers of the residual subnetwork. Specify nine residual blocks in the generator. The residual connection in the first residual block is between the additional layer of the first block and the final ReLU layer of the downsampling network. The residual connection in subsequent residual blocks are between the addition layer of the current block and the addition layer of the previous block.

The number of filters in the residual convolutional layers is equal to the number of filters in the last downsampling convolutional layer, 1024.

```

numResidualBlocks = 9;

for idx = 1:numResidualBlocks
    % Get the name of the layer that acts as the source of the residual connection
    res = generator.Layers(end).Name;

    % Specify the layer names of the residual block
    s = int2str(idx);
    convLayer1Name = strcat("rConv",s,"_1");
    convLayer2Name = strcat("rConv",s,"_2");
    gnLayer1Name = strcat("rGn",s,"_1");
    gnLayer2Name = strcat("rGn",s,"_2");
    pad1Name = strcat("rPad",s,"_1");
    pad2Name = strcat("rPad",s,"_2");

    residualBlockLayers = [
        reflectionPad2dLayer(1,pad1Name) ...
        convolution2dLayer(filterSize,numFilters,"Name",convLayer1Name) ...
        groupNormalizationLayer('channel-wise','Name',gnLayer1Name) ...
        reflectionPad2dLayer(1,pad2Name) ...
        convolution2dLayer(filterSize,numFilters,"Name",convLayer2Name) ...
        groupNormalizationLayer('channel-wise','Name',gnLayer2Name) ...
        reluLayer("Name",strcat("rRelu",s)) ...
        additionLayer(2,'Name',strcat("rAdd",s))
    ];

    % Add the layers to the layer graph
    lg = addLayers(generator,residualBlockLayers);
    generator = connectLayers(lg,generator.Layers(end).Name,residualBlockLayers(1).Name);

    % Link the residual connection
    generator = connectLayers(generator,res,strcat("rAdd",s,"/in2"));
end

```

Create layers of the upsampling subnetwork. Use four upsampling convolutional layers, which is the same as the number of downconvolutional layers. Each upsampling convolutional layer has half the number of filters as the previous convolutional layer.

```
for idx=1:numDownsamplingLayers

    % Compute the number of filters in the next convolutional layer
    numFilters = numFilters/2;

    s = int2str(idx);
    upsamplingLayers = [
        transposedConv2dLayer(filterSize,numFilters,"Name",strcat("uConv",s), ...
            "Stride",2,"Cropping","Same") ...
        groupNormalizationLayer('channel-wise','Name',strcat("uGn",s)) ...
        reluLayer('Name',strcat('uRelu',s));
    ];

    % Add the upsampling layers to the layer graph
    lg = addLayers(generator,upsamplingLayers);
    generator = connectLayers(lg,generator.Layers(end).Name,upsamplingLayers(1).Name);

end
```

Create layers of the final subnetwork. Specify the filter size and number of filters of the final convolutional layer of the generator. The final layer is a hyperbolic tangent layer, which produces activations in the range $[-1, 1]$.

```
filterSize = [7 7];
numFilters = 3;

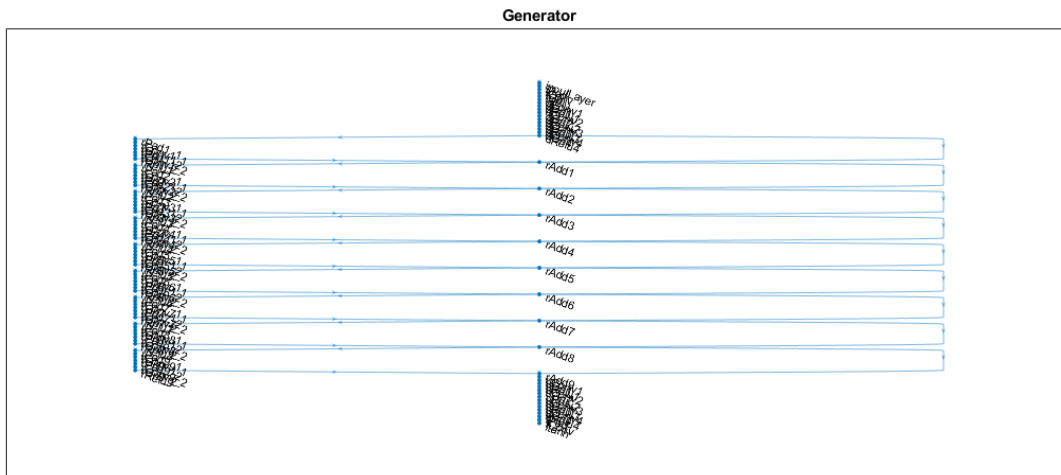
finalLayers = [
    reflectionPad2dLayer(3,'fPad') ...
    convolution2dLayer(filterSize,numFilters,'Name','fConv') ...
    tanhLayer('Name','ftanh')
];
```

Add the final subnetwork layers to the layer graph.

```
lg = addLayers(generator,finalLayers);
lgraphGenerator = connectLayers(lg,generator.Layers(end).Name,finalLayers(1).Name);
```

Visualize the generator network in a plot.

```
plot(lgraphGenerator)
title("Generator")
```

To train the network with a custom training loop and to enable automatic differentiation, convert the layer graph to a `dlnetwork` (Deep Learning Toolbox) object.

```
dlnetGenerator = dlnetwork(lgraphGenerator);
```

Visualize the network architecture using the Deep Network Designer (Deep Learning Toolbox) app.

```
deepNetworkDesigner(lgraphGenerator)
```

Create Discriminator Network

Define a discriminator network that classifies an input image as either real (1) or fake (0).

The input to the discriminator is the depth-wise concatenation of the one-hot encoded segmentation maps and the scene image to be classified. Specify the number of channels input to the discriminator as the total number of labeled classes and image color channels.

```
numImageChannels = 3;
numChannelsDiscriminator = numClasses + numImageChannels;
discriminatorInputSize = [imageSize numChannelsDiscriminator];
```

Specify the filter size and number of filters in the first convolutional layer of the discriminator.

```
filterSize = [4 4];
numFilters = 64;
```

Define the layers of the discriminator.

```
discriminator = [
    imageInputLayer(discriminatorInputSize,"Name","inputLayer","Normalization","none")
    convolution2dLayer(filterSize,numFilters,"Name","iConv", ...
        "Padding",2,"Stride",2)
    leakyReluLayer(0.2,"Name","lrelu1")
    convolution2dLayer(filterSize,numFilters*2,"Name","dConv1", ...
        "Padding",2,"Stride",2)
    groupNormalizationLayer('channel-wise',"Name","dGn1")
    leakyReluLayer(0.2,"Name","lrelu2")
    convolution2dLayer(filterSize,numFilters*4,"Name","dConv2", ...
```

```

        "Padding",2,"Stride",2);
groupNormalizationLayer('channel-wise',"Name","dGn2")
leakyReluLayer(0.2,"Name","lrelu3")
convolution2dLayer(filterSize,numFilters*8,"Name","dConv3", ...
    "Padding",2)
groupNormalizationLayer('channel-wise',"Name","dGn3")
leakyReluLayer(0.2,"Name","lrelu4")
convolution2dLayer(filterSize,1,"Name","fConv", ...
    "Padding",2)
];

```

Create the layer graph.

```
lgraphDiscriminator = layerGraph(discriminator);
```

Visualize the discriminator network in a plot.

```
plot(lgraphDiscriminator)
title("Discriminator")
```



To train the network with a custom training loop and to enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator);
```

Visualize the network architecture using the Deep Network Designer (Deep Learning Toolbox) app.

```
deepNetworkDesigner(lgraphDiscriminator)
```

Define Model Gradients and Loss Functions

The helper function `modelGradients` calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the feature matching loss and VGG loss for the generator. This function is defined in Supporting Functions on page 3-0 section of this example.

Generator Loss

The objective of the generator is to generate images that the discriminator classifies as real (1). The generator loss consists of three losses.

- The adversarial loss is computed as the squared difference between a vector of ones and the discriminator predictions on the generated image. $\hat{Y}_{generated}$ are discriminator predictions on the image generated by the generator. This loss is implemented using part of the `pix2pixhdAdversarialLoss` helper function defined in the Supporting Functions on page 3-0 section of this example.

$$lossAdversarialGenerator = (1 - \hat{Y}_{generated})^2$$

- The feature matching loss penalises the L^1 distance between the real and generated feature maps obtained as predictions from the discriminator network. T is total number of discriminator feature layers. Y_{real} and $\hat{Y}_{generated}$ are the ground truth images and generated images, respectively. This loss is implemented using the `pix2pixhdFeatureMatchingLoss` helper function defined in the Supporting Functions on page 3-0 section of this example

$$lossFeatureMatching = \sum_{i=1}^T \left\| \left\| Y_{real} - \hat{Y}_{generated} \right\|_1 \right\|_1$$

- The perceptual loss penalises the L^1 distance between real and generated feature maps obtained as predictions from a feature extraction network. T is total number of feature layers. $Y_{VggReal}$ and $\hat{Y}_{VggGenerated}$ are network predictions for ground truth images and generated images, respectively. This loss is implemented using the `pix2pixhdVggLoss` helper function defined in the Supporting Functions on page 3-0 section of this example. The feature extraction network is created in Load Feature Extraction Network on page 3-0 .

$$lossVgg = \sum_{i=1}^T \left\| \left\| Y_{VggReal} - \hat{Y}_{VggGenerated} \right\|_1 \right\|_1$$

The overall generator loss is a weighted sum of all three losses. λ_1 , λ_2 , and λ_3 are the weight factors for adversarial loss, feature matching loss, and perceptual loss, respectively.

$$lossGenerator = \lambda_1 * lossAdversarialGenerator + \lambda_2 * lossFeatureMatching + \lambda_3 * lossPerceptual$$

Note that the adversarial loss and feature matching loss for the generator are computed for two different scales.

Discriminator Loss

The objective of the discriminator is to correctly distinguish between ground truth images and generated images. The discriminator loss is a sum of two components:

- The squared difference between a vector of ones and the predictions of the discriminator on real images
- The squared difference between a vector of zeros and the predictions of the discriminator on generated images

$$lossDiscriminator = (1 - Y_{real})^2 + (0 - \hat{Y}_{generated})^2$$

The discriminator loss is implemented using part of the `pix2pixhdAdversarialLoss` helper function defined in the Supporting Functions on page 3-0 section of this example. Note that adversarial loss for the discriminator is computed for two different scales.

Load Feature Extraction Network

This example modifies a pretrained VGG-19 deep neural network to extract the features of the real and generated images at various layers. These multilayer features are used to compute the perceptual loss of the generator.

To get a pretrained VGG-19 network, install `vgg19` (Deep Learning Toolbox). If you do not have the required support packages installed, then the software provides a download link.

```
netVGG = vgg19;
```

Visualize the network architecture using the Deep Network Designer (Deep Learning Toolbox) app.

```
deepNetworkDesigner(netVGG)
```

To make the VGG-19 network suitable for feature extraction, keep the layers up to 'pool5' and remove all of the fully connected layers from the network. The resulting network is a fully convolutional network.

```
netVGG = layerGraph(netVGG.Layers(1:38));
```

Create a new image input layer with no normalization. Replace the original image input layer with the new layer.

```
inp = imageInputLayer([imageSize 3], "Normalization", "None", "Name", "Input");  
netVGG = replaceLayer(netVGG, "input", inp);  
netVGG = dlnetwork(netVGG);
```

Specify Training Options

Specify the options for Adam optimization. Train for 60 epochs. Specify identical options for the generator and discriminator networks.

- Specify an equal learning rate of 0.0002.
- Initialize the trailing average gradient and trailing average gradient-square decay rates with [].
- Use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.
- Use a mini-batch size of 1 for training.

```
numEpochs = 60;  
learningRate = 0.0002;  
trailingAvgGenerator = [];  
trailingAvgSqGenerator = [];  
trailingAvgDiscriminator = [];  
trailingAvgSqDiscriminator = [];  
gradientDecayFactor = 0.5;  
squaredGradientDecayFactor = 0.999;  
miniBatchSize = 1;
```

Create a `minibatchqueue` (Deep Learning Toolbox) object that manages the mini-batching of observations in a custom training loop. The `minibatchqueue` object also casts data to a `dlarray` (Deep Learning Toolbox) object that enables auto differentiation in deep learning applications.

Specify the mini-batch data extraction format as SSCB (spatial, spatial, channel, batch). Set the `DispatchInBackground` name-value pair argument as the boolean returned by `canUseGPU`. If a supported GPU is available for computation, then the `minibatchqueue` object preprocesses mini-batches in the background in a parallel pool during training.

```
mbqTrain = minibatchqueue(dsTrain,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","DispatchInBackground",canUseGPU);
```

Train the Network

By default, the example downloads a pretrained version of the Pix2PixHD network for the CamVid data set by using the helper function `downloadTrainedPix2PixHDNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for current mini-batch using the `next` (Deep Learning Toolbox) function.
- Evaluate the model gradients using the `dlfeval` (Deep Learning Toolbox) function and the `modelGradients` helper function.
- Update the network parameters using the `adamupdate` (Deep Learning Toolbox) function.
- Update the training progress plot for every iteration and display various computed losses.

A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training (requires Parallel Computing Toolbox™). Training takes about 22 hours on an NVIDIA™ Titan RTX and can take even longer depending on your GPU hardware. If your GPU device has less memory, try reducing the size of the input images by specifying the `imageSize` variable as [480 640] in the Preprocess Training Data on page 3-0 section of the example.

```
doTraining = false;
if doTraining
    fig = figure;

    lossPlotter = configureTrainingProgressPlotter(fig);
    iteration = 0;

    % Loop over epochs
    for epoch = 1:numEpochs

        % Reset and shuffle the data
        reset(mbqTrain);
        shuffle(mbqTrain);

        % Loop over each image
        while hasdata(mbqTrain)
            iteration = iteration + 1;

            % Read data from current mini-batch
            [dlInputSegMap,dlRealImage] = next(mbqTrain);

            % Evaluate the model gradients and the generator state using
            % dlfeval and the GANLoss function listed at the end of the
            % example
            [gradParamsG,gradParamsD,lossGGAN,lossGFM,lossGVGG,lossD] = dlfeval( ...
```

```

        @modelGradients,dlInputSegMap,dlRealImage,dlnetGenerator,dlnetDiscriminator,netV

% Update the generator parameters
[dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = adamupdate( ...
    dlnetGenerator,gradParamsG, ...
    trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

% Update the discriminator parameters
[dlnetDiscriminator,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = adamupdate( ...
    dlnetDiscriminator,gradParamsD, ...
    trailingAvgDiscriminator,trailingAvgSqDiscriminator,iteration, ...
    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

% Plot and display various losses
lossPlotter = updateTrainingProgressPlotter(lossPlotter,iteration, ...
    epoch,numEpochs,lossGGAN,lossGFM,lossGVGG,lossD);
end
end
save('trainedPix2PixHDNet.mat','dlnetGenerator');
else
    trainedPix2PixHDNet_url = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedPix2PixHDNet.mat';
    netDir = fullfile(tempdir,'CamVid');
    downloadTrainedPix2PixHDNet(trainedPix2PixHDNet_url,netDir);
    load(fullfile(netDir,'trainedPix2PixHDNet.mat'));
end
end

```

Evaluate Generated Images from Test Data

The performance of this trained Pix2PixHD network is limited because the number of CamVid training images is relatively small. Additionally, some images belong to an image sequence and therefore are correlated with other images in the training set. To improve the effectiveness of the Pix2PixHD network, train the network using a different data set that has a larger number of training images without correlation.

Because of the limitations, this Pix2PixHD network generates more realistic images for some test images than for others. To demonstrate the difference in results, compare the generated images for the first and third test image. The camera angle of the first test image has an uncommon vantage point that faces more perpendicular to the road than the typical training image. In contrast, the camera angle of the third test image has a typical vantage point that faces along the road and shows two lanes with lane markers. The network has significantly better performance generating a realistic image for the third test image than for the first test image.

Get the first ground truth scene image from the test data. Resize the image using bicubic interpolation.

```

idxToTest = 1;
gtImage = readimage(imdsTest,idxToTest);
gtImage = imresize(gtImage,imageSize,"bicubic");

```

Get the corresponding pixel label image from the test data. Resize the pixel label image using nearest neighbor interpolation.

```

segMap = readimage(pxdsTest,idxToTest);
segMap = imresize(segMap,imageSize,"nearest");

```

Convert the pixel label image to a multichannel one-hot segmentation map by using the `onehotencode` (Deep Learning Toolbox) function.

```
segMap1Hot = onehotencode(segMap,3, 'single');
```

Create a `dlarray` object that inputs data to the generator. If a supported GPU is available for computation, then perform inference on a GPU by converting the data to a `gpuArray` object.

```
dlSegMap = dlarray(segMap1Hot, 'SSCB');
if canUseGPU
    dlSegMap = gpuArray(dlSegMap);
end
```

Generate a scene image from the generator and one-hot segmentation map using the `predict` (Deep Learning Toolbox) function.

```
dlGeneratedImage = predict(dlnetGenerator,dlSegMap);
generatedImage = extractdata(gather(dlGeneratedImage));
```

The final layer of the generator network produces activations in the range $[-1, 1]$. For display, rescale the activations to the range $[0, 1]$.

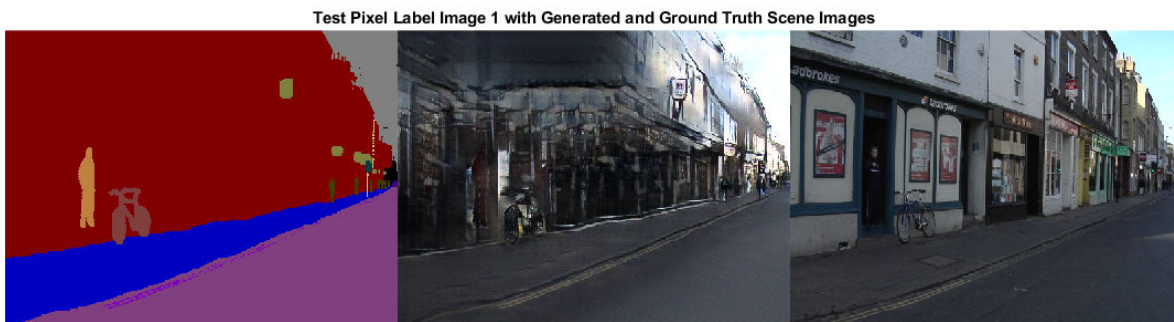
```
generatedImage = rescale(generatedImage);
```

For display, convert the labels from categorical labels to RGB colors by using the `label2rgb` function.

```
coloredSegMap = label2rgb(segMap,cmap);
```

Display the RGB pixel label image, generated scene image, and ground truth scene image in a montage.

```
figure
montage({coloredSegMap generatedImage gtImage},'Size',[1 3])
title(['Test Pixel Label Image ',num2str(idxToTest),' with Generated and Ground Truth Scene Images'])
```



Get the third ground truth scene image from the test data. Resize the image using bicubic interpolation.

```
idxToTest = 3;
gtImage = readimage(imdsTest,idxToTest);
gtImage = imresize(gtImage,imageSize,"bicubic");
```

To get the third pixel label image from the test data and to generate the corresponding scene image, you can use the helper function `evaluatePix2PixHD`. This helper function is attached to the example as a supporting file.

The `evaluatePix2PixHD` function performs the same operations as the evaluation of the first test image:

- Get a pixel label image from the test data. Resize the pixel label image using nearest neighbor interpolation.
- Convert the pixel label image to a multichannel one-hot segmentation map.
- Create a `dLarray` object to input data to the generator. For GPU inference, convert the data to a `gpuArray` object.
- Generate a scene image from the generator and one-hot segmentation map using the `predict` (Deep Learning Toolbox) function.
- Rescale the activations to the range `[0, 1]`.

```
[generatedImage,segMap] = evaluatePix2PixHD(pxdsTest,idxToTest,imageSize,dlnetGenerator);
```

For display, convert the labels from categorical labels to RGB colors by using the `label2rgb` function.

```
coloredSegMap = label2rgb(segMap,cmap);
```

Display the RGB pixel label image, generated scene image, and ground truth scene image in a montage.

```
figure
montage({coloredSegMap generatedImage gtImage},'Size',[1 3])
title(['Test Pixel Label Image ',num2str(idxToTest),' with Generated and Ground Truth Scene Images'])
```



Evaluate Generated Images from Custom Pixel Label Images

To evaluate how well the network generalizes to pixel label images outside the CamVid data set, generate scene images from custom pixel label images. This example uses pixel label images that were created using the Image Labeler app. The pixel label images are attached to the example as supporting files. No ground truth images are available.

Create a pixel label datastore that reads and processes the pixel label images in the current example directory.

```
cpdxs = pixelLabelDatastore(pwd,classes,labelIDs);
```


For each pixel label image in the datastore, generate a scene image using the helper function `evaluatePix2PixHD`.

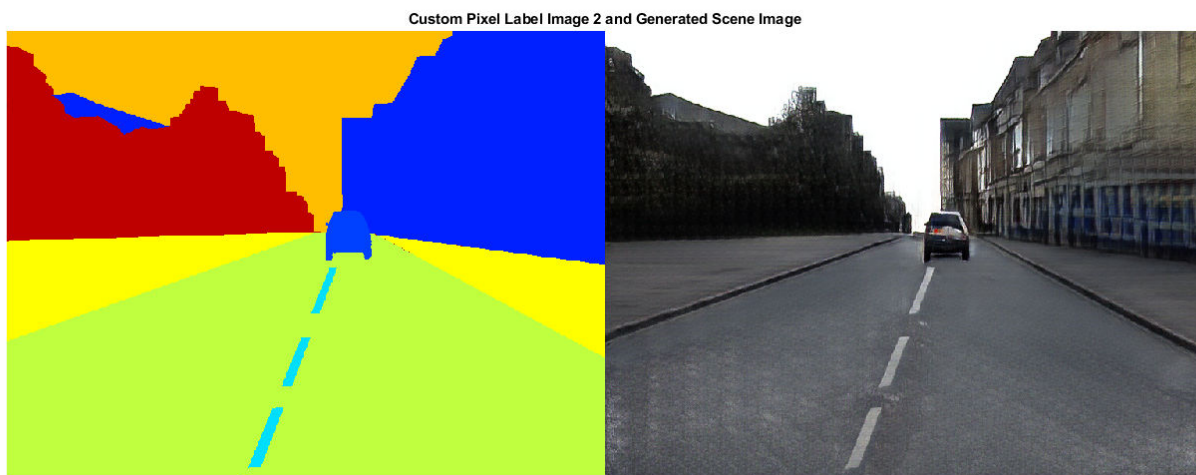
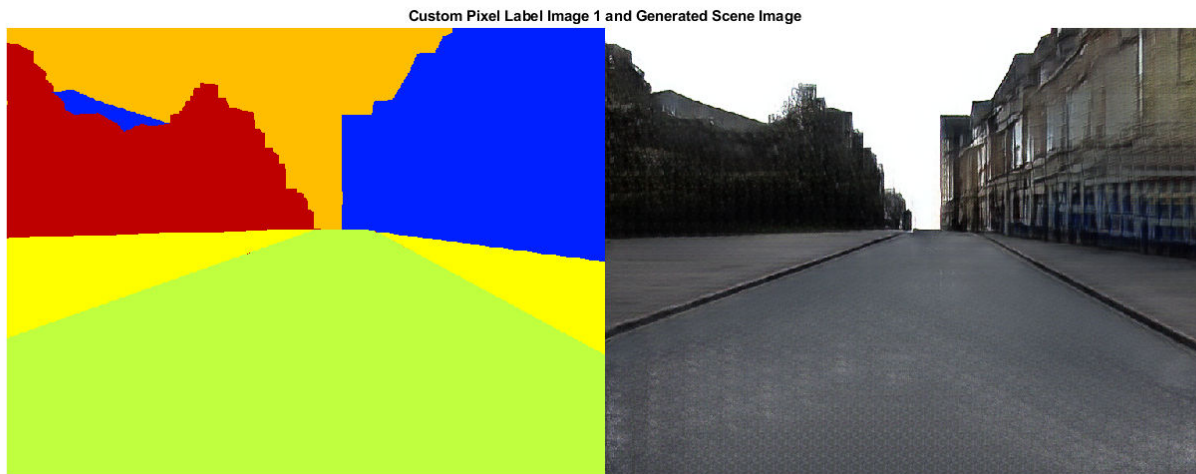
```
for idx = 1:length(cpxds.Files)

    % Get the pixel label image and generated scene image
    [generatedImage,segMap] = evaluatePix2PixHD(cpxds,idx,imageSize,dlnetGenerator);

    % For display, convert the labels from categorical labels to RGB colors
    coloredSegMap = label2rgb(segMap);

    % Display the pixel label image and generated scene image in a montage
    figure
    montage({coloredSegMap generatedImage})
    title(['Custom Pixel Label Image ',num2str(idx),' and Generated Scene Image'])

end
```



Supporting Functions

Model Gradients Function

The `modelGradients` helper function calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the feature matching loss and VGG loss for the generator.

```
function [gradParamsG,gradParamsD,lossGGAN,lossGFM,lossGVGG,lossD] = modelGradients(inputSegMap,

% Compute the image generated by the generator given the input semantic map
generatedImage = forward(generator,inputSegMap);

% Define the loss weights
lambdaDiscriminator = 1;
lambdaGenerator = 1;
lambdaFeatureMatching = 5;
lambdaVGG = 5;

% Concatenate the image to be classified and the semantic map
inpDiscriminatorReal = cat(3,inputSegMap,realImage);
inpDiscriminatorGenerated = cat(3,inputSegMap,generatedImage);

% Compute the adversarial loss for the discriminator and the generator.
[DLossScale1,GLossScale1,realPredScale1D,fakePredScale1G] = pix2pixHDAdverserialLoss(inpDisc

% Scale the generated image, the real image, and the input semantic map to
% half size
resizedRealImage = dlresize(realImage, 'Scale',0.5, 'Method',"linear");
resizedGeneratedImage = dlresize(generatedImage, 'Scale',0.5, 'Method',"linear");
resizedinputSegMap = dlresize(inputSegMap, 'Scale',0.5, 'Method',"nearest");

% Concatenate the image to be classified and the semantic map
inpDiscriminatorReal = cat(3,resizedinputSegMap,resizedRealImage);
inpDiscriminatorGenerated = cat(3,resizedinputSegMap,resizedGeneratedImage);

% Compute the adversarial loss for the discriminator and the generator
[DLossScale2,GLossScale2,realPredScale2D,fakePredScale2G] = pix2pixHDAdverserialLoss(inpDisc

% Compute the feature matching loss for scale 1
FMLossScale1 = pix2pixHDFeatureMatchingLoss(realPredScale1D,fakePredScale1G);
FMLossScale1 = FMLossScale1 * lambdaFeatureMatching;

% Compute the feature matching loss for scale 2
FMLossScale2 = pix2pixHDFeatureMatchingLoss(realPredScale2D,fakePredScale2G);
FMLossScale2 = FMLossScale2 * lambdaFeatureMatching;

% Compute the VGG loss
VGGLoss = pix2pixHDVGGLoss(realImage,generatedImage,netVGG);
VGGLoss = VGGLoss * lambdaVGG;

% Compute the combined generator loss
lossGCombined = GLossScale1 + GLossScale2 + FMLossScale1 + FMLossScale2 + VGGLoss;
lossGCombined = lossGCombined * lambdaGenerator;

% Compute gradients for the generator
gradParamsG = dlgradient(lossGCombined,generator.Learnables);
```

```

% Compute the combined discriminator loss
lossDCombined = (DLossScale1 + DLossScale2)/2 * lambdaDiscriminator;

% Compute gradients for the discriminator
gradParamsD = dlgradient(lossDCombined,discriminator.Learnables);

% Log the values for displaying later
lossD = gather(extractdata(lossDCombined));
lossGGAN = gather(extractdata(GLossScale1 + GLossScale2));
lossGFM = gather(extractdata(FMLossScale1 + FMLossScale2));
lossVGVG = gather(extractdata(VGGLoss));
end

```

Adversarial Loss Function

The helper function `pix2pixHDAdversarialLoss` computes the adversarial loss gradients for the generator and the discriminator. The function also returns feature maps of the real image and synthetic images.

```

function [DLoss,GLoss,realPredFtrsD,genPredFtrsG] = pix2pixHDAdversarialLoss(inpReal,inpGenerated)

% Discriminator layer names containing feature maps
featureNames = {'lrelu1','lrelu2','lrelu3','lrelu4','fConv'};

% Get the feature maps for the real image from the discriminator
realPredFtrsD = cell(size(featureNames));
[realPredFtrsD{:}] = forward(discriminator,inpReal,"Outputs",featureNames);

% Get the feature maps for the generated image from the discriminator
genPredFtrsD = cell(size(featureNames));
[genPredFtrsD{:}] = forward(discriminator,inpGenerated,"Outputs",featureNames);

% Get the feature map from the final layer to compute the loss
realPredD = realPredFtrsD{end};
genPredD = genPredFtrsD{end};

% Compute the discriminator loss
DLoss = (1 - realPredD).^2 + (genPredD).^2;
DLoss = mean(DLoss,"all");

% Compute the generator loss
genPredFtrsG = cell(size(featureNames));
[genPredFtrsG{:}] = forward(discriminator,inpGenerated,"Outputs",featureNames);
genPredG = genPredFtrsG{end};
GLoss = (1 - genPredG).^2;
GLoss = mean(GLoss,"all");
end

```

Feature Matching Loss Function

The helper function `pix2pixHDFeatureMatchingLoss` computes the feature matching loss between a real image and a synthetic image generated by the generator.

```

function featureMatchingLoss = pix2pixHDFeatureMatchingLoss(realPredFtrs,genPredFtrs)

% Number of features
numFtrsMaps = numel(realPredFtrs);

```

```
% Initialize the feature matching loss
featureMatchingLoss = 0;

for i = 1:numFtrsMaps
    % Get the feature maps of the real image
    a = extractdata(realPredFtrs{i});
    % Get the feature maps of the synthetic image
    b = genPredFtrs{i};

    % Compute the feature matching loss
    featureMatchingLoss = featureMatchingLoss + mean(abs(a - b),"all");
end
end
```

Perceptual VGG Loss Function

The helper function `pix2pixHDVGGLoss` computes the perceptual VGG loss between a real image and a synthetic image generated by the generator.

```
function vggLoss = pix2pixHDVGGLoss(realImage,generatedImage,netVGG)

    featureWeights = [1.0/32 1.0/16 1.0/8 1.0/4 1.0];

    % Initialize the VGG loss
    vggLoss = 0;

    % Specify the names of the layers with desired feature maps
    featureNames = ["relu1_1","relu2_1","relu3_1","relu4_1","relu5_1"];

    % Extract the feature maps for the real image
    activReal = cell(size(featureNames));
    [activReal{:}] = forward(netVGG,realImage,"Outputs",featureNames);

    % Extract the feature maps for the synthetic image
    activGenerated = cell(size(featureNames));
    [activGenerated{:}] = forward(netVGG,generatedImage,"Outputs",featureNames);

    % Compute the VGG loss
    for i = 1:numel(featureNames)
        vggLoss = vggLoss + featureWeights(i)*mean(abs(activReal{i} - activGenerated{i}),"all");
    end
end
```

References

- [1] Wang, Ting-Chun, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs." In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 8798-8807, 2018. <https://doi.org/10.1109/CVPR.2018.00917>.
- [2] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A High-Definition Ground Truth Database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp 88-97.

See Also

`combine` | `imageDatastore` | `pixelLabelDatastore` | `trainNetwork` | `trainingOptions` | `transform` | `vgg19`

More About

- [“Preprocess Images for Deep Learning”](#) (Deep Learning Toolbox)
- [“Datastores for Deep Learning”](#) (Deep Learning Toolbox)
- [“List of Deep Learning Layers”](#) (Deep Learning Toolbox)
- [“Define Custom Training Loops, Loss Functions, and Networks”](#) (Deep Learning Toolbox)

Create Simple Semantic Segmentation Network in Deep Network Designer

This example shows how to create and train a simple semantic segmentation network using Deep Network Designer.

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 road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” on page 14-43.

Preprocess Training Data

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. This example uses a simple data set of 32-by-32 images of triangles for illustration purposes. You can interactively label pixels and export the label data for computer vision applications using Image Labeler. For more information on creating training data for semantic segmentation applications, see “Label Pixels for Semantic Segmentation” on page 14-53.

Load the training data.

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

Create an ImageDatastore containing the images.

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

Create a PixelLabelDatastore containing the ground truth pixel labels. This data set has two classes: "triangle" and "background".

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

Combine the image datastore and the pixel label datastore into a CombinedDatastore object using the `combine` function. A combined datastore maintains parity between the pair of images in the underlying datastores.

```
cds = combine(imds, pxds);
```

Build Network

Open Deep Network Designer.

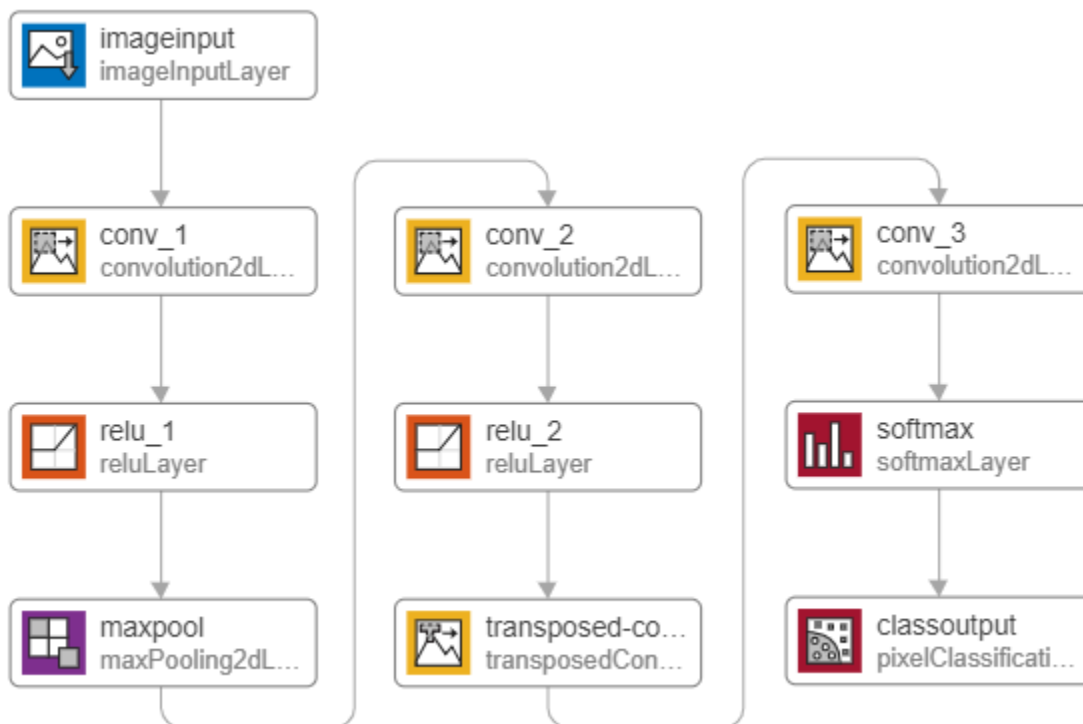
```
deepNetworkDesigner
```

In Deep Network Designer, you can build, edit, and train deep learning networks. Pause on **Blank Network** and click **New**.

Create a semantic segmentation network by dragging layers from the **Layer Library** to the **Designer** pane.

Connect the layers in this order:

- 1 imageInputLayer with InputSize set to 32,32,1
- 2 convolution2dLayer with FilterSize set to 3,3, NumFilters set to 64, and Padding set to 1,1,1,1
- 3 reluLayer
- 4 maxPooling2dLayer with PoolSize set to 2,2, Stride set to 2,2, and Padding set to 0,0,0,0
- 5 convolution2dLayer with FilterSize set to 3,3, NumFilters set to 64, and Padding set to 1,1,1,1
- 6 reluLayer
- 7 transposedConv2dLayer with FilterSize set to 4,4, NumFilters set to 64, Stride set to 2,2, and Cropping set to 1,1,1,1
- 8 convolution2dLayer with FilterSize set to 1,1, NumFilters set to 2, and Padding set to 0,0,0,0
- 9 softmaxLayer
- 10 pixelClassificationLayer



You can also create this network at the command line and then import the network into Deep Network Designer using `deepNetworkDesigner(layers)`.

```
layers = [
    imageInputLayer([32 32 1])
```

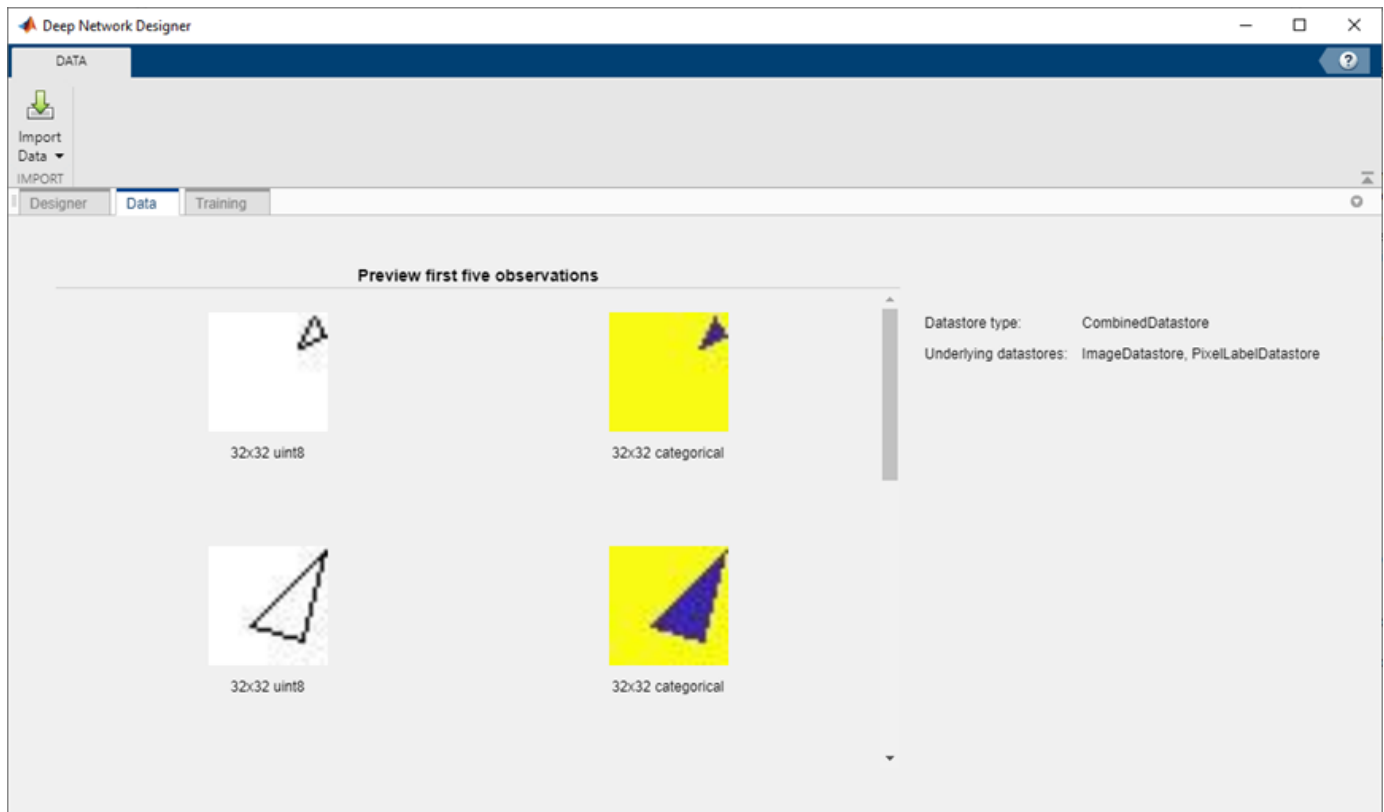
```
convolution2dLayer([3,3],64, 'Padding', [1,1,1,1])
reluLayer()
maxPooling2dLayer([2,2], 'Stride', [2,2])
convolution2dLayer([3,3],64, 'Padding', [1,1,1,1])
reluLayer()
transposedConv2dLayer([4,4],64, 'Stride', [2,2], 'Cropping', [1,1,1,1])
convolution2dLayer([1,1],2)
softmaxLayer()
pixelClassificationLayer()
];
```

This network is a simple semantic segmentation network based on a downsampling and upsampling design. For more information on constructing a semantic segmentation network, see “Create a Semantic Segmentation Network”.

Import Data

To import the training datastore, on the **Data** tab, select **Import Data > Import Datastore**. Select the **CombinedDatastore** object `cds` as the training data. For the validation data, select **None**. Import the training data by clicking **Import**.

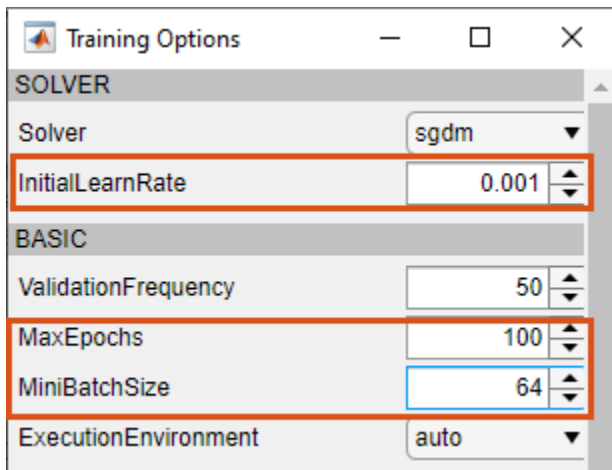
Deep Network Designer displays a preview of the imported semantic segmentation data. The preview displays the training images and the ground truth pixel labels. The network requires input images (left) and returns a classification for each pixel as either triangle or background (right).



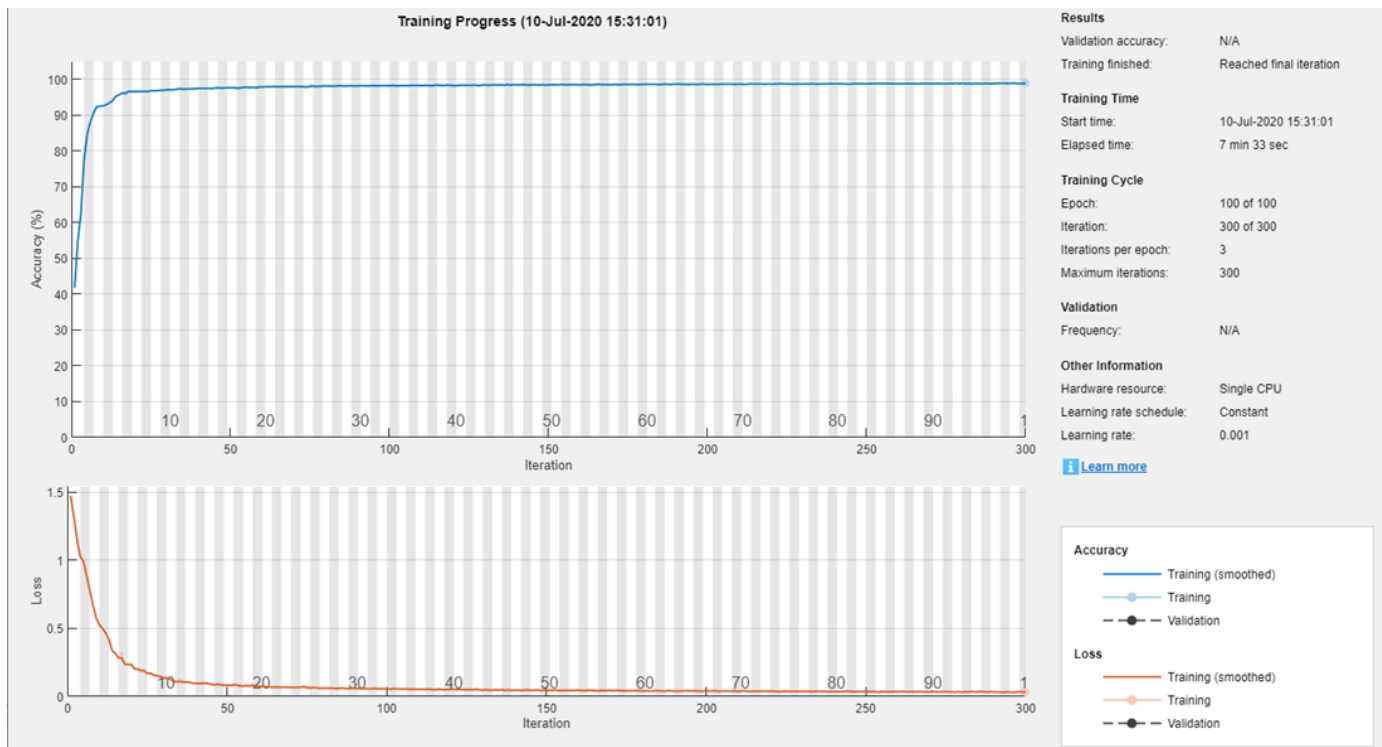
Train Network

Set the training options and train the network.

On the **Training** tab, click **Training Options**. Set **InitialLearnRate** to 0.001, **MaxEpochs** to 100, and **MiniBatchSize** to 64. Set the training options by clicking **Close**.



Train the network by clicking **Train**.



Once training is complete, click **Export** to export the trained network to the workspace. The trained network is stored in the variable `trainedNetwork_1`.

Test Network

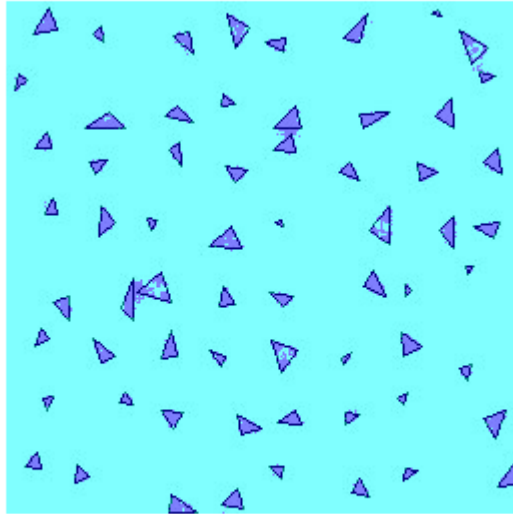
Make predictions using test data and the trained network.

Segment the test image using `semanticseg`. Display the labels over the image by using the `labeloverlay` function.

```
imgTest = imread('triangleTest.jpg');  
testSeg = semanticseg(imgTest,trainedNetwork_1);  
testImageSeg = labeloverlay(imgTest,testSeg);
```

Display the results.

```
figure  
imshow(testImageSeg)
```



The network successfully labels the triangles in the test image.

The semantic segmentation network trained in this example is very simple. To construct more complex semantic segmentation networks, you can use the Computer Vision Toolbox functions `segnetLayers`, `deeplabv3plusLayers`, and `unetLayers`. For an example showing how to use the `deeplabv3plusLayers` function to create a DeepLab v3+ network, see “Semantic Segmentation With Deep Learning”.

Train ACF-Based Stop Sign Detector

Use training data to train an ACF-based object detector for stop signs

Add the folder containing images to the MATLAB path.

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

Load ground truth data, which contains data for stops signs and cars.

```
load('stopSignsAndCarsGroundTruth.mat', 'stopSignsAndCarsGroundTruth')
```

View the label definitions to see the label types in the ground truth.

```
stopSignsAndCarsGroundTruth.LabelDefinitions
```

Select the stop sign data for training.

```
stopSignGroundTruth = selectLabels(stopSignsAndCarsGroundTruth, 'stopSign');
```

Create the training data for a stop sign object detector.

```
trainingData = objectDetectorTrainingData(stopSignGroundTruth);
summary(trainingData)
```

Variables:

```
imageFilename: 41x1 cell array of character vectors
stopSign: 41x1 cell
```

Train an ACF-based object detector.

```
acfDetector = trainACFObjectDetector(trainingData, 'NegativeSamplesFactor', 2);
```

ACF Object Detector Training

The training will take 4 stages. The model size is 34x31.

Sample positive examples(~100% Completed)

Compute approximation coefficients...Completed.

Compute aggregated channel features...Completed.

Stage 1:

Sample negative examples(~100% Completed)

Compute aggregated channel features...Completed.

Train classifier with 42 positive examples and 84 negative examples...Completed.

The trained classifier has 19 weak learners.

Stage 2:

Sample negative examples(~100% Completed)

Found 84 new negative examples for training.

Compute aggregated channel features...Completed.

Train classifier with 42 positive examples and 84 negative examples...Completed.

The trained classifier has 20 weak learners.

Stage 3:

Sample negative examples(~100% Completed)

Found 84 new negative examples for training.

```
Compute aggregated channel features...Completed.  
Train classifier with 42 positive examples and 84 negative examples...Completed.  
The trained classifier has 54 weak learners.  
-----
```

Stage 4:

```
Sample negative examples(~100% Completed)  
Found 84 new negative examples for training.  
Compute aggregated channel features...Completed.  
Train classifier with 42 positive examples and 84 negative examples...Completed.  
The trained classifier has 61 weak learners.  
-----
```

```
ACF object detector training is completed. Elapsed time is 30.3579 seconds.
```

Test the ACF-based detector on a sample image.

```
I = imread('stopSignTest.jpg');  
bboxes = detect(acfDetector,I);
```

Display the detected object.

```
annotation = acfDetector.ModelName;  
I = insertObjectAnnotation(I, 'rectangle', bboxes, annotation);
```

```
figure  
imshow(I)
```



Remove the image folder from the path.

```
rmpath(imageDir);
```

Activity Recognition from Video and Optical Flow Data Using Deep Learning

This example shows how to train an Inflated 3-D (I3D) two-stream convolutional neural network for activity recognition using RGB and optical flow data from videos [1] on page 3-0 .

Vision-based activity recognition involves predicting the action of an object, such as walking, swimming, or sitting, using a set of video frames. Activity recognition from video has many applications, such as human-computer interaction, robot learning, anomaly detection, surveillance, and object detection. For example, online prediction of multiple actions for incoming videos from multiple cameras can be important for robot learning. Compared to image classification, action recognition using videos is challenging to model because of the noisy labels in video data sets, the variety of actions that actors in a video can perform that are heavily class imbalanced, and the compute inefficiency in pretraining on large video data sets. Some deep learning techniques, such as I3D two-stream convolutional networks [1] on page 3-0 , have shown improved performance by leveraging pretraining on large image classification data sets.

Load Data

This example trains an I3D network using the HMDB51 data set. Use the `downloadHMDB51` supporting function, listed at the end of this example, to download the HMDB51 data set to a folder named `hmdb51`.

```
downloadFolder = fullfile(tempdir,"hmdb51");
downloadHMDB51(downloadFolder);
```

After the download is complete, extract the RAR file `hmdb51_org.rar` to the `hmdb51` folder. Next, use the `checkForHMDB51Folder` supporting function, listed at the end of this example, to confirm that the downloaded and extracted files are in place.

```
allClasses = checkForHMDB51Folder(downloadFolder);
```

The data set contains about 2 GB of video data for 7000 clips over 51 classes, such as *drink*, *run*, and *shake hands*. Each video frame has a height of 240 pixels and a minimum width of 176 pixels. The number of frames ranges from 18 to approximately 1000.

To reduce training time, this example trains an activity recognition network to classify 5 action classes instead of all 51 classes in the data set. Set `useAllData` to `true` to train with all 51 classes.

```
useAllData = false;

if useAllData
    classes = allClasses;
else
    classes = ["kiss","laugh","pick","pour","pushup"];
end
dataFolder = fullfile(downloadFolder, "hmdb51_org");
```

Split the data set into a training set for training the network, and a test set for evaluating the network. Use 80% of the data for the training set and the rest for the test set. Use `imageDatastore` to split the data based on each label into training and test data sets by randomly selecting a proportion of files from each label.

```
imds = imageDatastore(fullfile(dataFolder,classes),...
    'IncludeSubfolders', true,...
```

```
'LabelSource', 'foldernames', ...  
'FileExtensions', '.avi');  
  
[trainImds,testImds] = splitEachLabel(imds,0.8,'randomized');  
  
trainFileNames = trainImds.Files;  
testFileNames  = testImds.Files;
```

To normalize the input data for the network, the minimum and maximum values for the data set are provided in the MAT file `inputStatistics.mat`, attached to this example. To find the minimum and maximum values for a different data set, use the `inputStatistics` supporting function, listed at the end of this example.

```
inputStatsFilename = 'inputStatistics.mat';  
if ~exist(inputStatsFilename, 'file')  
    disp("Reading all the training data for input statistics...")  
    inputStats = inputStatistics(dataFolder);  
else  
    d = load(inputStatsFilename);  
    inputStats = d.inputStats;  
end
```

Create Datastores for Training Networks

Create two `FileDatastore` objects for training and validation by using the `createFileDatastore` supporting function, defined at the end of this example. Each datastore reads a video file to provide RGB data, optical flow data, and the corresponding label information.

Specify the number of frames for each read by the datastore. Typical values are 16, 32, 64, or 128. Using more frames helps capture more temporal information, but requires more memory for training and prediction. Set the number of frames to 64 to balance memory usage against performance. You might need to lower this value depending on your system resources.

```
numFrames = 64;
```

Specify the height and width of the frames for the datastore to read. Fixing the height and width to the same value makes batching data for the network easier. Typical values are [112, 112], [224, 224], and [256, 256]. The minimum height and width of the video frames in the HMDB51 data set are 240 and 176, respectively. Specify [112, 112] to capture a larger number of frames at the cost of spatial information. If you want to specify a frame size for the datastore to read that is larger than the minimum values, such as [256, 256], first resize the frames using `imresize`.

```
frameSize = [112,112];
```

Set `inputSize` to the `inputStats` structure so the read function of `fileDatastore` can read the specified input size.

```
inputSize = [frameSize, numFrames];  
inputStats.inputSize = inputSize;  
inputStats.Classes = classes;
```

Create two `FileDatastore` objects, one for training and another for validation.

```
isDataForValidation = false;  
dsTrain = createFileDatastore(trainFileNames,inputStats,isDataForValidation);  
  
isDataForValidation = true;
```

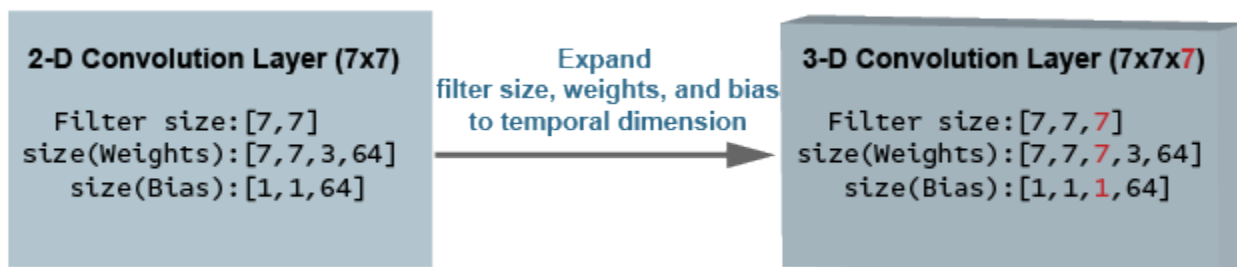
```
dsVal = createFileDatastore(testFileNames,inputStats,isDataForValidation);
disp("Training data size: " + string(numel(dsTrain.Files)))
Training data size: 436
disp("Validation data size: " + string(numel(dsVal.Files)))
Validation data size: 109
```

Define Network Architecture

I3D network

Using a 3-D CNN is a natural approach to extracting spatio-temporal features from videos. You can create an I3D network from a pretrained 2-D image classification network such as Inception v1 or ResNet-50 by expanding 2-D filters and pooling kernels into 3-D. This procedure reuses the weights learned from the image classification task to bootstrap the video recognition task.

The following figure is a sample showing how to inflate a 2-D convolution layer to a 3-D convolution layer. The inflation involves expanding the filter size, weights, and bias by adding a third dimension (the temporal dimension).

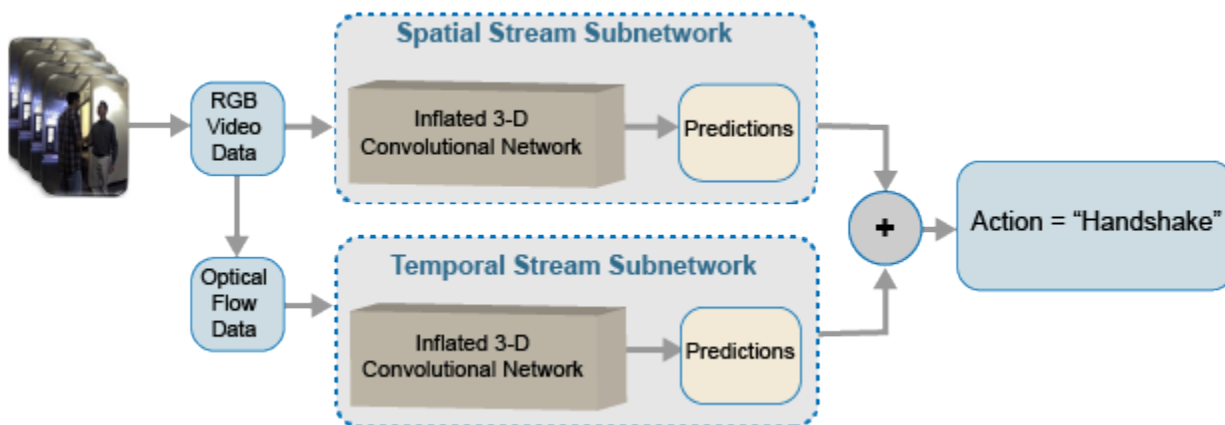


Two-Stream I3D Network

Video data can be considered to have two parts: a spatial component and a temporal component.

- The spatial component comprises information about the shape, texture, and color of objects in video. RGB data contains this information.
- The temporal component comprises information about the motion of objects across the frames and depicts important movements between the camera and the objects in a scene. Computing optical flow is a common technique for extracting temporal information from video.

A two-stream CNN incorporates a spatial subnetwork and a temporal subnetwork [2] on page 3-0 . A convolutional neural network trained on dense optical flow and a video data stream can achieve better performance with limited training data than with raw stacked RGB frames. The following illustration shows a typical two-stream I3D network.



Create Two-Stream I3D Network

In this example, you create an I3D network using GoogLeNet, a network pretrained on the ImageNet database.

Specify the number of channels as 3 for the RGB subnetwork, and 2 for the optical flow subnetwork. The two channels for optical flow data are the x and y components of velocity, V_x and V_y , respectively.

```
rgbChannels = 3;
flowChannels = 2;
```

Obtain the minimum and maximum values for the RGB and optical flow data from the `inputStats` structure loaded from the `inputStatistics.mat` file. These values are needed for the `image3dInputLayer` of the I3D networks to normalize the input data.

```
rgbInputSize = [frameSize, numFrames, rgbChannels];
flowInputSize = [frameSize, numFrames, flowChannels];
```

```
rgbMin = inputStats.rgbMin;
rgbMax = inputStats.rgbMax;
oflowMin = inputStats.oflowMin(:, :, 1:2);
oflowMax = inputStats.oflowMax(:, :, 1:2);
```

```
rgbMin = reshape(rgbMin, [1, size(rgbMin)]);
rgbMax = reshape(rgbMax, [1, size(rgbMax)]);
oflowMin = reshape(oflowMin, [1, size(oflowMin)]);
oflowMax = reshape(oflowMax, [1, size(oflowMax)]);
```

Specify the number of classes for training the network.

```
numClasses = numel(classes);
```

Create the I3D RGB and optical flow subnetworks by using the `Inflated3D` supporting function, which is attached to this example. The subnetworks are created from GoogLeNet.

```
cnnNet = googlenet;

netRGB = Inflated3D(numClasses, rgbInputSize, rgbMin, rgbMax, cnnNet);
netFlow = Inflated3D(numClasses, flowInputSize, oflowMin, oflowMax, cnnNet);
```

Create a `dlNetwork` object from the layer graph of each of the I3D networks.


```
dlnetRGB = dlnetwork(netRGB);
dlnetFlow = dlnetwork(netFlow);
```

Define Model Gradients Function

Create the supporting function `modelGradients`, listed at the end of this example. The `modelGradients` function takes as input the RGB subnetwork `dlnetRGB`, the optical flow subnetwork `dlnetFlow`, a mini-batch of input data `dIRGB` and `dIFlow`, and a mini-batch of ground truth label data `dLY`. The function returns the training loss value, the gradients of the loss with respect to the learnable parameters of the respective subnetworks, and the mini-batch accuracy of the subnetworks.

The loss is calculated by computing the average of the cross-entropy losses of the predictions from each of the subnetworks. The output predictions of the network are probabilities between 0 and 1 for each of the classes.

```
rgbLoss = crossentropy(rgbPrediction)

flowLoss = crossentropy(flowPrediction)

loss = mean([rgbLoss, flowLoss])
```

The accuracy of each of the subnetworks is calculated by taking the average of the RGB and optical flow predictions, and comparing it to the ground truth label of the inputs.

Specify Training Options

Train with a mini-batch size of 20 for 1500 iterations. Specify the iteration after which to save the model with the best validation accuracy by using the `SaveBestAfterIteration` parameter.

Specify the cosine-annealing learning rate schedule [3 on page 3-0] parameters. For both networks, use:

- A minimum learning rate of $1e-4$.
- A maximum learning rate of $1e-3$.
- Cosine number of iterations of 300, 500, and 700, after which the learning rate schedule cycle restarts. The option `CosineNumIterations` defines the width of each cosine cycle.

Specify the parameters for SGDM optimization. Initialize the SGDM optimization parameters at the beginning of the training for each of the RGB and optical flow networks. For both networks, use:

- A momentum of 0.9.
- An initial velocity parameter initialized as `[]`.
- An L2 regularization factor of 0.0005.

Specify to dispatch the data in the background using a parallel pool. If `DispatchInBackground` is set to true, open a parallel pool with the specified number of parallel workers, and create a `DispatchInBackgroundDatastore`, provided as part of this example, that dispatches the data in the background to speed up training using asynchronous data loading and preprocessing. By default, this example uses a GPU if one is available. Otherwise, it uses a CPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
params.Classes = classes;
params.MinibatchSize = 20;
```

```
params.NumIterations = 1500;
params.SaveBestAfterIteration = 900;
params.CosineNumIterations = [300, 500, 700];
params.MinLearningRate = 1e-4;
params.MaxLearningRate = 1e-3;
params.Momentum = 0.9;
params.VelocityRGB = [];
params.VelocityFlow = [];
params.L2Regularization = 0.0005;
params.ProgressPlot = false;
params.Verbose = true;
params.ValidationData = dsVal;
params.DispatchInBackground = false;
params.NumWorkers = 4;
```

Train Network

Train the subnetworks using the RGB data and optical flow data. Set the `doTraining` variable to `false` to download the pretrained subnetworks without having to wait for training to complete. Alternatively, if you want to train the subnetworks, set the `doTraining` variable to `true`.

```
doTraining = false;
```

For each epoch:

- Shuffle the data before looping over mini-batches of data.
- Use `minibatchqueue` to loop over the mini-batches. The supporting function `createMiniBatchQueue`, listed at the end of this example, uses the given training datastore to create a `minibatchqueue`.
- Use the validation data `dsVal` to validate the networks.
- Display the loss and accuracy results for each epoch using the supporting function `displayVerboseOutputEveryEpoch`, listed at the end of this example.

For each mini-batch:

- Convert the image data or optical flow data and the labels to `darray` objects with the underlying type `single`.
- Treat the temporal dimension of the the video and optical flow data as one of the spatial dimensions to enable processing using a 3-D CNN. Specify the dimension labels "SSSCB" (spatial, spatial, spatial, channel, batch) for the RGB or optical flow data, and "CB" for the label data.

The `minibatchqueue` object uses the supporting function `batchRGBAndFlow`, listed at the end of this example, to batch the RGB and optical flow data.

```
modelFilename = "I3D-RGBFlow-" + numClasses + "Classes-hmdb51.mat";
if doTraining
    epoch = 1;
    bestValAccuracy = 0;
    accTrain = [];
    accTrainRGB = [];
    accTrainFlow = [];
    lossTrain = [];

    iteration = 1;
    shuffled = shuffleTrainDs(dsTrain);
```

```

% Number of outputs is three: One for RGB frames, one for optical flow
% data, and one for ground truth labels.
numOutputs = 3;
mbq = createMiniBatchQueue(shuffled, numOutputs, params);
start = tic;
trainTime = start;

% Use the initializeTrainingProgressPlot and initializeVerboseOutput
% supporting functions, listed at the end of the example, to initialize
% the training progress plot and verbose output to display the training
% loss, training accuracy, and validation accuracy.
plotters = initializeTrainingProgressPlot(params);
initializeVerboseOutput(params);

while iteration <= params.NumIterations

    % Iterate through the data set.
    [dLX1,dLX2,dLY] = next(mbq);

    % Evaluate the model gradients and loss using dlfeval.
    [gradRGB,gradFlow,loss,acc,accRGB,accFlow,stateRGB,stateFlow] = ...
        dlfeval(@modelGradients,dlnetRGB,dlnetFlow,dLX1,dLX2,dLY);

    % Accumulate the loss and accuracies.
    lossTrain = [lossTrain, loss];
    accTrain = [accTrain, acc];
    accTrainRGB = [accTrainRGB, accRGB];
    accTrainFlow = [accTrainFlow, accFlow];
    % Update the network state.
    dlnetRGB.State = stateRGB;
    dlnetFlow.State = stateFlow;

    % Update the gradients and parameters for the RGB and optical flow
    % subnetworks using the SGDM optimizer.
    [dlnetRGB,gradRGB,params.VelocityRGB,learnRate] = ...
        updateDlNetwork(dlnetRGB,gradRGB,params,params.VelocityRGB,iteration);
    [dlnetFlow,gradFlow,params.VelocityFlow] = ...
        updateDlNetwork(dlnetFlow,gradFlow,params,params.VelocityFlow,iteration);

    if ~hasdata(mbq) || iteration == params.NumIterations
        % Current epoch is complete. Do validation and update progress.
        trainTime = toc(trainTime);

        [validationTime,cmat,lossValidation,accValidation,accValidationRGB,accValidationFlow, ...] =
            doValidation(params, dlnetRGB, dlnetFlow);

        % Update the training progress.
        displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
            mean(accTrain),mean(accTrainRGB),mean(accTrainFlow),...
            accValidation,accValidationRGB,accValidationFlow,...
            mean(lossTrain),lossValidation,trainTime,validationTime);
        updateProgressPlot(params,plotters,epoch,iteration,start,mean(lossTrain),mean(accTrain));

        % Save model with the trained dlnetwork and accuracy values.
        % Use the saveData supporting function, listed at the
        % end of this example.
        if iteration >= params.SaveBestAfterIteration

```

```
        if accValidation > bestValAccuracy
            bestValAccuracy = accValidation;
            saveData(modelFilename, dlnetRGB, dlnetFlow, cmat, accValidation);
        end
    end
end

if ~hasdata(mbq) && iteration < params.NumIterations
    % Current epoch is complete. Initialize the training loss, accuracy
    % values, and minibatchqueue for the next epoch.
    accTrain = [];
    accTrainRGB = [];
    accTrainFlow = [];
    lossTrain = [];

    trainTime = tic;
    epoch = epoch + 1;
    shuffled = shuffleTrainDs(dsTrain);
    numOutputs = 3;
    mbq = createMiniBatchQueue(shuffled, numOutputs, params);

end

    iteration = iteration + 1;
end

% Display a message when training is complete.
endVerboseOutput(params);

disp("Model saved to: " + modelFilename);
end

% Download the pretrained model and video file for prediction.
filename = "activityRecognition-I3D-HMDB51.zip";
downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + filename;

filename = fullfile(downloadFolder, filename);
if ~exist(filename, 'file')
    disp('Downloading the pretrained network...');
    websave(filename, downloadURL);
end

% Unzip the contents to the download folder.
unzip(filename, downloadFolder);
if ~doTraining
    modelFilename = fullfile(downloadFolder, modelFilename);
end
```

Evaluate Trained Network

Use the test data set to evaluate the accuracy of the trained subnetworks.

Load the best model saved during training.

```
d = load(modelFilename);
dlnetRGB = d.data.dlnetRGB;
dlnetFlow = d.data.dlnetFlow;
```

Create a minibatchqueue object to load batches of the test data.

```
numOutputs = 3;
mbq = createMiniBatchQueue(params.ValidationData, numOutputs, params);
```

For each batch of test data, make predictions using the RGB and optical flow subnetworks, take the average of the predictions, and compute the prediction accuracy using a confusion matrix.

```
cmat = sparse(numClasses,numClasses);
while hasdata(mbq)
    [dlRGB, dlFlow, dlY] = next(mbq);

    % Pass the video input as RGB and optical flow data through the
    % two-stream subnetworks to get the separate predictions.
    dlYPredRGB = predict(dlnetRGB,dlRGB);
    dlYPredFlow = predict(dlnetFlow,dlFlow);

    % Fuse the predictions by calculating the average of the predictions.
    dlYPred = (dlYPredRGB + dlYPredFlow)/2;

    % Calculate the accuracy of the predictions.
    [~,YTest] = max(dlY,[],1);
    [~,YPred] = max(dlYPred,[],1);

    cmat = aggregateConfusionMetric(cmat,YTest,YPred);
end
```

Compute the average classification accuracy for the trained subnetworks.

```
accuracyEval = sum(diag(cmat))./sum(cmat,"all")
accuracyEval =
    0.60909
```

Display the confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```

	kiss	21	4	1	15	1
	laugh	12	33	1	14	7
	pick	2	3	6	3	9
True Class	pour				61	3
	pushup		7	3	1	13
		kiss	laugh	pick	pour	pushup
		Predicted Class				

Due to the limited number of training samples, increasing the accuracy beyond 61% is challenging. To improve the robustness of the network, additional training with a large data set is required. In addition, pretraining on a larger data set, such as Kinetics [1] on page 3-0 , can help improve results.

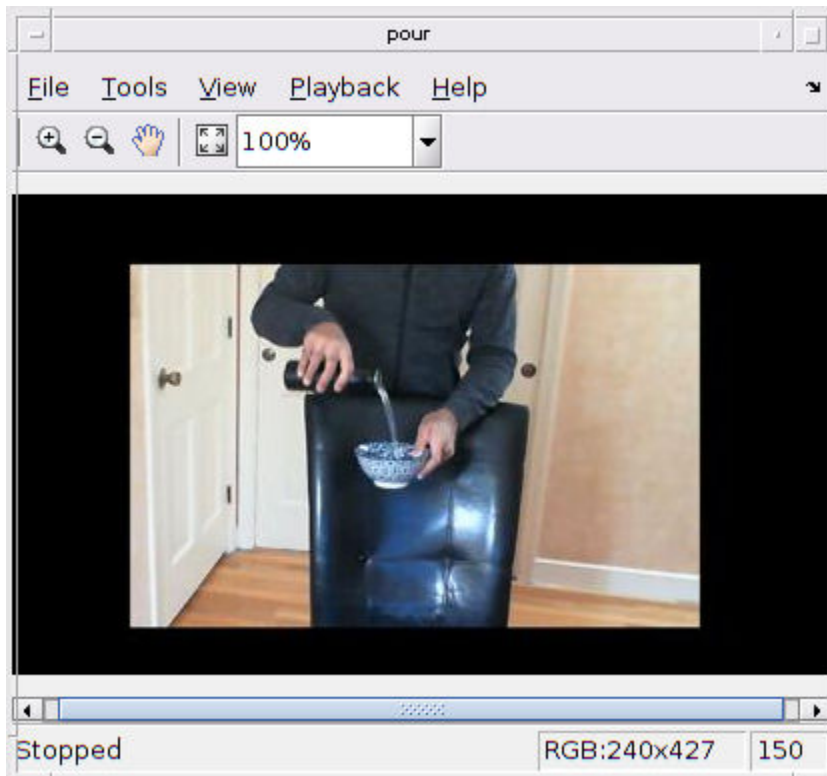
Predict Using New Video

You can now use the trained subnetworks to predict actions in new videos. Read and display the video `pour.avi` using `VideoReader` and `vision.VideoPlayer`.

```
videoFilename = fullfile(downloadFolder, "pour.avi");

videoReader = VideoReader(videoFilename);
videoPlayer = vision.VideoPlayer;
videoPlayer.Name = "pour";

while hasFrame(videoReader)
    frame = readFrame(videoReader);
    step(videoPlayer, frame);
end
release(videoPlayer);
```



Use the `readRGBAndFlow` supporting function, listed at the end of this example, to read the RGB and optical flow data.

```
isDataForValidation = true;
readFcn = @(f,u) readRGBAndFlow(f,u,inputStats,isDataForValidation);
```

The read function returns a logical `isDone` value that indicates whether there is more data to read from the file. Use the `batchRGBAndFlow` supporting function, defined at the end of this example, to batch the data to pass through the two-stream subnetworks to obtain the predictions.

```
hasdata = true;
userdata = [];
YPred = [];
while hasdata
    [data,userdata,isDone] = readFcn(videoFilename,userdata);

    [dLRGB, dLFlow] = batchRGBAndFlow(data(:,1),data(:,2),data(:,3));

    % Pass video input as RGB and optical flow data through the two-stream
    % subnetworks to get the separate predictions.
    dLYPredRGB = predict(dLnetRGB,dLRGB);
    dLYPredFlow = predict(dLnetFlow,dLFlow);

    % Fuse the predictions by calculating the average of the predictions.
    dLYPred = (dLYPredRGB + dLYPredFlow)/2;
    [~,YPredCurr] = max(dLYPred,[],1);
    YPred = horzcat(YPred,YPredCurr);
    hasdata = ~isDone;
end
YPred = extractdata(YPred);
```

Count the number of correct predictions using `histcounts`, and obtain the predicted action using the maximum number of correct predictions.

```
classes = params.Classes;
counts = histcounts(YPred,1:numel(classes));
[~,clsIdx] = max(counts);
action = classes(clsIdx)

action =
"pour"
```

Supporting Functions

inputStatistics

The `inputStatistics` function takes as input the name of the folder containing the HMDB51 data, and calculates the minimum and maximum values for the RGB data and the optical flow data. The minimum and maximum values are used as normalization inputs to the input layer of the networks. This function also obtains the number of frames in each of the video files to use later during training and testing the network. In order to find the minimum and maximum values for a different data set, use this function with a folder name containing the data set.

```
function inputStats = inputStatistics(dataFolder)
    ds = createDatastore(dataFolder);
    ds.ReadFcn = @getMinMax;

    tic;
    tt = tall(ds);
    varnames = {'rgbMax','rgbMin','oflowMax','oflowMin'};
    stats = gather(groupsummary(tt,[],{'max','min'}, varnames));
    inputStats.FileName = gather(tt.FileName);
    inputStats.NumFrames = gather(tt.NumFrames);
    inputStats.rgbMax = stats.max_rgbMax;
    inputStats.rgbMin = stats.min_rgbMin;
    inputStats.oflowMax = stats.max_oflowMax;
    inputStats.oflowMin = stats.min_oflowMin;
    save('inputStatistics.mat','inputStats');
    toc;
end

function data = getMinMax(filename)
    reader = VideoReader(filename);
    opticFlow = opticalFlowFarneback;
    data = [];
    while hasFrame(reader)
        frame = readFrame(reader);
        [rgb,oflow] = findMinMax(frame,opticFlow);
        data = assignMinMax(data, rgb, oflow);
    end

    totalFrames = floor(reader.Duration * reader.FrameRate);
    totalFrames = min(totalFrames, reader.NumFrames);

    [labelName, filename] = getLabelFilename(filename);
    data.FileName = fullfile(labelName, filename);
    data.NumFrames = totalFrames;

    data = struct2table(data,'AsArray',true);
```



```

end

function data = assignMinMax(data, rgb, oflow)
    if isempty(data)
        data.rgbMax = rgb.Max;
        data.rgbMin = rgb.Min;
        data.oflowMax = oflow.Max;
        data.oflowMin = oflow.Min;
        return;
    end
    data.rgbMax = max(data.rgbMax, rgb.Max);
    data.rgbMin = min(data.rgbMin, rgb.Min);

    data.oflowMax = max(data.oflowMax, oflow.Max);
    data.oflowMin = min(data.oflowMin, oflow.Min);
end

function [rgbMinMax,oflowMinMax] = findMinMax(rgb, opticFlow)
    rgbMinMax.Max = max(rgb,[],[1,2]);
    rgbMinMax.Min = min(rgb,[],[1,2]);

    gray = rgb2gray(rgb);
    flow = estimateFlow(opticFlow,gray);
    oflow = cat(3,flow.Vx,flow.Vy,flow.Magnitude);

    oflowMinMax.Max = max(oflow,[],[1,2]);
    oflowMinMax.Min = min(oflow,[],[1,2]);
end

function ds = createDatastore(folder)
    ds = fileDatastore(folder,...
        'IncludeSubfolders', true,...
        'FileExtensions', '.avi',...
        'UniformRead', true,...
        'ReadFcn', @getMinMax);
    disp("NumFiles: " + numel(ds.Files));
end

```

createFileDatastore

The `createFileDatastore` function creates a `FileDatastore` object using the given file names. The `FileDatastore` object reads the data in 'partialfile' mode, so every read can return partially read frames from videos. This feature helps with reading large video files, if all of the frames do not fit in memory.

```

function datastore = createFileDatastore(filenamees,inputStats,isDataForValidation)
    readFcn = @(f,u)readRGBAndFlow(f,u,inputStats,isDataForValidation);
    datastore = fileDatastore(filenamees,...
        'ReadFcn',readFcn,...
        'ReadMode','partialfile');
end

```

readRGBAndFlow

The `readRGBAndFlow` function reads RGB frames, the corresponding optical flow data, and the label values for a given video file. During training, the read function reads the specific number of frames as per the network input size, with a randomly chosen starting frame. Optical flow data is calculated from the beginning of the video file, but skipped until the starting frame is reached. During testing,

all the frames are sequentially read, and corresponding optical flow data is calculated. The RGB frames and optical flow data are randomly cropped to the required network input size for training, and center cropped for testing and validation.

```
function [data,userdata,done] = readRGBAndFlow(filename,userdata,inputStats,isDataForValidation)
    if isempty(userdata)
        userdata.reader      = VideoReader(filename);
        userdata.batchesRead = 0;
        userdata.opticalFlow = opticalFlowFarneback;

        [totalFrames,userdata.label] = getTotalFramesAndLabel(inputStats,filename);
        if isempty(totalFrames)
            totalFrames = floor(userdata.reader.Duration * userdata.reader.FrameRate);
            totalFrames = min(totalFrames, userdata.reader.NumFrames);
        end
        userdata.totalFrames = totalFrames;
    end
    reader      = userdata.reader;
    totalFrames = userdata.totalFrames;
    label       = userdata.label;
    batchesRead = userdata.batchesRead;
    opticalFlow = userdata.opticalFlow;

    inputSize = inputStats.inputSize;
    H = inputSize(1);
    W = inputSize(2);
    rgbC = 3;
    flowC = 2;
    numFrames = inputSize(3);

    if numFrames > totalFrames
        numBatches = 1;
    else
        numBatches = floor(totalFrames/numFrames);
    end

    imH = userdata.reader.Height;
    imW = userdata.reader.Width;
    imsz = [imH,imW];

    if ~isDataForValidation

        augmentFcn = augmentTransform([imsz,3]);
        cropWindow = randomCropWindow2d(imsz, inputSize(1:2));
        % 1. Randomly select required number of frames,
        %    starting randomly at a specific frame.
        if numFrames >= totalFrames
            idx = 1:totalFrames;
            % Add more frames to fill in the network input size.
            additional = ceil(numFrames/totalFrames);
            idx = repmat(idx,1,additional);
            idx = idx(1:numFrames);
        else
            startIdx = randperm(totalFrames - numFrames);
            startIdx = startIdx(1);
            endIdx = startIdx + numFrames - 1;
            idx = startIdx:endIdx;
        end
    end
end
```

```

video = zeros(H,W,rgbC,numFrames);
oflow = zeros(H,W,flowC,numFrames);
i = 1;
% Discard the first set of frames to initialize the optical flow.
for ii = 1:idx(1)-1
    frame = read(reader,ii);
    getRGBAndFlow(frame,opticalFlow,augmentFcn,cropWindow);
end
% Read the next set of required number of frames for training.
for ii = idx
    frame = read(reader,ii);
    [rgb,vxvy] = getRGBAndFlow(frame,opticalFlow,augmentFcn,cropWindow);
    video(:,:,:,i) = rgb;
    oflow(:,:,:,i) = vxvy;
    i = i + 1;
end
else
    augmentFcn = @(data)(data);
    cropWindow = centerCropWindow2d(imsz, inputSize(1:2));
    toRead = min([numFrames,totalFrames]);
    video = zeros(H,W,rgbC,toRead);
    oflow = zeros(H,W,flowC,toRead);
    i = 1;
    while hasFrame(reader) && i <= numFrames
        frame = readFrame(reader);
        [rgb,vxvy] = getRGBAndFlow(frame,opticalFlow,augmentFcn,cropWindow);
        video(:,:,:,i) = rgb;
        oflow(:,:,:,i) = vxvy;
        i = i + 1;
    end
    if numFrames > totalFrames
        additional = ceil(numFrames/totalFrames);
        video = repmat(video,1,1,1,additional);
        oflow = repmat(oflow,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
        oflow = oflow(:,:,:,1:numFrames);
    end
end
end

% The network expects the video and optical flow input in
% the following darray format:
% "SSSCB" ==> Height x Width x Frames x Channels x Batch
%
% Permute the data
% from
%     Height x Width x Channels x Frames
% to
%     Height x Width x Frames x Channels
video = permute(video, [1,2,4,3]);
oflow = permute(oflow, [1,2,4,3]);

data = {video, oflow, label};

batchesRead = batchesRead + 1;

userdata.batchesRead = batchesRead;

```

```

    % Set the done flag to true, if the reader has read all the frames or
    % if it is training.
    done = batchesRead == numBatches || ~isDataForValidation;
end

function [rgb,vxvy] = getRGBAndFlow(rgb,opticalFlow,augmentFcn,cropWindow)
    rgb = augmentFcn(rgb);
    gray = rgb2gray(rgb);
    flow = estimateFlow(opticalFlow,gray);
    vxvy = cat(3,flow.Vx,flow.Vy,flow.Vy);

    rgb = imcrop(rgb, cropWindow);
    vxvy = imcrop(vxvy, cropWindow);
    vxvy = vxvy(:,:,1:2);
end

function [label,fname] = getLabelFilename(filename)
    [folder,name,ext] = fileparts(string(filename));
    [~,label] = fileparts(folder);
    fname = name + ext;
    label = string(label);
    fname = string(fname);
end

function [totalFrames,label] = getTotalFramesAndLabel(info, filename)
    filenames = info.Filename;
    frames = info.NumFrames;
    [labelName, fname] = getLabelFilename(filename);
    idx = strcmp(filenames, fullfile(labelName,fname));
    totalFrames = frames(idx);
    label = categorical(string(labelName), string(info.Classes));
end

```

augmentTransform

The `augmentTransform` function creates an augmentation method with random left-right flipping and scaling factors.

```

function augmentFcn = augmentTransform(sz)
% Randomly flip and scale the image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');

augmentFcn = @(data)augmentData(data,tform,rout);

function data = augmentData(data,tform,rout)
    data = imwarp(data,tform,'OutputView',rout);
end
end

```

modelGradients

The `modelGradients` function takes as input a mini-batch of RGB data `dLRGB`, the corresponding optical flow data `dLFlow`, and the corresponding target `dLY`, and returns the corresponding loss, the gradients of the loss with respect to the learnable parameters, and the training accuracy. To compute the gradients, evaluate the `modelGradients` function using the `dLfeval` function in the training loop.

```

function [gradientsRGB,gradientsFlow,loss,acc,accRGB,accFlow,stateRGB,stateFlow] = modelGradient...

% Pass video input as RGB and optical flow data through the two-stream
% network.
[dLYPredRGB,stateRGB] = forward(dlnetRGB,dlRGB);
[dLYPredFlow,stateFlow] = forward(dlnetFlow,dlFlow);

% Calculate fused loss, gradients, and accuracy for the two-stream
% predictions.
rgbLoss = crossentropy(dLYPredRGB,Y);
flowLoss = crossentropy(dLYPredFlow,Y);
% Fuse the losses.
loss = mean([rgbLoss,flowLoss]);

gradientsRGB = dlgradient(loss,dlnetRGB.Learnables);
gradientsFlow = dlgradient(loss,dlnetFlow.Learnables);

% Fuse the predictions by calculating the average of the predictions.
dLYPred = (dLYPredRGB + dLYPredFlow)/2;

% Calculate the accuracy of the predictions.
[~,YTest] = max(Y,[],1);
[~,YPred] = max(dLYPred,[],1);

acc = gather(extractdata(sum(YTest == YPred)./numel(YTest)));

% Calculate the accuracy of the RGB and flow predictions.
[~,YTest] = max(Y,[],1);
[~,YPredRGB] = max(dLYPredRGB,[],1);
[~,YPredFlow] = max(dLYPredFlow,[],1);

accRGB = gather(extractdata(sum(YTest == YPredRGB)./numel(YTest)));
accFlow = gather(extractdata(sum(YTest == YPredFlow)./numel(YTest)));
end

```

doValidation

The doValidation function validates the network using the validation data.

```

function [validationTime, cmat, lossValidation, accValidation, accValidationRGB, accValidationFlow] = doValidation(params,dlRGB,dlFlow,YTest)

validationTime = tic;

numOutputs = 3;
mbq = createMiniBatchQueue(params.ValidationData, numOutputs, params);

lossValidation = [];
numClasses = numel(params.Classes);
cmat = sparse(numClasses,numClasses);
cmatRGB = sparse(numClasses,numClasses);
cmatFlow = sparse(numClasses,numClasses);
while hasdata(mbq)

    [dlX1,dlX2,dlY] = next(mbq);

    [loss,YTest,YPred,YPredRGB,YPredFlow] = predictValidation(dlnetRGB,dlnetFlow,dlX1,dlX2,dY);
end

```

```

        lossValidation = [lossValidation,loss];
        cmat = aggregateConfusionMetric(cmat,YTest,YPred);
        cmatRGB = aggregateConfusionMetric(cmatRGB,YTest,YPredRGB);
        cmatFlow = aggregateConfusionMetric(cmatFlow,YTest,YPredFlow);
    end
    lossValidation = mean(lossValidation);
    accValidation = sum(diag(cmat))./sum(cmat,"all");
    accValidationRGB = sum(diag(cmatRGB))./sum(cmatRGB,"all");
    accValidationFlow = sum(diag(cmatFlow))./sum(cmatFlow,"all");

    validationTime = toc(validationTime);
end

```

predictValidation

The predictValidation function calculates the loss and prediction values using the provided dlnetwork objects for RGB and optical flow data.

```

function [loss,YTest,YPred,YPredRGB,YPredFlow] = predictValidation(dlNetRGB,dlNetFlow,dLRGB,dLFlow)

% Pass the video input through the two-stream
% network.
dLYPredRGB = predict(dlNetRGB,dLRGB);
dLYPredFlow = predict(dlNetFlow,dLFlow);

% Calculate the cross-entropy separately for the two-stream
% outputs.
rgbLoss = crossentropy(dLYPredRGB,Y);
flowLoss = crossentropy(dLYPredFlow,Y);

% Fuse the losses.
loss = mean([rgbLoss,flowLoss]);

% Fuse the predictions by calculating the average of the predictions.
dLYPred = (dLYPredRGB + dLYPredFlow)/2;

% Calculate the accuracy of the predictions.
[~,YTest] = max(Y,[],1);
[~,YPred] = max(dLYPred,[],1);

[~,YPredRGB] = max(dLYPredRGB,[],1);
[~,YPredFlow] = max(dLYPredFlow,[],1);

end

```

updateDlNetwork

The updateDlNetwork function updates the provided dlNetwork object with gradients and other parameters using SGDM optimization function sgdmupdate.

```

function [dlNet,gradients,velocity,learnRate] = updateDlNetwork(dlNet,gradients,params,velocity,
    % Determine the learning rate using the cosine-annealing learning rate schedule.
    learnRate = cosineAnnealingLearnRate(iteration, params);

    % Apply L2 regularization to the weights.
    idx = dlNet.Learnables.Parameter == "Weights";
    gradients(idx,:) = dlupdate(@(g,w) g + params.L2Regularization*w, gradients(idx,:), dlNet.Lea

```

```

    % Update the network parameters using the SGDM optimizer.
    [dlnet, velocity] = sgdmupdate(dlnet, gradients, velocity, learnRate, params.Momentum);
end

```

cosineAnnealingLearnRate

The `cosineAnnealingLearnRate` function computes the learning rate based on the current iteration number, minimum learning rate, maximum learning rate, and number of iterations for annealing [3 on page 3-0].

```

function lr = cosineAnnealingLearnRate(iteration, params)
    if iteration == params.NumIterations
        lr = params.MinLearningRate;
        return;
    end
    cosineNumIter = [0, params.CosineNumIterations];
    csum = cumsum(cosineNumIter);
    block = find(csum >= iteration, 1, 'first');
    cosineIter = iteration - csum(block - 1);
    annealingIteration = mod(cosineIter, cosineNumIter(block));
    cosineIteration = cosineNumIter(block);
    minR = params.MinLearningRate;
    maxR = params.MaxLearningRate;
    cosMult = 1 + cos(pi * annealingIteration / cosineIteration);
    lr = minR + ((maxR - minR) * cosMult / 2);
end

```

aggregateConfusionMetric

The `aggregateConfusionMetric` function incrementally fills a confusion matrix based on the predicted results `YPred` and the expected results `YTest`.

```

function cmat = aggregateConfusionMetric(cmat, YTest, YPred)
YTest = gather(extractdata(YTest));
YPred = gather(extractdata(YPred));
[m,n] = size(cmat);
cmat = cmat + full(sparse(YTest, YPred, 1, m, n));
end

```

createMiniBatchQueue

The `createMiniBatchQueue` function creates a `minibatchqueue` object that provides `miniBatchSize` amount of data from the given datastore. It also creates a `DispatchInBackgroundDatastore` if a parallel pool is open.

```

function mbq = createMiniBatchQueue(datastore, numOutputs, params)
if params.DispatchInBackground && isempty(gcp('nocreate'))
    % Start a parallel pool, if DispatchInBackground is true, to dispatch
    % data in the background using the parallel pool.
    c = parcluster('local');
    c.NumWorkers = params.NumWorkers;
    parpool('local', params.NumWorkers);
end
p = gcp('nocreate');
if ~isempty(p)
    datastore = DispatchInBackgroundDatastore(datastore, p.NumWorkers);
end
end

```

```
inputFormat(1:numOutputs-1) = "SSSCB";
outputFormat = "CB";
mbq = minibatchqueue(datastore, numOutputs, ...
    "MiniBatchSize", params.MinibatchSize, ...
    "MiniBatchFcn", @batchRGBAndFlow, ...
    "MiniBatchFormat", [inputFormat,outputFormat]);
end
```

batchRGBAndFlow

The `batchRGBAndFlow` function batches the image, flow, and label data into corresponding `darray` values in the data formats "SSSCB", "SSSCB", and "CB", respectively.

```
function [dLX1,dLX2,dLY] = batchRGBAndFlow(images, flows, labels)
% Batch dimension: 5
X1 = cat(5,images{:});
X2 = cat(5,flows{:});

% Batch dimension: 2
labels = cat(2,labels{:});

% Feature dimension: 1
Y = onehotencode(labels,1);

% Cast data to single for processing.
X1 = single(X1);
X2 = single(X2);
Y = single(Y);

% Move data to the GPU if possible.
if canUseGPU
    X1 = gpuArray(X1);
    X2 = gpuArray(X2);
    Y = gpuArray(Y);
end

% Return X and Y as darray objects.
dLX1 = darray(X1,"SSSCB");
dLX2 = darray(X2,"SSSCB");
dLY = darray(Y,"CB");
end
```

shuffleTrainDs

The `shuffleTrainDs` function shuffles the files present in the training datastore `dsTrain`.

```
function shuffled = shuffleTrainDs(dsTrain)
shuffled = copy(dsTrain);
n = numel(shuffled.Files);
shuffledIndices = randperm(n);
shuffled.Files = shuffled.Files(shuffledIndices);
reset(shuffled);
end
```

saveData

The `saveData` function saves the given `dlnetwork` objects and accuracy values to a MAT file.


```
function saveData(modelFilename, dlnetRGB, dlnetFlow, cmat, accValidation)
dlnetRGB = gatherFromGPUSave(dlnetRGB);
dlnetFlow = gatherFromGPUSave(dlnetFlow);
data.ValidationAccuracy = accValidation;
data.cmat = cmat;
data.dlnetRGB = dlnetRGB;
data.dlnetFlow = dlnetFlow;
save(modelFilename, 'data');
end
```

gatherFromGPUSave

The gatherFromGPUSave function gathers data from the GPU in order to save the model to disk.

```
function dlnet = gatherFromGPUSave(dlnet)
if ~canUseGPU
    return;
end
dlnet.Learnables = gatherValues(dlnet.Learnables);
dlnet.State = gatherValues(dlnet.State);
function tbl = gatherValues(tbl)
    for ii = 1:height(tbl)
        tbl.Value{ii} = gather(tbl.Value{ii});
    end
end
end
```

checkForHMDB51Folder

The checkForHMDB51Folder function checks for the downloaded data in the download folder.

```
function classes = checkForHMDB51Folder(dataLoc)
hmdbFolder = fullfile(dataLoc, "hmdb51_org");
if ~exist(hmdbFolder, "dir")
    error("Download 'hmdb51_org.rar' file using the supporting function 'downloadHMDB51' before running");
end

classes = ["brush_hair", "cartwheel", "catch", "chew", "clap", "climb", "climb_stairs", ...
    "dive", "draw_sword", "dribble", "drink", "eat", "fall_floor", "fencing", ...
    "flic_flac", "golf", "handstand", "hit", "hug", "jump", "kick", "kick_ball", ...
    "kiss", "laugh", "pick", "pour", "pullup", "punch", "push", "pushup", "ride_bike", ...
    "ride_horse", "run", "shake_hands", "shoot_ball", "shoot_bow", "shoot_gun", ...
    "sit", "situp", "smile", "smoke", "somersault", "stand", "swing_baseball", "sword", ...
    "sword_exercise", "talk", "throw", "turn", "walk", "wave"];
expectFolders = fullfile(hmdbFolder, classes);
if ~all(arrayfun(@(x) exist(x, 'dir'), expectFolders))
    error("Download hmdb51_org.rar using the supporting function 'downloadHMDB51' before running");
end
end
```

downloadHMDB51

The downloadHMDB51 function downloads the data set and saves it to a directory.

```
function downloadHMDB51(dataLoc)

if nargin == 0
    dataLoc = pwd;
end
```

```
dataLoc = string(dataLoc);

if ~exist(dataLoc,"dir")
    mkdir(dataLoc);
end

dataUrl      = "http://serre-lab.clps.brown.edu/wp-content/uploads/2013/10/hmdb51_org.rar";
options      = weboptions('Timeout', Inf);
rarFileName = fullfile(dataLoc, 'hmdb51_org.rar');
fileExists   = exist(rarFileName, 'file');

% Download the RAR file and save it to the download folder.
if ~fileExists
    disp("Downloading hmdb51_org.rar (2 GB) to the folder:")
    disp(dataLoc)
    disp("This download can take a few minutes...")
    websave(rarFileName, dataUrl, options);
    disp("Download complete.")
    disp("Extract the hmdb51_org.rar file contents to the folder: ")
    disp(dataLoc)
end
end
```

initializeTrainingProgressPlot

The `initializeTrainingProgressPlot` function configures two plots for displaying the training loss, training accuracy, and validation accuracy.

```
function plotters = initializeTrainingProgressPlot(params)
if params.ProgressPlot
    % Plot the loss, training accuracy, and validation accuracy.
    figure

    % Loss plot
    subplot(2,1,1)
    plotters.LossPlotter = animatedline;
    xlabel("Iteration")
    ylabel("Loss")

    % Accuracy plot
    subplot(2,1,2)
    plotters.TrainAccPlotter = animatedline('Color','b');
    plotters.ValAccPlotter = animatedline('Color','g');
    legend('Training Accuracy','Validation Accuracy','Location','northwest');
    xlabel("Iteration")
    ylabel("Accuracy")
else
    plotters = [];
end
end
```

initializeVerboseOutput

The `initializeVerboseOutput` function displays the column headings for the table of training values, which shows the epoch, mini-batch accuracy, and other training values.

```
function initializeVerboseOutput(params)
if params.Verbose
```

```

disp(" ")
if canUseGPU
    disp("Training on GPU.")
else
    disp("Training on CPU.")
end
p = gcp('nocreate');
if ~isempty(p)
    disp("Training on parallel cluster '" + p.Cluster.Profile + "'. ")
end
disp("NumIterations:" + string(params.NumIterations));
disp("MiniBatchSize:" + string(params.MiniBatchSize));
disp("Classes:" + join(string(params.Classes), ","));
disp("=====")
disp("| Epoch | Iteration | Time Elapsed | Mini-Batch Accuracy | Validation Accuracy")
disp("|      |          | (hh:mm:ss)   | (Avg:RGB:Flow)     | (Avg:RGB:Flow)    ")
disp("=====")
end
end

```

displayVerboseOutputEveryEpoch

The `displayVerboseOutputEveryEpoch` function displays the verbose output of the training values, such as the epoch, mini-batch accuracy, validation accuracy, and mini-batch loss.

```

function displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
    accTrain,accTrainRGB,accTrainFlow,accValidation,accValidationRGB,accValidationFlow,lossT
if params.Verbose
    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    trainTime = duration(0,0,trainTime,'Format','hh:mm:ss');
    validationTime = duration(0,0,validationTime,'Format','hh:mm:ss');

    lossValidation = gather(extractdata(lossValidation));
    lossValidation = compose('%.4f',lossValidation);

    accValidation = composePadAccuracy(accValidation);
    accValidationRGB = composePadAccuracy(accValidationRGB);
    accValidationFlow = composePadAccuracy(accValidationFlow);

    accVal = join([accValidation,accValidationRGB,accValidationFlow], " : ");

    lossTrain = gather(extractdata(lossTrain));
    lossTrain = compose('%.4f',lossTrain);

    accTrain = composePadAccuracy(accTrain);
    accTrainRGB = composePadAccuracy(accTrainRGB);
    accTrainFlow = composePadAccuracy(accTrainFlow);

    accTrain = join([accTrain,accTrainRGB,accTrainFlow], " : ");
    learnRate = compose('%.13f',learnRate);

    disp(" | " + ...
        pad(string(epoch),5,'both') + " | " + ...
        pad(string(iteration),9,'both') + " | " + ...
        pad(string(D),12,'both') + " | " + ...
        pad(string(accTrain),26,'both') + " | " + ...
        pad(string(accVal),26,'both') + " | " + ...
        pad(string(lossTrain),10,'both') + " | " + ...

```

```
        pad(string(lossValidation),10,'both') + " | " + ...
        pad(string(learnRate),13,'both') + " | " + ...
        pad(string(trainTime),10,'both') + " | " + ...
        pad(string(validationTime),15,'both') + " |")
    end

end

function acc = composePadAccuracy(acc)
    acc = compose('%.2f',acc*100) + "%";
    acc = pad(string(acc),6,'left');
end
```

endVerboseOutput

The endVerboseOutput function displays the end of verbose output during training.

```
function endVerboseOutput(params)
if params.Verbose
    disp(" |=====|")
end
end
```

updateProgressPlot

The updateProgressPlot function updates the progress plot with loss and accuracy information during training.

```
function updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accuracyTrain,accuracyValidation)
if params.ProgressPlot

    % Update the training progress.
    D = duration(0,0,toc(start),"Format","hh:mm:ss");
    title(plotters.LossPlotter.Parent,"Epoch: " + epoch + ", Elapsed: " + string(D));
    addpoints(plotters.LossPlotter,iteration,double(gather(extractdata(lossTrain))));
    addpoints(plotters.TrainAccPlotter,iteration,accuracyTrain);
    addpoints(plotters.ValAccPlotter,iteration,accuracyValidation);
    drawnow

end
end
```

References

- [1] Carreira, Joao, and Andrew Zisserman. "Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*: 6299-6308. Honolulu, HI: IEEE, 2017.
- [2] Simonyan, Karen, and Andrew Zisserman. "Two-Stream Convolutional Networks for Action Recognition in Videos." *Advances in Neural Information Processing Systems 27*, Long Beach, CA: NIPS, 2017.
- [3] Loshchilov, Ilya, and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts." *International Conference on Learning Representations 2017*. Toulon, France: ICLR, 2017.

Train Fast R-CNN Stop Sign Detector

Load training data.

```
data = load('rcnnStopSigns.mat', 'stopSigns', 'fastRCNNLayers');
stopSigns = data.stopSigns;
fastRCNNLayers = data.fastRCNNLayers;
```

Add fullpath to image files.

```
stopSigns.imageFilename = fullfile(toolboxdir('vision'),'visiondata', ...
    stopSigns.imageFilename);
```

Randomly shuffle data for training.

```
rng(0);
shuffledIdx = randperm(height(stopSigns));
stopSigns = stopSigns(shuffledIdx,:);
```

Create an imageDatastore using the files from the table.

```
imds = imageDatastore(stopSigns.imageFilename);
```

Create a boxLabelDatastore using the label columns from the table.

```
blds = boxLabelDatastore(stopSigns(:,2:end));
```

Combine the datastores.

```
ds = combine(imds, blds);
```

The stop sign training images have different sizes. Preprocess the data to resize the image and boxes to a predefined size.

```
ds = transform(ds,@(data)preprocessData(data,[920 968 3]));
```

Set the network training options.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize', 10, ...
    'InitialLearnRate', 1e-3, ...
    'MaxEpochs', 10, ...
    'CheckpointPath', tempdir);
```

Train the Fast R-CNN detector. Training can take a few minutes to complete.

```
frcnn = trainFastRCNNObjectDetector(ds, fastRCNNLayers, options, ...
    'NegativeOverlapRange', [0 0.1], ...
    'PositiveOverlapRange', [0.7 1]);
```

```
*****
Training a Fast R-CNN Object Detector for the following object classes:
```

```
* stopSign
```

```
--> Extracting region proposals from training datastore...done.
```

```
Training on single GPU.
```

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Loss	Mini-batch Accuracy	Mini-batch RMSE	Base Le Rate
1	1	00:00:29	0.3787	93.59%	0.96	
10	10	00:05:14	0.3032	98.52%	0.95	

Detector training complete.

Test the Fast R-CNN detector on a test image.

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

Run the detector.

```
[bbox, score, label] = detect(frcnn, img);
```

Display detection results.

```
detectedImg = insertObjectAnnotation(img, 'rectangle', bbox, score);  
figure  
imshow(detectedImg)
```



Supporting Functions

```
function data = preprocessData(data, targetSize)  
% Resize image and bounding boxes to the targetSize.  
scale = targetSize(1:2)./size(data{1}, [1 2]);  
data{1} = imresize(data{1}, targetSize(1:2));
```

```
bboxes = round(data{2});  
data{2} = bboxresize(bboxes, scale);  
end
```


Feature Detection and Extraction Examples

- “Automatically Detect and Recognize Text in Natural Images” on page 4-2
- “Digit Classification Using HOG Features” on page 4-14
- “Find Image Rotation and Scale Using Automated Feature Matching” on page 4-22
- “Feature Based Panoramic Image Stitching” on page 4-27
- “Cell Counting” on page 4-33
- “Object Counting” on page 4-36
- “Pattern Matching” on page 4-38
- “Recognize Text Using Optical Character Recognition (OCR)” on page 4-43
- “Cell Counting” on page 4-56

Automatically Detect and Recognize Text in Natural Images

This example shows how to detect regions in an image that contain text. This is a common task performed on unstructured scenes. Unstructured scenes are images that contain undetermined or random scenarios. For example, you can detect and recognize text automatically from captured video to alert a driver about a road sign. This is different than structured scenes, which contain known scenarios where the position of text is known beforehand.

Segmenting text from an unstructured scene greatly helps with additional tasks such as optical character recognition (OCR). The automated text detection algorithm in this example detects a large number of text region candidates and progressively removes those less likely to contain text.

Step 1: Detect Candidate Text Regions Using MSER

The MSER feature detector works well for finding text regions [1]. It works well for text because the consistent color and high contrast of text leads to stable intensity profiles.

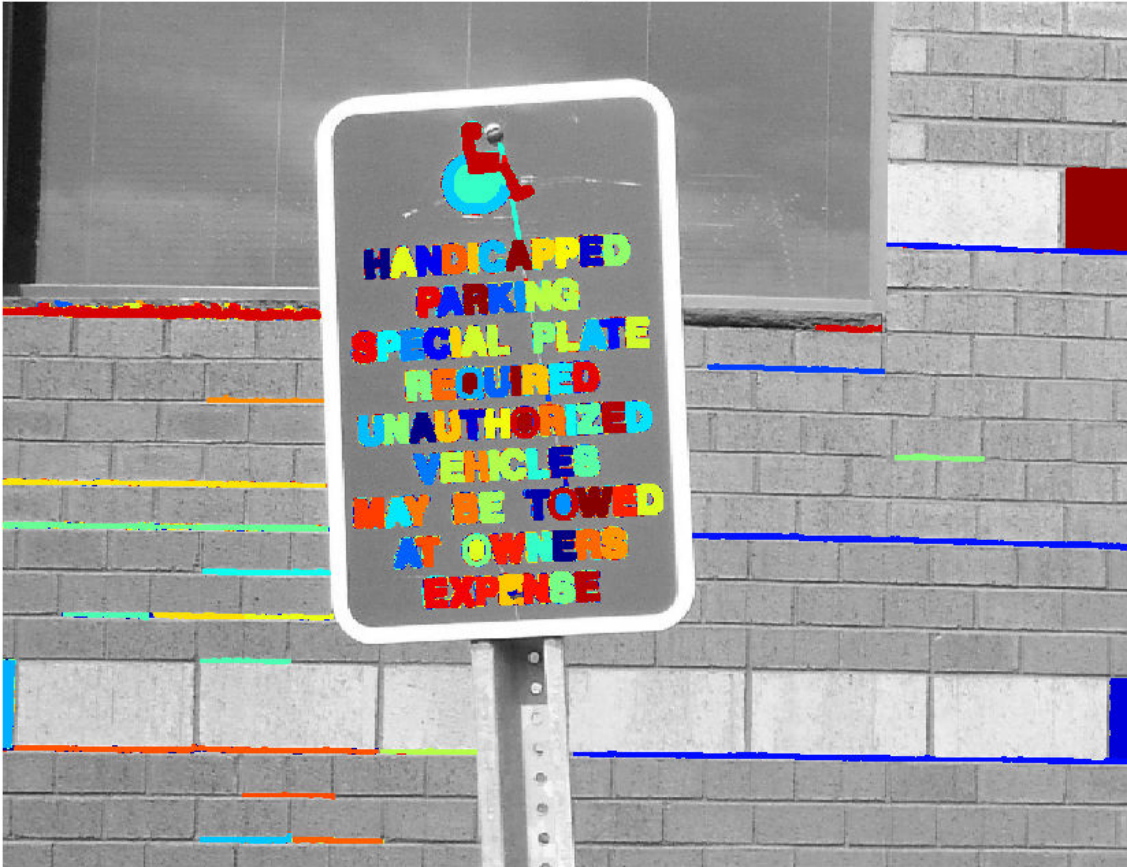
Use the `detectMSERFeatures` function to find all the regions within the image and plot these results. Notice that there are many non-text regions detected alongside the text.

```
colorImage = imread('handicapSign.jpg');
I = rgb2gray(colorImage);

% Detect MSER regions.
[mserRegions, mserConnComp] = detectMSERFeatures(I, ...
    'RegionAreaRange',[200 8000], 'ThresholdDelta',4);

figure
imshow(I)
hold on
plot(mserRegions, 'showPixelList', true, 'showEllipses', false)
title('MSER regions')
hold off
```

MSER regions



Step 2: Remove Non-Text Regions Based On Basic Geometric Properties

Although the MSER algorithm picks out most of the text, it also detects many other stable regions in the image that are not text. You can use a rule-based approach to remove non-text regions. For example, geometric properties of text can be used to filter out non-text regions using simple thresholds. Alternatively, you can use a machine learning approach to train a text vs. non-text classifier. Typically, a combination of the two approaches produces better results [4]. This example uses a simple rule-based approach to filter non-text regions based on geometric properties.

There are several geometric properties that are good for discriminating between text and non-text regions [2,3], including:

- Aspect ratio
- Eccentricity
- Euler number
- Extent
- Solidity

Use `regionprops` to measure a few of these properties and then remove regions based on their property values.

```
% Use regionprops to measure MSER properties
mserStats = regionprops(mserConnComp, 'BoundingBox', 'Eccentricity', ...
    'Solidity', 'Extent', 'Euler', 'Image');

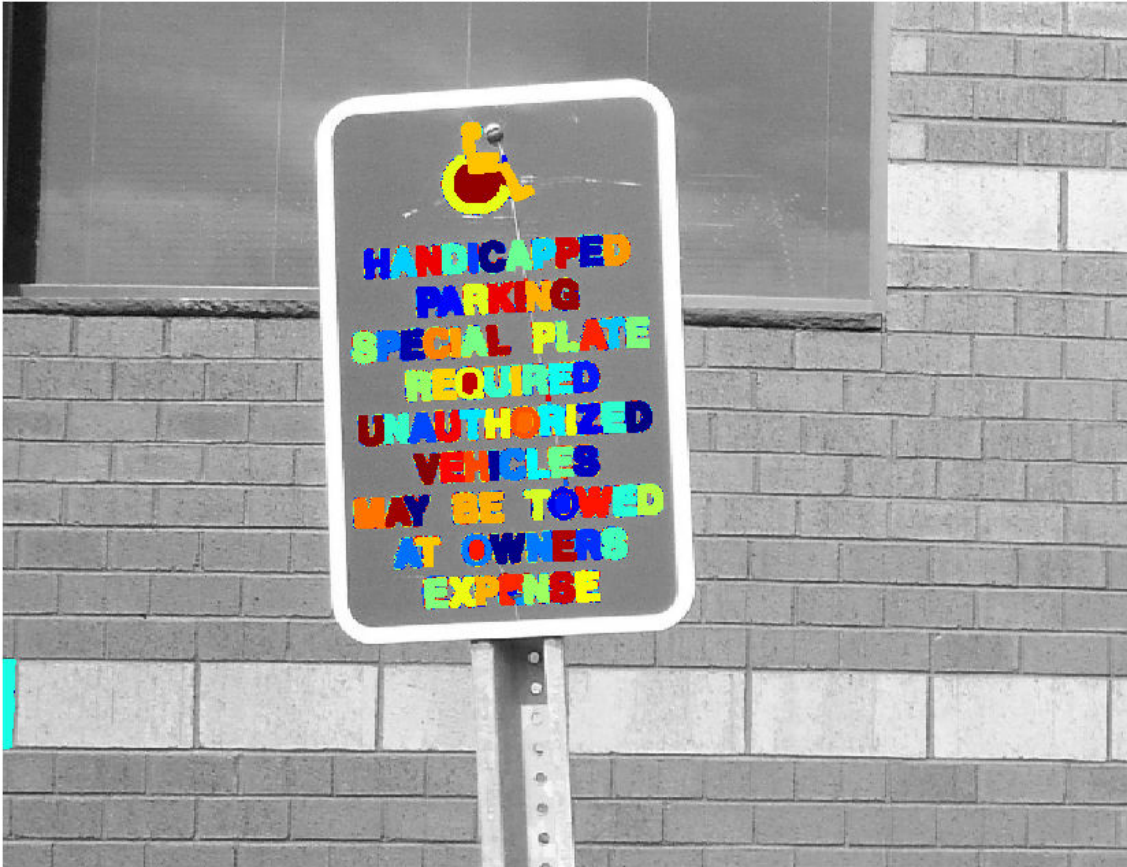
% Compute the aspect ratio using bounding box data.
bbox = vertcat(mserStats.BoundingBox);
w = bbox(:,3);
h = bbox(:,4);
aspectRatio = w./h;

% Threshold the data to determine which regions to remove. These thresholds
% may need to be tuned for other images.
filterIdx = aspectRatio > 3;
filterIdx = filterIdx | [mserStats.Eccentricity] > .995 ;
filterIdx = filterIdx | [mserStats.Solidity] < .3;
filterIdx = filterIdx | [mserStats.Extent] < 0.2 | [mserStats.Extent] > 0.9;
filterIdx = filterIdx | [mserStats.EulerNumber] < -4;

% Remove regions
mserStats(filterIdx) = [];
mserRegions(filterIdx) = [];

% Show remaining regions
figure
imshow(I)
hold on
plot(mserRegions, 'showPixelList', true, 'showEllipses', false)
title('After Removing Non-Text Regions Based On Geometric Properties')
hold off
```

After Removing Non-Text Regions Based On Geometric Properties



Step 3: Remove Non-Text Regions Based On Stroke Width Variation

Another common metric used to discriminate between text and non-text is stroke width. *Stroke width* is a measure of the width of the curves and lines that make up a character. Text regions tend to have little stroke width variation, whereas non-text regions tend to have larger variations.

To help understand how the stroke width can be used to remove non-text regions, estimate the stroke width of one of the detected MSER regions. You can do this by using a distance transform and binary thinning operation [3].

```
% Get a binary image of the a region, and pad it to avoid boundary effects
% during the stroke width computation.
regionImage = mserStats(6).Image;
regionImage = padarray(regionImage, [1 1]);

% Compute the stroke width image.
distanceImage = bwdist(~regionImage);
skeletonImage = bwmorph(regionImage, 'thin', inf);

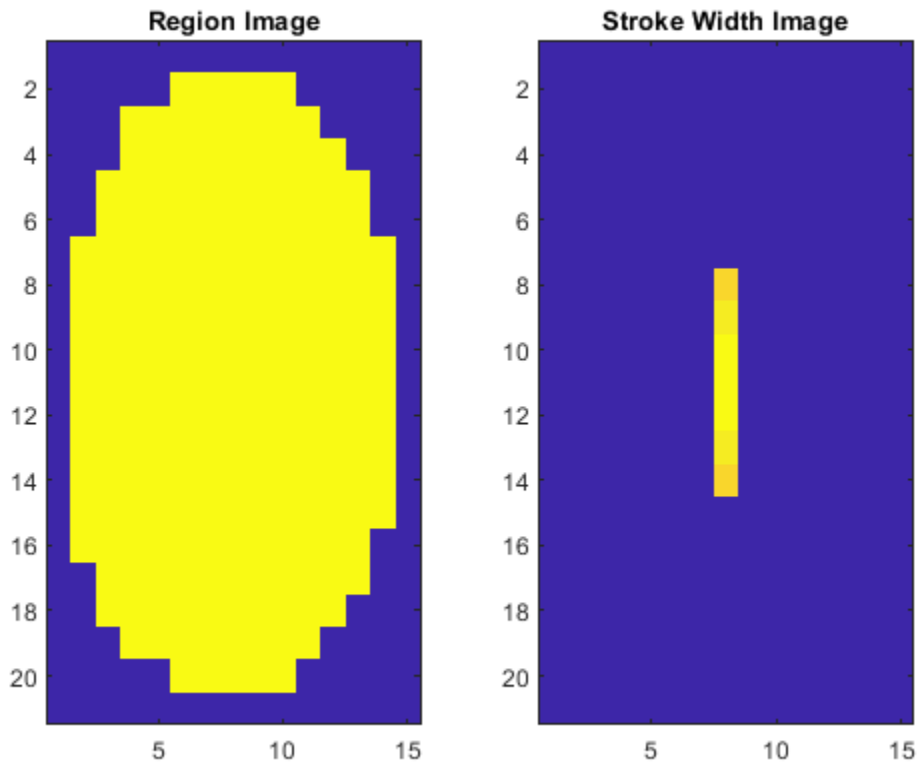
strokeWidthImage = distanceImage;
strokeWidthImage(~skeletonImage) = 0;
```

```

% Show the region image alongside the stroke width image.
figure
subplot(1,2,1)
imagesc(regionImage)
title('Region Image')

subplot(1,2,2)
imagesc(strokeWidthImage)
title('Stroke Width Image')

```



In the images shown above, notice how the stroke width image has very little variation over most of the region. This indicates that the region is more likely to be a text region because the lines and curves that make up the region all have similar widths, which is a common characteristic of human readable text.

In order to use stroke width variation to remove non-text regions using a threshold value, the variation over the entire region must be quantified into a single metric as follows:

```

% Compute the stroke width variation metric
strokeWidthValues = distanceImage(skeletonImage);
strokeWidthMetric = std(strokeWidthValues)/mean(strokeWidthValues);

```

Then, a threshold can be applied to remove the non-text regions. Note that this threshold value may require tuning for images with different font styles.

```

% Threshold the stroke width variation metric
strokeWidthThreshold = 0.4;
strokeWidthFilterIdx = strokeWidthMetric > strokeWidthThreshold;

```

The procedure shown above must be applied separately to each detected MSER region. The following for-loop processes all the regions, and then shows the results of removing the non-text regions using stroke width variation.

```
% Process the remaining regions
for j = 1:numel(mserStats)

    regionImage = mserStats(j).Image;
    regionImage = padarray(regionImage, [1 1], 0);

    distanceImage = bwdist(~regionImage);
    skeletonImage = bwmorph(regionImage, 'thin', inf);

    strokeWidthValues = distanceImage(skeletonImage);

    strokeWidthMetric = std(strokeWidthValues)/mean(strokeWidthValues);

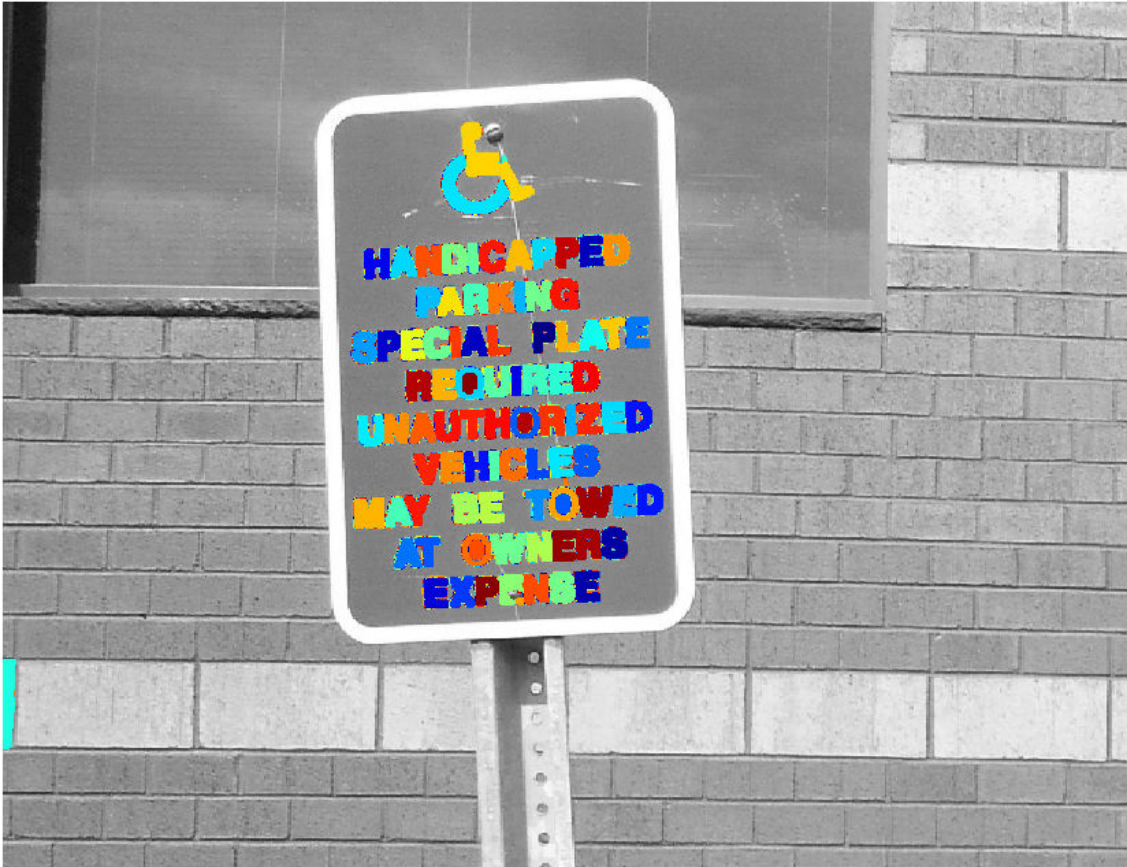
    strokeWidthFilterIdx(j) = strokeWidthMetric > strokeWidthThreshold;

end

% Remove regions based on the stroke width variation
mserRegions(strokeWidthFilterIdx) = [];
mserStats(strokeWidthFilterIdx) = [];

% Show remaining regions
figure
imshow(I)
hold on
plot(mserRegions, 'showPixelList', true, 'showEllipses', false)
title('After Removing Non-Text Regions Based On Stroke Width Variation')
hold off
```

After Removing Non-Text Regions Based On Stroke Width Variation



Step 4: Merge Text Regions For Final Detection Result

At this point, all the detection results are composed of individual text characters. To use these results for recognition tasks, such as OCR, the individual text characters must be merged into words or text lines. This enables recognition of the actual words in an image, which carry more meaningful information than just the individual characters. For example, recognizing the string 'EXIT' vs. the set of individual characters {'X','E','T','T'}, where the meaning of the word is lost without the correct ordering.

One approach for merging individual text regions into words or text lines is to first find neighboring text regions and then form a bounding box around these regions. To find neighboring regions, expand the bounding boxes computed earlier with `regionprops`. This makes the bounding boxes of neighboring text regions overlap such that text regions that are part of the same word or text line form a chain of overlapping bounding boxes.

```
% Get bounding boxes for all the regions
bboxes = vertcat(msrStats.BoundingBox);

% Convert from the [x y width height] bounding box format to the [xmin ymin
% xmax ymax] format for convenience.
xmin = bboxes(:,1);
```



```
ymin = bboxes(:,2);
xmax = xmin + bboxes(:,3) - 1;
ymax = ymin + bboxes(:,4) - 1;

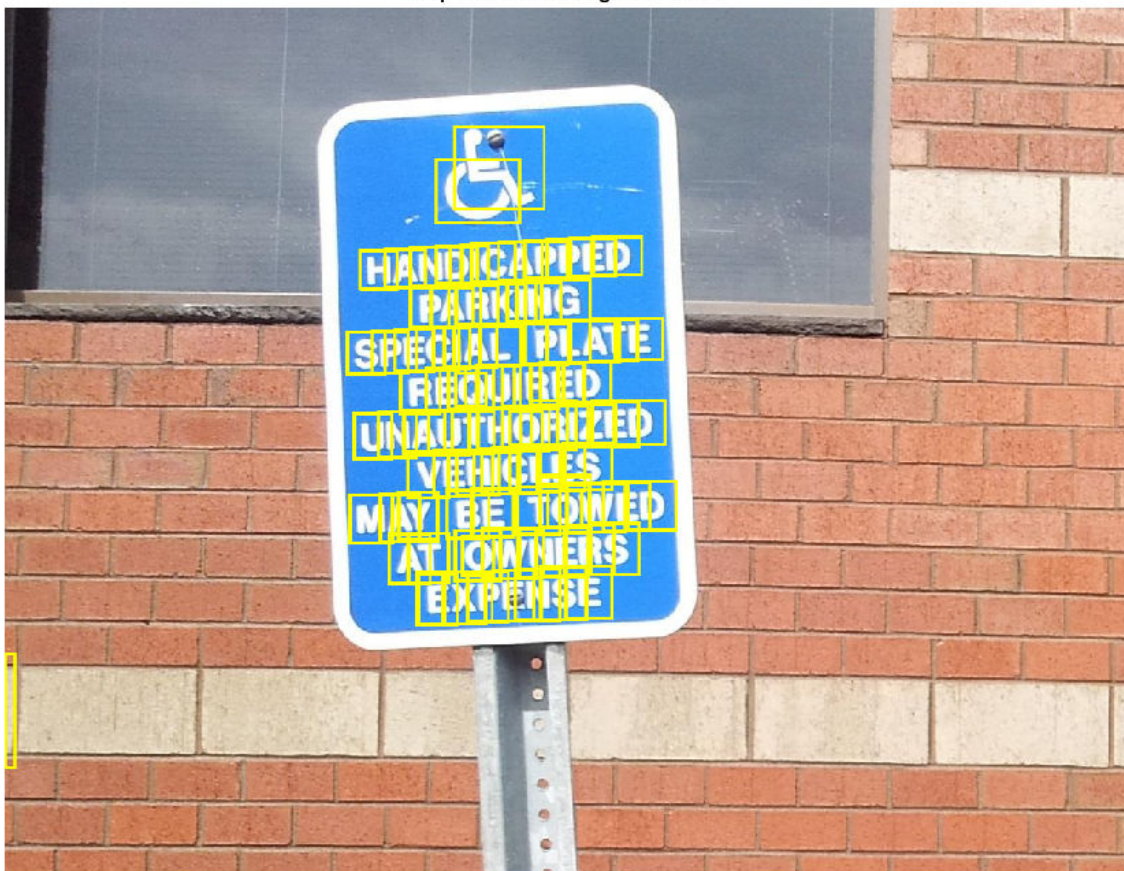
% Expand the bounding boxes by a small amount.
expansionAmount = 0.02;
xmin = (1-expansionAmount) * xmin;
ymin = (1-expansionAmount) * ymin;
xmax = (1+expansionAmount) * xmax;
ymax = (1+expansionAmount) * ymax;

% Clip the bounding boxes to be within the image bounds
xmin = max(xmin, 1);
ymin = max(ymin, 1);
xmax = min(xmax, size(I,2));
ymax = min(ymax, size(I,1));

% Show the expanded bounding boxes
expandedBBboxes = [xmin ymin xmax-xmin+1 ymax-ymin+1];
IExpandedBBboxes = insertShape(colorImage, 'Rectangle', expandedBBboxes, 'LineWidth', 3);

figure
imshow(IExpandedBBboxes)
title('Expanded Bounding Boxes Text')
```

Expanded Bounding Boxes Text



Now, the overlapping bounding boxes can be merged together to form a single bounding box around individual words or text lines. To do this, compute the overlap ratio between all bounding box pairs. This quantifies the distance between all pairs of text regions so that it is possible to find groups of neighboring text regions by looking for non-zero overlap ratios. Once the pair-wise overlap ratios are computed, use a graph to find all the text regions "connected" by a non-zero overlap ratio.

Use the `bbboxOverlapRatio` function to compute the pair-wise overlap ratios for all the expanded bounding boxes, then use `graph` to find all the connected regions.

```
% Compute the overlap ratio
overlapRatio = bbboxOverlapRatio(expandedBBoxes, expandedBBoxes);

% Set the overlap ratio between a bounding box and itself to zero to
% simplify the graph representation.
n = size(overlapRatio,1);
overlapRatio(1:n+1:n^2) = 0;

% Create the graph
g = graph(overlapRatio);

% Find the connected text regions within the graph
componentIndices = conncomp(g);
```

The output of `conncomp` are indices to the connected text regions to which each bounding box belongs. Use these indices to merge multiple neighboring bounding boxes into a single bounding box by computing the minimum and maximum of the individual bounding boxes that make up each connected component.

```
% Merge the boxes based on the minimum and maximum dimensions.
xmin = accumarray(componentIndices', xmin, [], @min);
ymin = accumarray(componentIndices', ymin, [], @min);
xmax = accumarray(componentIndices', xmax, [], @max);
ymax = accumarray(componentIndices', ymax, [], @max);

% Compose the merged bounding boxes using the [x y width height] format.
textBBoxes = [xmin ymin xmax-xmin+1 ymax-ymin+1];
```

Finally, before showing the final detection results, suppress false text detections by removing bounding boxes made up of just one text region. This removes isolated regions that are unlikely to be actual text given that text is usually found in groups (words and sentences).

```
% Remove bounding boxes that only contain one text region
numRegionsInGroup = histcounts(componentIndices);
textBBoxes(numRegionsInGroup == 1, :) = [];

% Show the final text detection result.
ITextRegion = insertShape(colorImage, 'Rectangle', textBBoxes, 'LineWidth',3);

figure
imshow(ITextRegion)
title('Detected Text')
```

Detected Text



Step 5: Recognize Detected Text Using OCR

After detecting the text regions, use the `ocr` function to recognize the text within each bounding box. Note that without first finding the text regions, the output of the `ocr` function would be considerably more noisy.

```
ocrtxt = ocr(I, textBBoxes);  
[ocrtxt.Text]
```

```
ans =  
    'HANDICIXPPED  
    PARKING  
    SPECIAL PLATE  
    REQUIRED  
    UNAUTHORIZED  
    VEHICLES  
    MAY BE TOWED  
    AT OWNERS  
    EXPENSE
```

This example showed you how to detect text in an image using the MSER feature detector to first find candidate text regions, and then it described how to use geometric measurements to remove all the non-text regions. This example code is a good starting point for developing more robust text detection algorithms. Note that without further enhancements this example can produce reasonable results for a variety of other images, for example, posters.jpg or licensePlates.jpg.

References

- [1] Chen, Huizhong, et al. "Robust Text Detection in Natural Images with Edge-Enhanced Maximally Stable Extremal Regions." Image Processing (ICIP), 2011 18th IEEE International Conference on. IEEE, 2011.
- [2] Gonzalez, Alvaro, et al. "Text location in complex images." Pattern Recognition (ICPR), 2012 21st International Conference on. IEEE, 2012.
- [3] Li, Yao, and Huchuan Lu. "Scene text detection via stroke width." Pattern Recognition (ICPR), 2012 21st International Conference on. IEEE, 2012.
- [4] Neumann, Lukas, and Jiri Matas. "Real-time scene text localization and recognition." Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on. IEEE, 2012.

Digit Classification Using HOG Features

This example shows how to classify digits using HOG features and a multiclass SVM classifier.

Object classification is an important task in many computer vision applications, including surveillance, automotive safety, and image retrieval. For example, in an automotive safety application, you may need to classify nearby objects as pedestrians or vehicles. Regardless of the type of object being classified, the basic procedure for creating an object classifier is:

- Acquire a labeled data set with images of the desired object.
- Partition the data set into a training set and a test set.
- Train the classifier using features extracted from the training set.
- Test the classifier using features extracted from the test set.

To illustrate, this example shows how to classify numerical digits using HOG (Histogram of Oriented Gradient) features [1] and a multiclass SVM (Support Vector Machine) classifier. This type of classification is often used in many Optical Character Recognition (OCR) applications.

The example uses the `fitcecoc` function from the Statistics and Machine Learning Toolbox™ and the `extractHOGFeatures` function from the Computer Vision Toolbox™.

Digit Data Set

Synthetic digit images are used for training. The training images each contain a digit surrounded by other digits, which mimics how digits are normally seen together. Using synthetic images is convenient and it enables the creation of a variety of training samples without having to manually collect them. For testing, scans of handwritten digits are used to validate how well the classifier performs on data that is different than the training data. Although this is not the most representative data set, there is enough data to train and test a classifier, and show the feasibility of the approach.

```
% Load training and test data using |imageDatastore|.
syntheticDir = fullfile(toolboxdir('vision'), 'visiondata','digits','synthetic');
handwrittenDir = fullfile(toolboxdir('vision'), 'visiondata','digits','handwritten');

% |imageDatastore| recursively scans the directory tree containing the
% images. Folder names are automatically used as labels for each image.
trainingSet = imageDatastore(syntheticDir, 'IncludeSubfolders', true, 'LabelSource', 'foldernames');
testSet = imageDatastore(handwrittenDir, 'IncludeSubfolders', true, 'LabelSource', 'foldernames');
```

Use `countEachLabel` to tabulate the number of images associated with each label. In this example, the training set consists of 101 images for each of the 10 digits. The test set consists of 12 images per digit.

```
countEachLabel(trainingSet)
```

```
ans=10x2 table
    Label    Count
    ----    -
         0     101
         1     101
         2     101
         3     101
         4     101
         5     101
```

```
6      101
7      101
8      101
9      101
```

```
countEachLabel(testSet)
```

```
ans=10x2 table
```

Label	Count
0	12
1	12
2	12
3	12
4	12
5	12
6	12
7	12
8	12
9	12

Show a few of the training and test images

```
figure;
```

```
subplot(2,3,1);
imshow(trainingSet.Files{102});
```

```
subplot(2,3,2);
imshow(trainingSet.Files{304});
```

```
subplot(2,3,3);
imshow(trainingSet.Files{809});
```

```
subplot(2,3,4);
imshow(testSet.Files{13});
```

```
subplot(2,3,5);
imshow(testSet.Files{37});
```

```
subplot(2,3,6);
imshow(testSet.Files{97});
```



Prior to training and testing a classifier, a pre-processing step is applied to remove noise artifacts introduced while collecting the image samples. This provides better feature vectors for training the classifier.

```
% Show pre-processing results
exTestImage = readimage(testSet,37);
processedImage = imbinarize(rgb2gray(exTestImage));

figure;

subplot(1,2,1)
imshow(exTestImage)

subplot(1,2,2)
imshow(processedImage)
```




Using HOG Features

The data used to train the classifier are HOG feature vectors extracted from the training images. Therefore, it is important to make sure the HOG feature vector encodes the right amount of information about the object. The `extractHOGFeatures` function returns a visualization output that can help form some intuition about just what the "right amount of information" means. By varying the HOG cell size parameter and visualizing the result, you can see the effect the cell size parameter has on the amount of shape information encoded in the feature vector:

```
img = readimage(trainingSet, 206);

% Extract HOG features and HOG visualization
[hog_2x2, vis2x2] = extractHOGFeatures(img, 'CellSize', [2 2]);
[hog_4x4, vis4x4] = extractHOGFeatures(img, 'CellSize', [4 4]);
[hog_8x8, vis8x8] = extractHOGFeatures(img, 'CellSize', [8 8]);

% Show the original image
figure;
subplot(2,3,1:3); imshow(img);

% Visualize the HOG features
subplot(2,3,4);
plot(vis2x2);
title({'CellSize = [2 2]'; ['Length = ' num2str(length(hog_2x2))]});

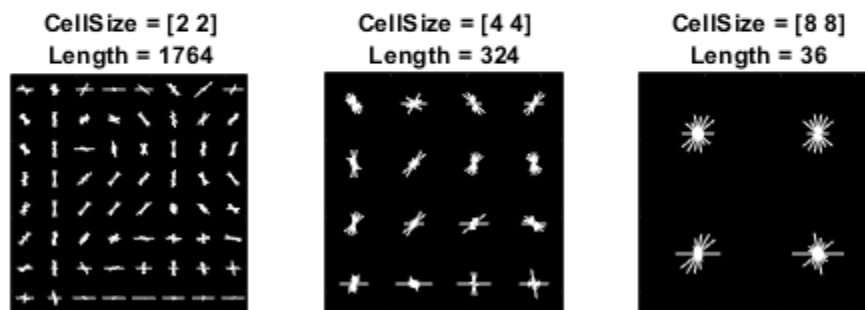
subplot(2,3,5);
plot(vis4x4);
```

```

title({'CellSize = [4 4]'; ['Length = ' num2str(length(hog_4x4))]});

subplot(2,3,6);
plot(vis8x8);
title({'CellSize = [8 8]'; ['Length = ' num2str(length(hog_8x8))]});

```



The visualization shows that a cell size of [8 8] does not encode much shape information, while a cell size of [2 2] encodes a lot of shape information but increases the dimensionality of the HOG feature vector significantly. A good compromise is a 4-by-4 cell size. This size setting encodes enough spatial information to visually identify a digit shape while limiting the number of dimensions in the HOG feature vector, which helps speed up training. In practice, the HOG parameters should be varied with repeated classifier training and testing to identify the optimal parameter settings.

```

cellSize = [4 4];
hogFeatureSize = length(hog_4x4);

```

Train a Digit Classifier

Digit classification is a multiclass classification problem, where you have to classify an image into one out of the ten possible digit classes. In this example, the `fitcecoc` function from the Statistics and Machine Learning Toolbox™ is used to create a multiclass classifier using binary SVMs.

Start by extracting HOG features from the training set. These features will be used to train the classifier.

```

% Loop over the trainingSet and extract HOG features from each image. A
% similar procedure will be used to extract features from the testSet.

```

```

numImages = numel(trainingSet.Files);
trainingFeatures = zeros(numImages, hogFeatureSize, 'single');

for i = 1:numImages
    img = readimage(trainingSet, i);

    img = rgb2gray(img);

    % Apply pre-processing steps
    img = imbinarize(img);

    trainingFeatures(i, :) = extractHOGFeatures(img, 'CellSize', cellSize);
end

% Get labels for each image.
trainingLabels = trainingSet.Labels;

```

Next, train a classifier using the extracted features.

```

% fitcecoc uses SVM learners and a 'One-vs-One' encoding scheme.
classifier = fitcecoc(trainingFeatures, trainingLabels);

```

Evaluate the Digit Classifier

Evaluate the digit classifier using images from the test set, and generate a confusion matrix to quantify the classifier accuracy.

As in the training step, first extract HOG features from the test images. These features will be used to make predictions using the trained classifier.

```

% Extract HOG features from the test set. The procedure is similar to what
% was shown earlier and is encapsulated as a helper function for brevity.
[testFeatures, testLabels] = helperExtractHOGFeaturesFromImageSet(testSet, hogFeatureSize, cellSize);

% Make class predictions using the test features.
predictedLabels = predict(classifier, testFeatures);

% Tabulate the results using a confusion matrix.
confMat = confusionmat(testLabels, predictedLabels);

helperDisplayConfusionMatrix(confMat)

```

digit	0	1	2	3	4	5	6	7	8	9
0	0.25	0.00	0.08	0.00	0.00	0.00	0.58	0.00	0.08	0.00
1	0.00	0.75	0.00	0.00	0.08	0.00	0.00	0.08	0.08	0.00
2	0.00	0.00	0.67	0.17	0.00	0.00	0.08	0.00	0.00	0.08
3	0.00	0.00	0.00	0.58	0.00	0.00	0.33	0.00	0.00	0.08
4	0.00	0.08	0.00	0.17	0.75	0.00	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.00	0.00	0.33	0.58	0.00	0.08	0.00
6	0.00	0.00	0.00	0.00	0.25	0.00	0.67	0.00	0.08	0.00
7	0.00	0.08	0.08	0.33	0.00	0.00	0.17	0.25	0.00	0.08
8	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.08	0.67	0.17
9	0.00	0.08	0.00	0.25	0.17	0.00	0.08	0.00	0.00	0.42

The table shows the confusion matrix in percentage form. The columns of the matrix represent the predicted labels, while the rows represent the known labels. For this test set, digit 0 is often

misclassified as 6, most likely due to their similar shapes. Similar errors are seen for 9 and 3. Training with a more representative data set like MNIST [2] or SVHN [3], which contain thousands of handwritten characters, is likely to produce a better classifier compared with the one created using this synthetic data set.

Summary

This example illustrated the basic procedure for creating a multiclass object classifier using the `extractHOGfeatures` function from the Computer Vision Toolbox and the `fitcecoc` function from the Statistics and Machine Learning Toolbox™. Although HOG features and an ECOC classifier were used here, other features and machine learning algorithms can be used in the same way. For instance, you can explore using different feature types for training the classifier; or you can see the effect of using other machine learning algorithms available in the Statistics and Machine Learning Toolbox™ such as k-nearest neighbors.

Supporting Functions

```
function helperDisplayConfusionMatrix(confMat)
% Display the confusion matrix in a formatted table.

% Convert confusion matrix into percentage form
confMat = bsxfun(@rdivide,confMat,sum(confMat,2));

digits = '0':'9';
colHeadings = arrayfun(@(x)sprintf('%d',x),0:9,'UniformOutput',false);
format = repmat('%-9s',1,11);
header = sprintf(format,'digit |',colHeadings{:});
fprintf('\n%s\n%s\n',header, repmat('-',size(header)));
for idx = 1:numel(digits)
    fprintf('%-9s', [digits(idx) ' |']);
    fprintf('%-9.2f', confMat(idx,:));
    fprintf('\n')
end
end

function [features, setLabels] = helperExtractHOGFeaturesFromImageSet(imds, hogFeatureSize, cellSize)
% Extract HOG features from an imageDatastore.

setLabels = imds.Labels;
numImages = numel(imds.Files);
features = zeros(numImages, hogFeatureSize, 'single');

% Process each image and extract features
for j = 1:numImages
    img = readimage(imds, j);
    img = rgb2gray(img);

    % Apply pre-processing steps
    img = imbinarize(img);

    features(j, :) = extractHOGFeatures(img,'CellSize',cellSize);
end
end
```

References

[1] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection", Proc. IEEE Conf. Computer Vision and Pattern Recognition, vol. 1, pp. 886-893, 2005.

[2] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86, 2278-2324.

[3] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A.Y. Ng, Reading Digits in Natural Images with Unsupervised Feature Learning NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011.

Find Image Rotation and Scale Using Automated Feature Matching

This example shows how to automatically determine the geometric transformation between a pair of images. When one image is distorted relative to another by rotation and scale, use `detectSURFFeatures` and `estimateGeometricTransform2D` to find the rotation angle and scale factor. You can then transform the distorted image to recover the original image.

Step 1: Read Image

Bring an image into the workspace.

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

Step 2: Resize and Rotate the Image

```
scale = 0.7;  
J = imresize(original, scale); % Try varying the scale factor.  
  
theta = 30;  
distorted = imrotate(J,theta); % Try varying the angle, theta.  
figure, imshow(distorted)
```



You can experiment by varying the scale and rotation of the input image. However, note that there is a limit to the amount you can vary the scale before the feature detector fails to find enough features.

Step 3: Find Matching Features Between Images

Detect features in both images.

```
ptsOriginal = detectSURFFeatures(original);
ptsDistorted = detectSURFFeatures(distorted);
```

Extract feature descriptors.

```
[featuresOriginal, validPtsOriginal] = extractFeatures(original, ptsOriginal);
[featuresDistorted, validPtsDistorted] = extractFeatures(distorted, ptsDistorted);
```

Match features by using their descriptors.

```
indexPairs = matchFeatures(featuresOriginal, featuresDistorted);
```

Retrieve locations of corresponding points for each image.

```
matchedOriginal = validPtsOriginal(indexPairs(:,1));
matchedDistorted = validPtsDistorted(indexPairs(:,2));
```

Show putative point matches.

```
figure;
showMatchedFeatures(original,distorted,matchedOriginal,matchedDistorted);
title('Putatively matched points (including outliers)');
```

Putatively matched points (including outliers)



Step 4: Estimate Transformation

Find a transformation corresponding to the matching point pairs using the statistically robust M-estimator SAmple Consensus (MSAC) algorithm, which is a variant of the RANSAC algorithm. It removes outliers while computing the transformation matrix. You may see varying results of the transformation computation because of the random sampling employed by the MSAC algorithm.

```
[tform, inlierIdx] = estimateGeometricTransform2D(...
    matchedDistorted, matchedOriginal, 'similarity');
inlierDistorted = matchedDistorted(inlierIdx, :);
inlierOriginal  = matchedOriginal(inlierIdx, :);
```

Display matching point pairs used in the computation of the transformation.

```
figure;
showMatchedFeatures(original,distorted,inlierOriginal,inlierDistorted);
title('Matching points (inliers only)');
legend('ptsOriginal','ptsDistorted');
```




Step 5: Solve for Scale and Angle

Use the geometric transform, `tform`, to recover the scale and angle. Since we computed the transformation from the distorted to the original image, we need to compute its inverse to recover the distortion.

```
Let sc = s*cos(theta)
Let ss = s*sin(theta)
```

```
Then, Tinv = [sc -ss 0;
              ss  sc 0;
              tx  ty 1]
```

where `tx` and `ty` are `x` and `y` translations, respectively.

Compute the inverse transformation matrix.

```
Tinv = tform.invert.T;
```

```
ss = Tinv(2,1);
sc = Tinv(1,1);
scaleRecovered = sqrt(ss*ss + sc*sc)
thetaRecovered = atan2(ss,sc)*180/pi
```

```
scaleRecovered =
```

```
    single
```

```
    0.7010
```

```
thetaRecovered =
```

```
single  
30.2351
```

The recovered values should match your scale and angle values selected in **Step 2: Resize and Rotate the Image**.

Step 6: Recover the Original Image

Recover the original image by transforming the distorted image.

```
outputView = imref2d(size(original));  
recovered = imwarp(distorted,tform,'OutputView',outputView);
```

Compare recovered to original by looking at them side-by-side in a montage.

```
figure, imshowpair(original,recovered,'montage')
```



The recovered (right) image quality does not match the original (left) image because of the distortion and recovery process. In particular, the image shrinking causes loss of information. The artifacts around the edges are due to the limited accuracy of the transformation. If you were to detect more points in **Step 3: Find Matching Features Between Images**, the transformation would be more accurate. For example, we could have used a corner detector, `detectFASTFeatures`, to complement the SURF feature detector which finds blobs. Image content and image size also impact the number of detected features.

Feature Based Panoramic Image Stitching

This example shows how to automatically create a panorama using feature based image registration techniques.

Overview

Feature detection and matching are powerful techniques used in many computer vision applications such as image registration, tracking, and object detection. In this example, feature based techniques are used to automatically stitch together a set of images. The procedure for image stitching is an extension of feature based image registration. Instead of registering a single pair of images, multiple image pairs are successively registered relative to each other to form a panorama.

Step 1 - Load Images

The image set used in this example contains pictures of a building. These were taken with an uncalibrated smart phone camera by sweeping the camera from left to right along the horizon, capturing all parts of the building.

As seen below, the images are relatively unaffected by any lens distortion so camera calibration was not required. However, if lens distortion is present, the camera should be calibrated and the images undistorted prior to creating the panorama. You can use the Camera Calibrator App to calibrate a camera if needed.

```
% Load images.
buildingDir = fullfile(toolboxdir('vision'), 'visiondata', 'building');
buildingScene = imageDatastore(buildingDir);

% Display images to be stitched
montage(buildingScene.Files)
```



Step 2 - Register Image Pairs

To create the panorama, start by registering successive image pairs using the following procedure:

- 1 Detect and match features between $I(n)$ and $I(n - 1)$.
- 2 Estimate the geometric transformation, $T(n)$, that maps $I(n)$ to $I(n - 1)$.
- 3 Compute the transformation that maps $I(n)$ into the panorama image as $T(n) * T(n - 1) * \dots * T(1)$.

```

% Read the first image from the image set.
I = readimage(buildingScene, 1);

% Initialize features for I(1)
grayImage = rgb2gray(I);
points = detectSURFFeatures(grayImage);
[features, points] = extractFeatures(grayImage, points);

% Initialize all the transforms to the identity matrix. Note that the
% projective transform is used here because the building images are fairly
% close to the camera. Had the scene been captured from a further distance,
% an affine transform would suffice.
numImages = numel(buildingScene.Files);
tforms(numImages) = projective2d(eye(3));

% Initialize variable to hold image sizes.
imageSize = zeros(numImages,2);

% Iterate over remaining image pairs
for n = 2:numImages

    % Store points and features for I(n-1).
    pointsPrevious = points;
    featuresPrevious = features;

    % Read I(n).
    I = readimage(buildingScene, n);

    % Convert image to grayscale.
    grayImage = rgb2gray(I);

    % Save image size.
    imageSize(n,:) = size(grayImage);

    % Detect and extract SURF features for I(n).
    points = detectSURFFeatures(grayImage);
    [features, points] = extractFeatures(grayImage, points);

    % Find correspondences between I(n) and I(n-1).
    indexPairs = matchFeatures(features, featuresPrevious, 'Unique', true);

    matchedPoints = points(indexPairs(:,1), :);
    matchedPointsPrev = pointsPrevious(indexPairs(:,2), :);

    % Estimate the transformation between I(n) and I(n-1).
    tforms(n) = estimateGeometricTransform2D(matchedPoints, matchedPointsPrev,...
        'projective', 'Confidence', 99.9, 'MaxNumTrials', 2000);

    % Compute T(n) * T(n-1) * ... * T(1)
    tforms(n).T = tforms(n).T * tforms(n-1).T;
end

```

At this point, all the transformations in `tforms` are relative to the first image. This was a convenient way to code the image registration procedure because it allowed sequential processing of all the images. However, using the first image as the start of the panorama does not produce the most aesthetically pleasing panorama because it tends to distort most of the images that form the panorama. A nicer panorama can be created by modifying the transformations such that the center of

the scene is the least distorted. This is accomplished by inverting the transform for the center image and applying that transform to all the others.

Start by using the `projective2d` `outputLimits` method to find the output limits for each transform. The output limits are then used to automatically find the image that is roughly in the center of the scene.

```
% Compute the output limits for each transform
for i = 1:numel(tforms)
    [xlim(i,:), ylim(i,:)] = outputLimits(tforms(i), [1 imageSize(i,2)], [1 imageSize(i,1)]);
end
```

Next, compute the average X limits for each transforms and find the image that is in the center. Only the X limits are used here because the scene is known to be horizontal. If another set of images are used, both the X and Y limits may need to be used to find the center image.

```
avgXLim = mean(xlim, 2);
[~, idx] = sort(avgXLim);
centerIdx = floor((numel(tforms)+1)/2);
centerImageIdx = idx(centerIdx);
```

Finally, apply the center image's inverse transform to all the others.

```
Tinv = invert(tforms(centerImageIdx));
for i = 1:numel(tforms)
    tforms(i).T = tforms(i).T * Tinv.T;
end
```

Step 3 - Initialize the Panorama

Now, create an initial, empty, panorama into which all the images are mapped.

Use the `outputLimits` method to compute the minimum and maximum output limits over all transformations. These values are used to automatically compute the size of the panorama.

```
for i = 1:numel(tforms)
    [xlim(i,:), ylim(i,:)] = outputLimits(tforms(i), [1 imageSize(i,2)], [1 imageSize(i,1)]);
end

maxImageSize = max(imageSize);

% Find the minimum and maximum output limits
xMin = min([1; xlim(:)]);
xMax = max([maxImageSize(2); xlim(:)]);

yMin = min([1; ylim(:)]);
yMax = max([maxImageSize(1); ylim(:)]);

% Width and height of panorama.
width = round(xMax - xMin);
height = round(yMax - yMin);

% Initialize the "empty" panorama.
panorama = zeros([height width 3], 'like', I);
```

Step 4 - Create the Panorama

Use `imwarp` to map images into the panorama and use `vision.AlphaBlender` to overlay the images together.

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

% Create a 2-D spatial reference object defining the size of the panorama.
xLimits = [xMin xMax];
yLimits = [yMin yMax];
panoramaView = imref2d([height width], xLimits, yLimits);

% Create the panorama.
for i = 1:numImages

    I = readimage(buildingScene, i);

    % Transform I into the panorama.
    warpedImage = imwarp(I, tforms(i), 'OutputView', panoramaView);

    % Generate a binary mask.
    mask = imwarp(true(size(I,1),size(I,2)), tforms(i), 'OutputView', panoramaView);

    % Overlay the warpedImage onto the panorama.
    panorama = step(blender, panorama, warpedImage, mask);
end

figure
imshow(panorama)
```



Conclusion

This example showed you how to automatically create a panorama using feature based image registration techniques. Additional techniques can be incorporated into the example to improve the blending and alignment of the panorama images[1].

References

- [1] Matthew Brown and David G. Lowe. 2007. Automatic Panoramic Image Stitching using Invariant Features. *Int. J. Comput. Vision* 74, 1 (August 2007), 59-73.

Cell Counting

This example shows how to use a combination of basic morphological operators and blob analysis to extract information from a video stream. In this case, the example counts the number of E. Coli bacteria in each video frame. Note that the cells are of varying brightness, which makes the task of segmentation more challenging.

Initialization

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

```
VideoSize = [432 528];
```

Create a System object to read video from avi file.

```
filename = 'ecolicells.avi';
hvfr = VideoReader(filename);
```

Create a BlobAnalysis System object to find the centroid of the segmented cells in the video.

```
hblob = vision.BlobAnalysis( ...
    'AreaOutputPort', false, ...
    'BoundingBoxOutputPort', false, ...
    'OutputDataType', 'single', ...
    'MinimumBlobArea', 7, ...
    'MaximumBlobArea', 300, ...
    'MaximumCount', 1500);
```

```
% Acknowledgement
```

```
ackText = ['Data set courtesy of Jonathan Young and Michael Elowitz, ' ...
    'California Institute of Technology'];
```

Create a System object to display the video.

```
hVideo = vision.VideoPlayer;
hVideo.Name = 'Results';
hVideo.Position(1) = round(hVideo.Position(1));
hVideo.Position(2) = round(hVideo.Position(2));
hVideo.Position([4 3]) = 30+VideoSize;
```

Stream Processing Loop

Create a processing loop to count the number of cells in the input video. This loop uses the System objects you instantiated above.

```
frameCount = int16(1);
while hasFrame(hvfr)
    % Read input video frame
    image = rgb2gray(im2single(readFrame(hvfr)));

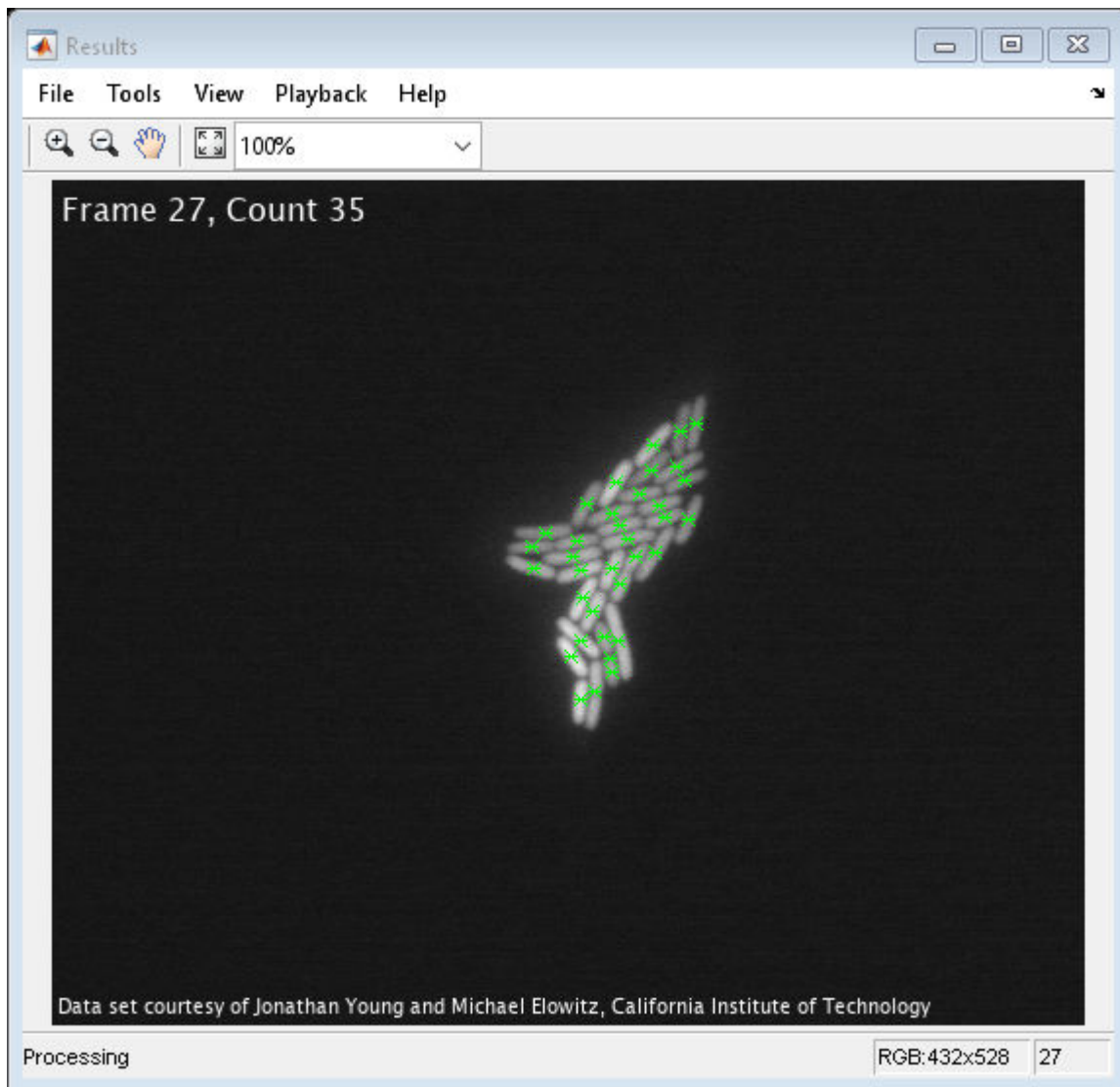
    % Apply a combination of morphological dilation and image arithmetic
    % operations to remove uneven illumination and to emphasize the
    % boundaries between the cells.
    y1 = 2*image - imdilate(image, strel('square',7));
    y1(y1<0) = 0;
    y1(y1>1) = 1;
    y2 = imdilate(y1, strel('square',7)) - y1;
```

```
th = multithresh(y2);      % Determine threshold using Otsu's method
y3 = (y2 <= th*0.7);     % Binarize the image.

Centroid = step(hblob, y3); % Calculate the centroid
numBlobs = size(Centroid,1); % and number of cells.
% Display the number of frames and cells.
frameBlobTxt = sprintf('Frame %d, Count %d', frameCount, numBlobs);
image = insertText(image, [1 1], frameBlobTxt, ...
    'FontSize', 16, 'BoxOpacity', 0, 'TextColor', 'white');
image = insertText(image, [1 size(image,1)], ackText, ...
    'FontSize', 10, 'AnchorPoint', 'LeftBottom', ...
    'BoxOpacity', 0, 'TextColor', 'white');

% Display video
image_out = insertMarker(image, Centroid, '*', 'Color', 'green');
step(hVideo, image_out);

frameCount = frameCount + 1;
end
```



Summary

In the Results window the original video is shown and the green markers indicate the centroid locations of the cells. The frame number and the number of cells are displayed in the upper left corner.

Data Set Credits

The data set for this example was provided by Jonathan Young and Michael Elowitz from California Institute of Technology. It is used with permission. For additional information about this data, see

N. Rosenfeld, J. Young, U. Alon, P. Swain, and M.B. Elowitz, "Gene Regulation at the Single-Cell Level," *Science* 2005, Vol. 307, pp. 1962-1965.

Object Counting

This example shows how to use morphological operations to count objects in a video stream.

Introduction

The input video stream contains images of staples. In this example, you use the top-hat morphological operation to remove uneven illumination, and the opening morphological operation to remove gaps between the staples. You then convert the images to binary, using a different threshold for each frame. Once this threshold is applied, you count the number of staples and calculate the centroid of each staple.

Initialization

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

Create a System object to read video from avi file.

```
filename = 'staples.mp4';
hVideoSrc = VideoReader(filename);
```

Create a blob analysis System object to count the staples and find their centroids.

```
hBlob = vision.BlobAnalysis( ...
    'AreaOutputPort', false, ...
    'BoundingBoxOutputPort', false, ...
    'OutputDataType', 'single');
```

Create a System object to display the output video.

```
hVideoOut = vision.VideoPlayer('Name', 'Counted Staples');
hVideoOut.Position(3:4) = [650 350];
```

Stream Processing Loop

Here you call the processing loop to count the staples in the input video. This loop uses the System objects you instantiated.

The loop is stopped when you reach the end of the input file, which is detected by the BinaryFileReader System object.

```
while hasFrame(hVideoSrc)
    I = rgb2gray(readFrame(hVideoSrc));
    Im = imtophat(I, strel('square',18));
    Im = imopen(Im, strel('rect',[15 3]));
    th = multithresh(Im); % Determine threshold using Otsu's method
    BW = Im > th;
    Centroids = step(hBlob, BW); % Blob Analysis

    StaplesCount = int32(size(Centroids,1));
    txt = sprintf('Staple count: %d', StaplesCount);
    It = insertText(I,[10 280],txt,'FontSize',22); % Display staples count

    Centroids(:, 2) = Centroids(1,2); % Align markers horizontally

    It = insertMarker(It, Centroids, 'o', 'Size', 6, 'Color', 'r');
    It = insertMarker(It, Centroids, 'o', 'Size', 5, 'Color', 'r');
```

```
It = insertMarker(It, Centroids, '+', 'Size', 5, 'Color', 'r');  
step(hVideoOut, It);  
end
```



Summary

The output video shows the individual staples marked with a circle and plus sign. It also displays the number of staples that appear in each frame.

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 = VideoReader('vipboard.mp4');
```

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);
```

```
% 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.

```
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.

```
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 VideoReader object.

```
while hasFrame(hVideoSrc)
    Im = rgb2gray(im2single(readFrame(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.
videopatternplots('update', hPlot, norm_Corr_value);
step(hROIPattern, Imf);
end

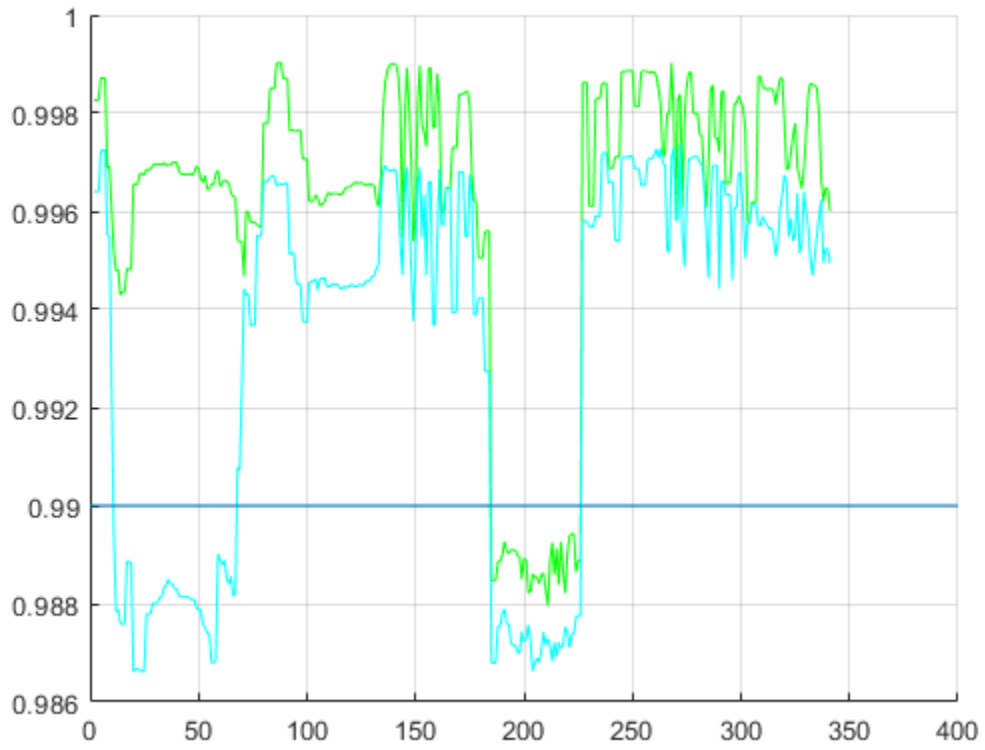
snapnow

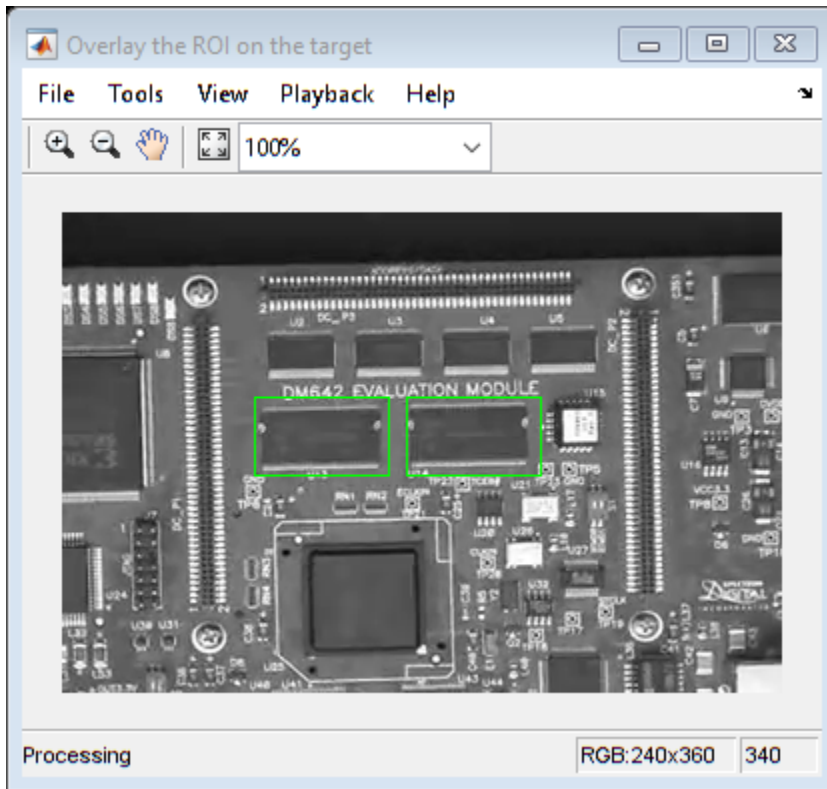
% Function to compute pyramid image at a particular level.
function outI = multilevelPyramid(inI, level)

I = inI;
outI = I;

for i=1:level
    outI = impyramid(I, 'reduce');
    I = outI;
end

end
```



Summary

This example shows use of Computer Vision 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

Recognize Text Using Optical Character Recognition (OCR)

This example shows how to use the `ocr` function from the Computer Vision Toolbox™ to perform Optical Character Recognition.

Text Recognition Using the `ocr` Function

Recognizing text in images is useful in many computer vision applications such as image search, document analysis, and robot navigation. The `ocr` function provides an easy way to add text recognition functionality to a wide range of applications.

```
% Load an image.
I = imread('businessCard.png');

% Perform OCR.
results = ocr(I);

% Display one of the recognized words.
word = results.Words{2}

word =
'MathWorks®'

% Location of the word in I
wordBBox = results.WordBoundingBoxes(2,:)

wordBBox = 1×4

    173    75   376    61

% Show the location of the word in the original image.
figure;
Iname = insertObjectAnnotation(I,'rectangle',wordBBox,word);
imshow(Iname);
```



Information Returned By the ocr Function

The `ocr` functions returns the recognized text, the recognition confidence, and the location of the text in the original image. You can use this information to identify the location of misclassified text within the image.

```
% Find characters with low confidence.
lowConfidenceIdx = results.CharacterConfidences < 0.5;

% Get the bounding box locations of the low confidence characters.
lowConfBBoxes = results.CharacterBoundingBoxes(lowConfidenceIdx, :);

% Get confidence values.
lowConfVal = results.CharacterConfidences(lowConfidenceIdx);

% Annotate image with character confidences.
str = sprintf('confidence = %f', lowConfVal);
Ilowconf = insertObjectAnnotation(I, 'rectangle', lowConfBBoxes, str);

figure;
imshow(Ilowconf);
```



Here, the logo in the business card is incorrectly classified as a text character. These kind of OCR errors can be identified using the confidence values before any further processing takes place.

Challenges Obtaining Accurate Results

ocr performs best when the text is located on a uniform background and is formatted like a document. When the text appears on a non-uniform background, additional pre-processing steps are required to get the best OCR results. In this part of the example, you will try to locate the digits on a keypad. Although, the keypad image may appear to be easy for OCR, it is actually quite challenging because the text is on a non-uniform background.

```
I = imread('keypad.jpg');  
I = rgb2gray(I);
```

```
figure;  
imshow(I)
```



```
% Run OCR on the image  
results = ocr(I);
```

```
results.Text
```

```
ans =  
,
```

The empty `results.Text` indicates that no text is recognized. In the keypad image, the text is sparse and located on an irregular background. In this case, the heuristics used for document layout analysis within `ocr` might be failing to find blocks of text within the image, and, as a result, text recognition fails. In this situation, disabling the automatic layout analysis, using the 'TextLayout' parameter, may help improve the results.

```
% Set 'TextLayout' to 'Block' to instruct ocr to assume the image  
% contains just one block of text.  
results = ocr(I, 'TextLayout', 'Block');
```

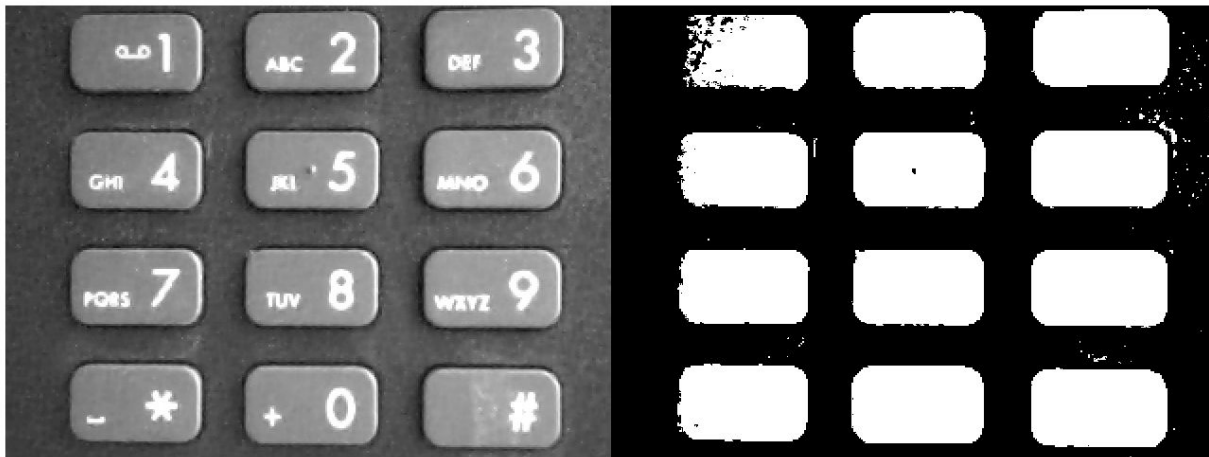
```
results.Text
```

```
ans =
    0x0 empty char array
```

What Went Wrong?

Adjusting the 'TextLayout' parameter did not help. To understand why OCR continues to fail, you have to investigate the initial binarization step performed within `ocr`. You can use `imbinarize` to check this initial binarization step because both `ocr` and the default 'global' method in `imbinarize` use Otsu's method for image binarization.

```
BW = imbinarize(I);
figure;
imshowpair(I,BW,'montage');
```



After thresholding, the binary image contains no text. This is why `ocr` failed to recognize any text in the original image. You can help improve the results by pre-processing the image to improve text segmentation. The next part of the example explores two useful pre-processing techniques.

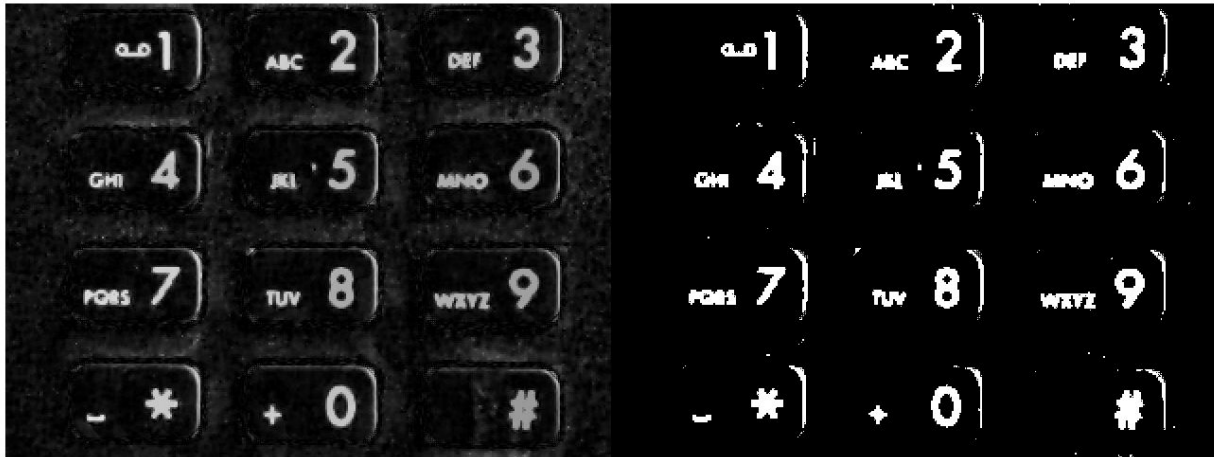
Image Pre-processing Techniques To Improve Results

The poor text segmentation seen above is caused by the non-uniform background in the image, i.e. the light-gray keys surrounded by dark gray. You can use the following pre-processing technique to remove the background variations and improve the text segmentation. Additional details about this technique are given in the example entitled "Correct Nonuniform Illumination and Analyze Foreground Objects".

```
% Remove keypad background.
Icorrected = imtophat(I,strel('disk',15));

BW1 = imbinarize(Icorrected);

figure;
imshowpair(Icorrected,BW1,'montage');
```



After removing the background variation, the digits are now visible in the binary image. However, there are a few artifacts at the edge of the keys and the small text next to the digits that may continue to hinder accurate OCR of the whole image. Additional pre-processing using morphological reconstruction helps to remove these artifacts and produce a cleaner image for OCR.

```
% Perform morphological reconstruction and show binarized image.
```

```
marker = imerode(Icorrected, strel('line',10,0));
```

```
Iclean = imreconstruct(marker, Icorrected);
```

```
BW2 = imbinarize(Iclean);
```

```
figure;
```

```
imshowpair(Iclean,BW2,'montage');
```



After these pre-processing steps, the digits are now well segmented from the background and ocr produces better results.


```
results = ocr(BW2, 'TextLayout', 'Block');
```

```
results.Text
```

```
ans =
    '<-1 ..c2 .3
     ....4 .5 .....6
     W7 M8 M9
     -*1..o fl
     ,
```

There is some "noise" in the results due to the smaller text next to the digits. Also, the digit 0, is falsely recognized as the letter 'o'. This type of error may happen when two characters have similar shapes and there is not enough surrounding text for the `ocr` function to determine the best classification for a specific character. Despite the "noisy" results, you can still find the digit locations in the original image using the `locateText` method with the OCR results.

The `locateText` method supports regular expressions so you can ignore irrelevant text.

```
% The regular expression, '\d', matches the location of any digit in the
% recognized text and ignores all non-digit characters.
regularExpr = '\d';
```

```
% Get bounding boxes around text that matches the regular expression
bboxes = locateText(results,regularExpr,'UseRegexp',true);
```

```
digits = regexp(results.Text,regularExpr,'match');
```

```
% draw boxes around the digits
Idigits = insertObjectAnnotation(I,'rectangle',bboxes,digits);
```

```
figure;
imshow(Idigits);
```



Another approach to improve the results is to leverage a priori knowledge about the text within the image. In this example, the text you are interested in contains only numeric digits. You can improve the results by constraining ocr to only select the best matches from the set '0123456789'.

```
% Use the 'CharacterSet' parameter to constrain OCR
results = ocr(BW2, 'CharacterSet', '0123456789', 'TextLayout', 'Block');
```

```
results.Text
```

```
ans =
    ' 1 1 2 3
      5 4 5 06
        7 3 9
          4 1 0 51
        '
```

The results now only have characters from the digit character set. However, you can see that several non-digit characters in the image are falsely recognized as digits. This can happen when a non-digit character closely resembles one of the digits.

You can use the fact that there are only 10 digits on the keypad along with the character confidences to find the 10 best digits.

```
% Sort the character confidences.
[sortedConf, sortedIndex] = sort(results.CharacterConfidences, 'descend');

% Keep indices associated with non-NaN confidences values.
indexesNaNsRemoved = sortedIndex( ~isnan(sortedConf) );

% Get the top ten indexes.
topTenIndexes = indexesNaNsRemoved(1:10);

% Select the top ten results.
digits = num2cell(results.Text(topTenIndexes));
bboxes = results.CharacterBoundingBoxes(topTenIndexes, :);

Idigits = insertObjectAnnotation(I, 'rectangle', bboxes, digits);

figure;
imshow(Idigits);
```



ROI-based Processing To Improve Results

In some situations, just pre-processing the image may not be sufficient to achieve good OCR results. One approach to use in this situation, is to identify specific regions in the image that ocr should process. In the keypad example image, these regions would be those that just contain the digits. You may select the regions manually using `imrect`, or you can automate the process. One method for automating text detection is given in the example entitled “Automatically Detect and Recognize Text in Natural Images” on page 4-2. In this example, you will use `vision.BlobAnalysis` to find the digits on the keypad.

```
% Initialize the blob analysis System object(TM).
blobAnalyzer = vision.BlobAnalysis('MaximumCount',500);

% Run the blob analyzer to find connected components and their statistics.
[area,centroids,roi] = step(blobAnalyzer,BW1);

% Show all the connected regions.
img = insertShape(I,'rectangle',roi);
figure;
imshow(img);
```



There are many connected regions within the keypad image. Small regions are not likely to contain any text and can be removed using the area statistic returned by `vision.BlobAnalysis`. Here, regions having an area smaller than 300 are removed.

```
areaConstraint = area > 300;

% Keep regions that meet the area constraint.
roi = double(roi(areaConstraint, :));

% Show remaining blobs after applying the area constraint.
img = insertShape(I, 'rectangle', roi);
figure;
imshow(img);
```



Further processing based on a region's aspect ratio is applied to identify regions that are likely to contain a single character. This helps to remove the smaller text characters that are jumbled together next to the digits. In general, the larger the text the easier it is for OCR to recognize.

```
% Compute the aspect ratio.
width = roi(:,3);
height = roi(:,4);
aspectRatio = width ./ height;
```

```

% An aspect ratio between 0.25 and 1 is typical for individual characters
% as they are usually not very short and wide or very tall and skinny.
roi = roi( aspectRatio > 0.25 & aspectRatio < 1 ,:);

% Show regions after applying the area and aspect ratio constraints.
img = insertShape(I,'rectangle',roi);
figure;
imshow(img);

```



The remaining regions can be passed into the `ocr` function, which accepts rectangular regions of interest as input. The size of the regions are increased slightly to include additional background pixels around the text characters. This helps to improve the internal heuristics used to determine the polarity of the text on the background (e.g. light text on a dark background vs. dark text on a light background).

```

roi(:,1:2) = roi(:,1:2) - 4;
roi(:,3:4) = roi(:,3:4) + 8;
results = ocr(BW1, roi, 'TextLayout', 'Block');

```

The recognized text can be displayed on the original image using `insertObjectAnnotation`. The `deblank` function is used to remove any trailing characters, such as white space or new lines. There are a few missing classifications in these results (e.g. the digit 8) that are correctable using additional pre-processing techniques.

```

text = deblank( {results.Text} );
img = insertObjectAnnotation(I, 'rectangle', roi, text);

figure;
imshow(img)

```



Although `vision.BlobAnalysis` enabled you to find the digits in the keypad image, it may not work as well for images of natural scenes where there are many objects in addition to the text. For these types of images, the technique shown in the example entitled “Automatically Detect and Recognize Text in Natural Images” on page 4-2 may provide better text detection results.

Summary

This example showed how the `ocr` function can be used to recognize text in images, and how a seemingly easy image for OCR required extra pre-processing steps to produce good results.

References

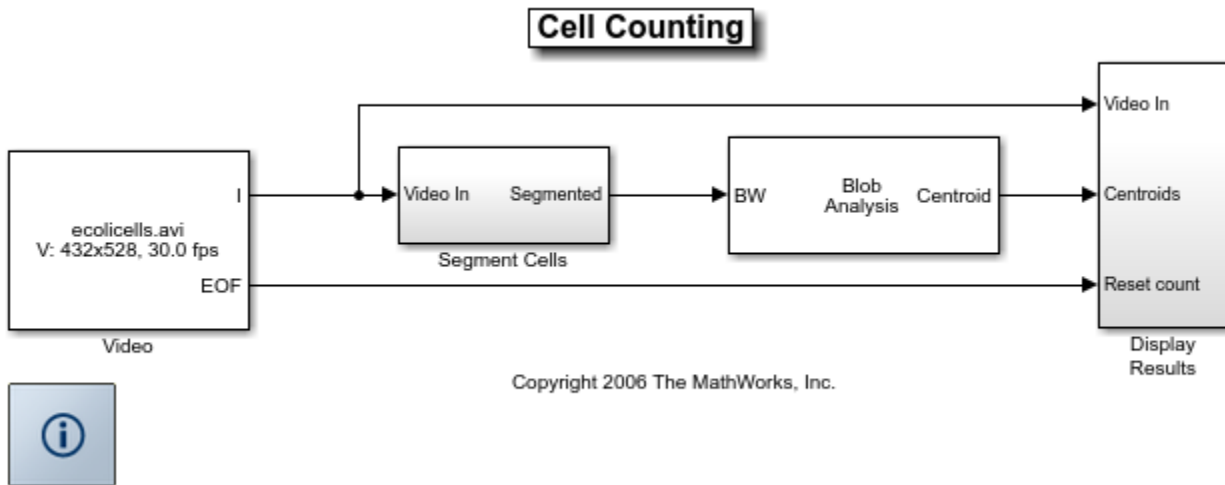
[1] Ray Smith. Hybrid Page Layout Analysis via Tab-Stop Detection. Proceedings of the 10th international conference on document analysis and recognition. 2009.

Cell Counting

This example shows how to use a combination of basic morphological operators and blob analysis to extract information from a video stream. In this case, the example counts the number of E. Coli bacteria in each video frame. Note that the cells are of varying brightness, which makes the task of segmentation more challenging.

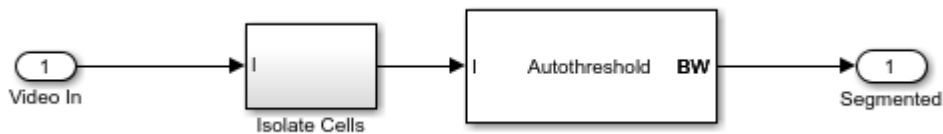
Example Model

The following figure shows the Cell Counting example model.

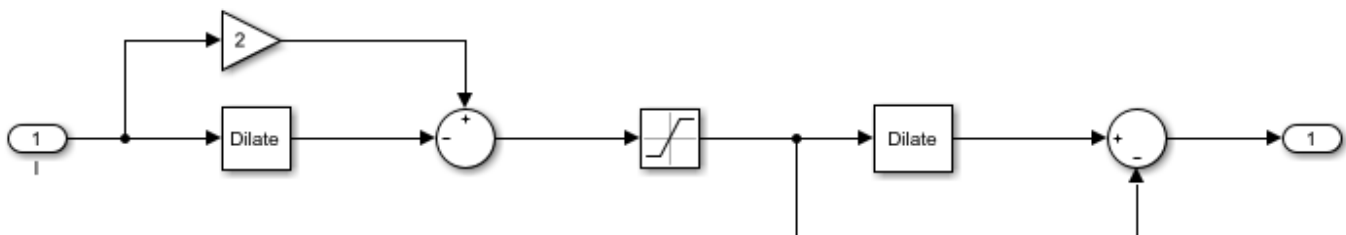


Segment Cells Subsystem

Inside the Isolate Cells subsystem, the example uses a combination of morphological dilation and image arithmetic operations to remove uneven illumination and to emphasize the boundaries between the cells. Due to changes in overall lighting intensity, the example cannot apply a single threshold value to all of the video frames. The example uses the Autothreshold block to compute a threshold for each frame.



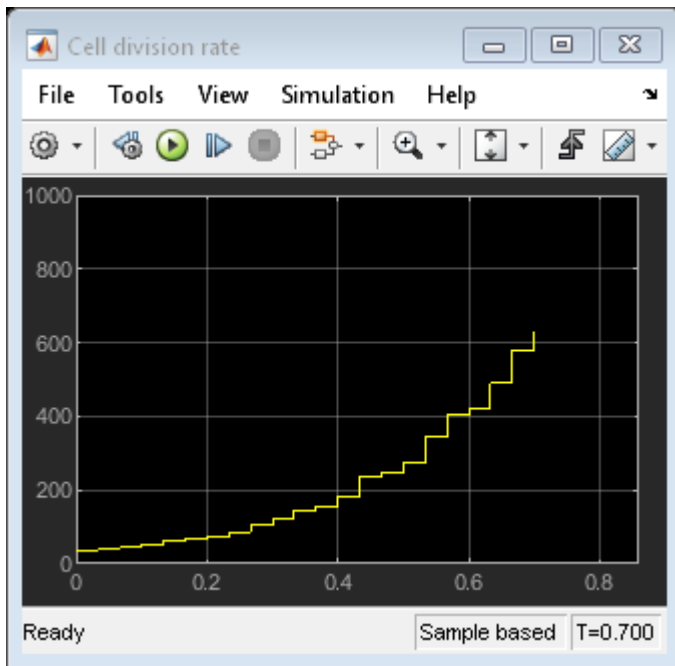
Isolate Cells subsystem:



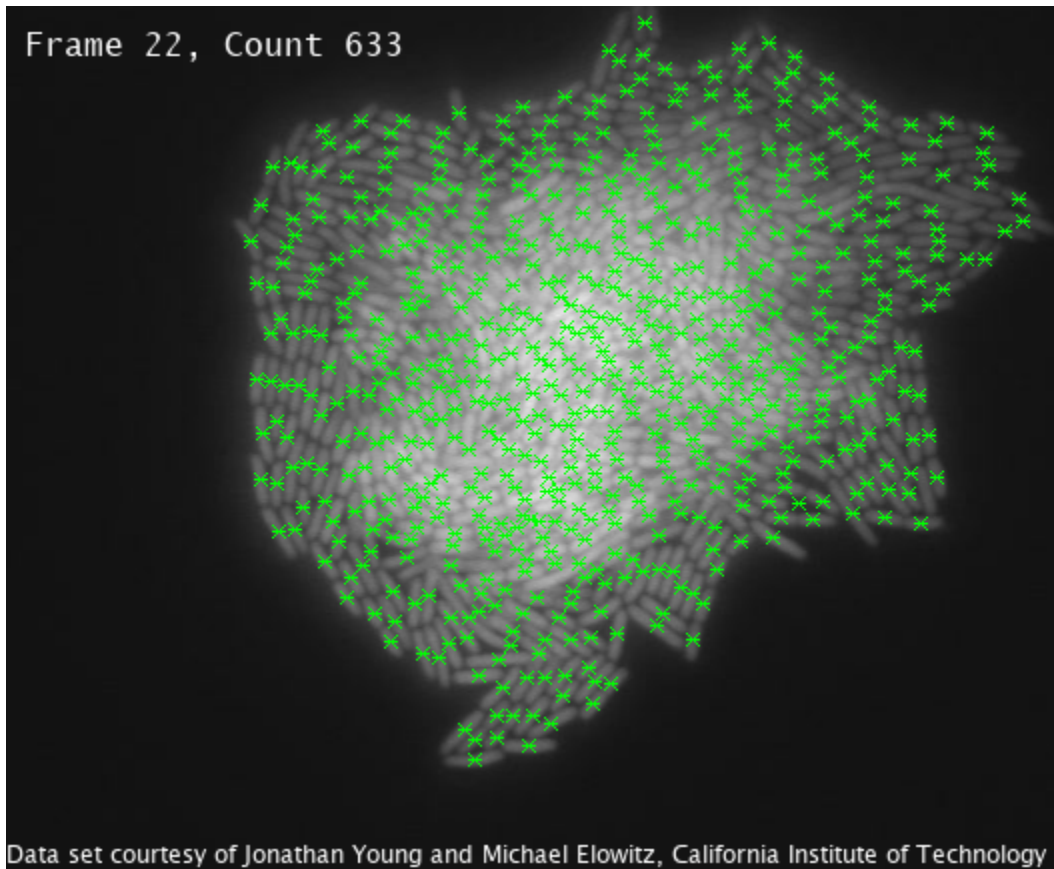
Cell Counting Results

After the example applies the threshold and separates the cells, it uses the Blob Analysis block to count the number of cells in each frame and to calculate the centroid of each cell. The example passes the total number of cells in each frame to the Insert Text block, which is in the Display Results subsystem. This block embeds this information on each video frame.

The Cell division rate window shows the exponential growth of the bacteria.



The Results window displays one frame of the original video and green markers indicating centroid locations of the found cells. The frame number and the number of cells are displayed in the upper left corner.



Data Set Credits

The data set for this example was provided by Jonathan Young and Michael Elowitz from California Institute of Technology®. It is used with permission. For additional information about this data, see

N. Rosenfeld, J. Young, U. Alon, P. Swain, and M.B. Elowitz, "Gene Regulation at the Single-Cell Level," *Science* 2005, Vol. 307, pp. 1962-1965.

Lidar and Point Cloud Processing Examples

- “Augment Point Cloud Data For Deep Learning” on page 5-2
- “Import Point Cloud Data For Deep Learning” on page 5-7
- “Encode Point Cloud Data For Deep Learning” on page 5-11
- “Build a Map from Lidar Data” on page 5-17
- “Build a Map from Lidar Data Using SLAM” on page 5-38
- “3-D Point Cloud Registration and Stitching” on page 5-54

Augment Point Cloud Data For Deep Learning

This example demonstrates how to setup a basic randomized data augmentation pipeline when working with point cloud data in deep learning based workflows. Data augmentation is almost always desirable when working with deep learning because it helps to reduce overfitting during training and can add robustness to types of data transformations which may not be well represented in the original training data.

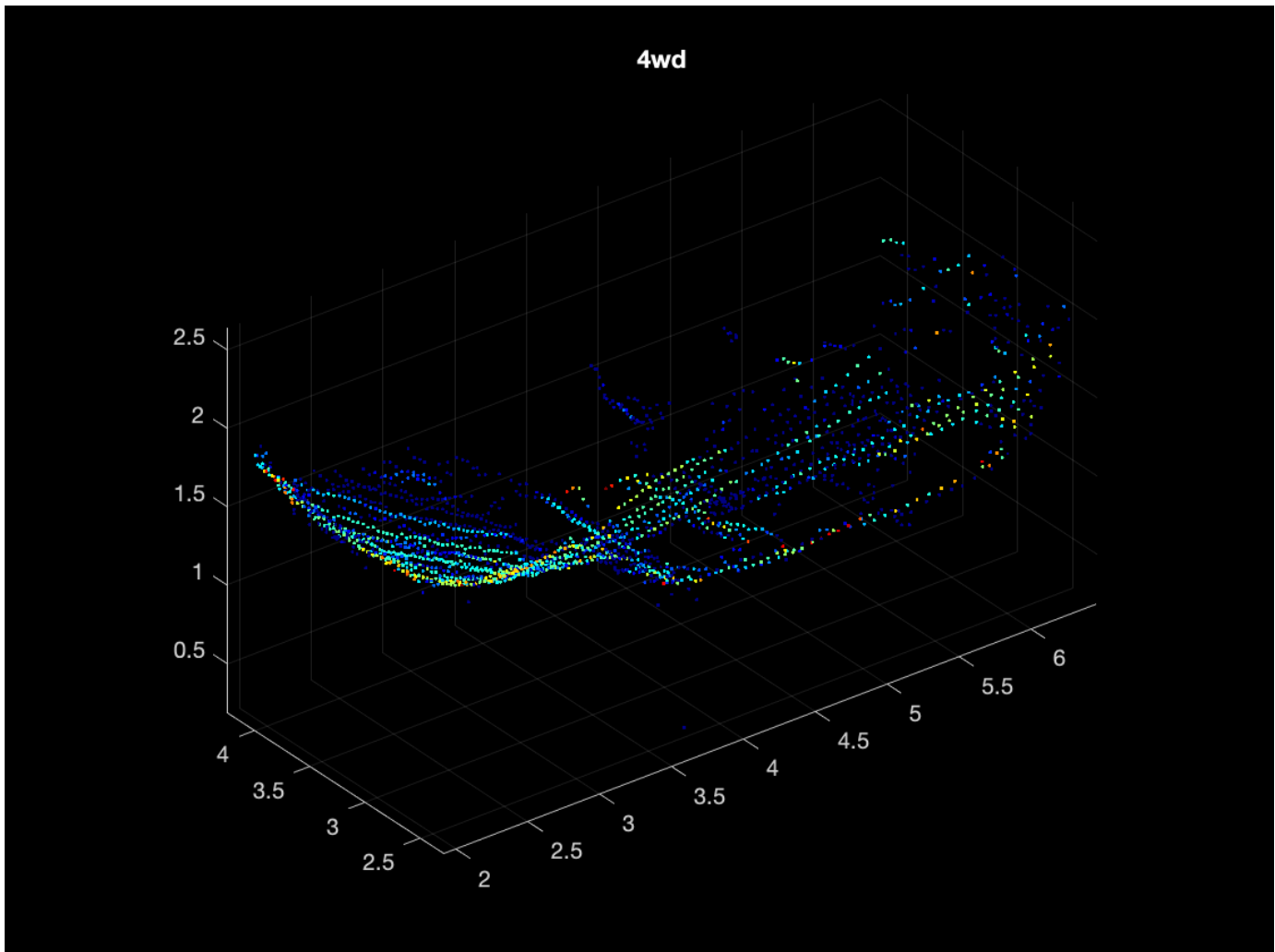
Import point cloud data

```
dataPath = downloadSydneyUrbanObjects(tempdir);  
dsTrain = sydneyUrbanObjectsClassificationDatastore(dataPath);  
dataOut = preview(dsTrain)
```

```
dataOut=1x2 cell array  
    {1x1 pointCloud}    {[4wd]}
```

The datastore `dsTrain` yields a `pointCloud` object and an associated scalar categorical label for each observation.

```
figure  
pcshow(dataOut{1});  
title(dataOut{2});
```



Define augmentation pipeline

The transform function of a datastore is a convenient tool for defining augmentation pipelines.

```
dsAugmented = transform(dsTrain,@augmentPointCloud);
```

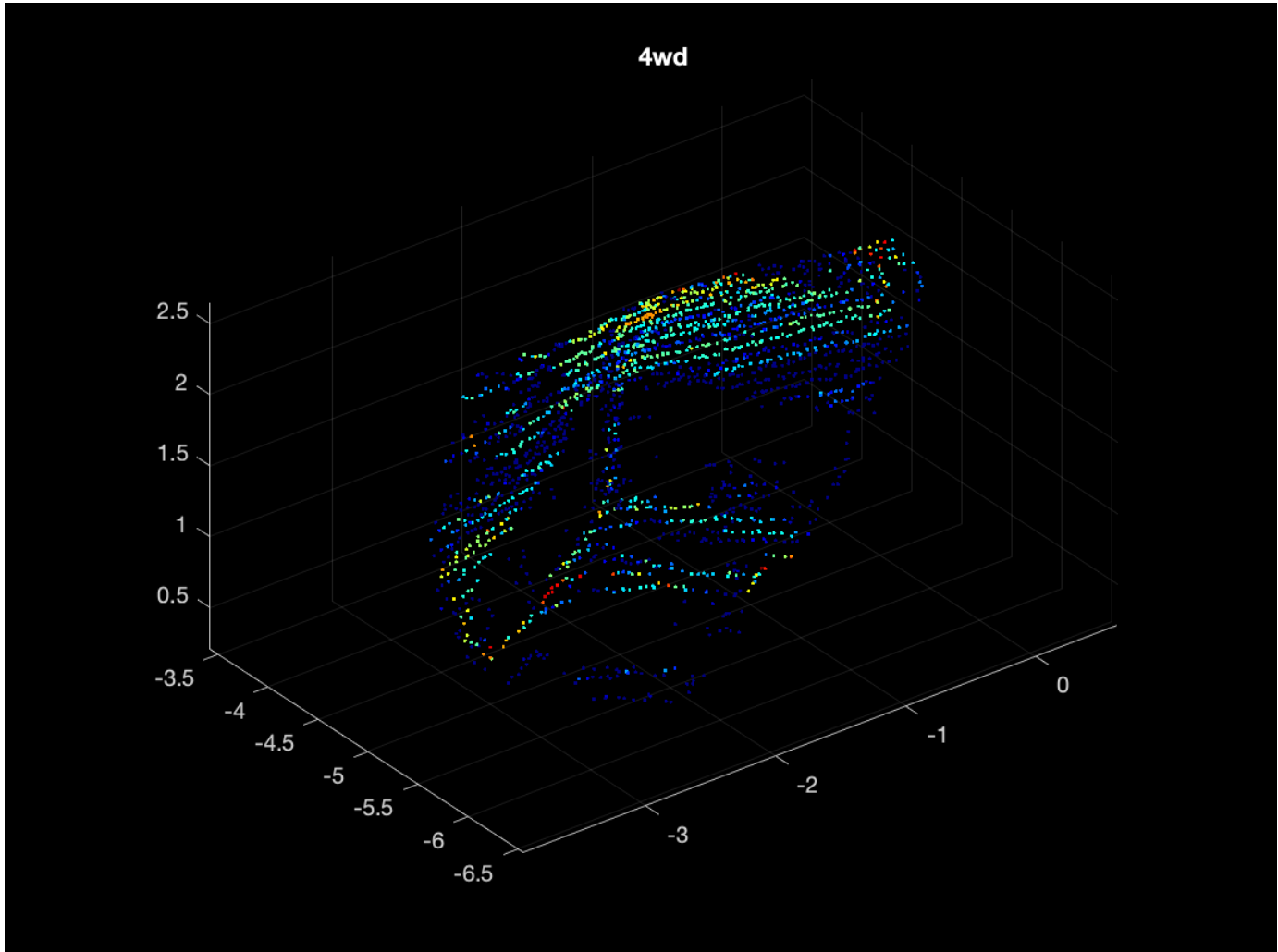
The `augmentPointCloud` function shown below, applies randomized rotation, homogenous scale, randomized reflection across the x- and y-axes, and randomized per point jitter to each observation using the `randomAffine3d` function to construct randomized affine transformations and the `pcttransform` function to apply these transformations to each input point cloud.

```
dataOut = preview(dsAugmented)
```

```
dataOut=1x2 cell array
    {1x1 pointCloud}    {[4wd]}
```

It is always a good idea to visually inspect the data that comes out of any augmentation that is done on training data to make sure that the data looks as expected. The point cloud below is the same as the original shown previously, but with randomized affine warping with per point jitter added.

```
figure
pcshow(dataOut{1});
title(dataOut{2});
```



The resulting `TransformedDatastore` and `dsAugmented` can be passed to deep learning functions including `trainNetwork`, `predict`, and `classify` for use in training and inference.

Supporting Functions

```
function datasetPath = downloadSydneyUrbanObjects(dataLoc)

if nargin == 0
    dataLoc = pwd();
end

dataLoc = string(dataLoc);

url = "http://www.acfr.usyd.edu.au/papers/data/";
name = "sydney-urban-objects-dataset.tar.gz";

if ~exist(fullfile(dataLoc, 'sydney-urban-objects-dataset'), 'dir')
```

```

        disp('Downloading Sydney Urban Objects Dataset...');
        untar(fullfile(url,name),dataLoc);
    end

datasetPath = dataLoc.append('sydney-urban-objects-dataset');

end

function ds = sydneyUrbanObjectsClassificationDatastore(datapath,folds)
% sydneyUrbanObjectsClassificationDatastore Datastore with point clouds and
% associated categorical labels for Sydney Urban Objects dataset.
%
% ds = sydneyUrbanObjectsDatastore(datapath) constructs a datastore that
% represents point clouds and associated categories for the Sydney Urban
% Objects dataset. The input, datapath, is a string or char array which
% represents the path to the root directory of the Sydney Urban Objects
% Dataset.
%
% ds = sydneyUrbanObjectsDatastore(___,folds) optionally allows
% specification of desired folds that you wish to be included in the
% output ds. For example, [1 2 4] specifies that you want the first,
% second, and fourth folds of the Dataset. Default: [1 2 3 4].

if nargin < 2
    folds = 1:4;
end

datapath = string(datapath);
path = fullfile(datapath,'objects',filesep);

% For now, include all folds in Datastore
foldNames{1} = importdata(fullfile(datapath,'folds','fold0.txt'));
foldNames{2} = importdata(fullfile(datapath,'folds','fold1.txt'));
foldNames{3} = importdata(fullfile(datapath,'folds','fold2.txt'));
foldNames{4} = importdata(fullfile(datapath,'folds','fold3.txt'));
names = foldNames(folds);
names = vertcat(names{:});

fullfilenames = append(path,names);
ds = fileDatastore(fullfilenames,'ReadFcn',@extractTrainingData,'FileExtensions','.bin');

end

function dataOut = extractTrainingData(fname)

[pointData,intensity] = readbin(fname);

[~,name] = fileparts(fname);
name = string(name);
name = extractBefore(name, '.');

labelNames = ["4wd","bench","bicycle","biker",...
    "building","bus","car","cyclist","excavator","pedestrian","pillar",...
    "pole","post","scooter","ticket_machine","traffic_lights","traffic_sign",...
    "trailer","trash","tree","truck","trunk","umbrella","ute","van","vegetation"];

label = categorical(name,labelNames);

```

```

dataOut = {pointCloud(pointData, 'Intensity', intensity), label};

end

function [pointData, intensity] = readbin(fname)
% readbin Read point and intensity data from Sydney Urban Object binary
% files.

% names = ['t', 'intensity', 'id', ...
%         'x', 'y', 'z', ...
%         'azimuth', 'range', 'pid']
%
% formats = ['int64', 'uint8', 'uint8', ...
%           'float32', 'float32', 'float32', ...
%           'float32', 'float32', 'int32']

fid = fopen(fname, 'r');
c = onCleanup(@() fclose(fid));

fseek(fid, 10, -1); % Move to the first X point location 10 bytes from beginning
X = fread(fid, inf, 'single', 30);
fseek(fid, 14, -1);
Y = fread(fid, inf, 'single', 30);
fseek(fid, 18, -1);
Z = fread(fid, inf, 'single', 30);

fseek(fid, 8, -1);
intensity = fread(fid, inf, 'uint8', 33);

pointData = [X, Y, Z];

end

function dataOut = augmentPointCloud(data)

ptCloud = data{1};
label = data{2};

% Apply randomized rotation about Z axis.
tform = randomAffine3d('Rotation', @() deal([0 0 1], 360*rand), 'Scale', [0.98, 1.02], 'XReflection', t);
ptCloud = pctransform(ptCloud, tform);

% Apply jitter to each point in point cloud
amountOfJitter = 0.01;
numPoints = size(ptCloud.Location, 1);
D = zeros(size(ptCloud.Location), 'like', ptCloud.Location);
D(:, 1) = diff(ptCloud.XLimits)*rand(numPoints, 1);
D(:, 2) = diff(ptCloud.YLimits)*rand(numPoints, 1);
D(:, 3) = diff(ptCloud.ZLimits)*rand(numPoints, 1);
D = amountOfJitter.*D;
ptCloud = pctransform(ptCloud, D);

dataOut = {ptCloud, label};

end

```


Import Point Cloud Data For Deep Learning

To use point cloud data in deep learning workflows, the data must be read in from its raw form in a data set into MATLAB. In this example, we are working with the Sydney Urban Objects Dataset [1 on page 5-0]. This example shows how to use MATLAB Datastores to read in and represent data for deep learning.

Download Sydney Urban Objects Dataset

The Sydney Urban Objects data is 122 MB in its uncompressed form and may take a few moments to download depending on your network connection speed.

```
sydneyUrbanObjectsPath = downloadSydneyUrbanObjects(tempdir());
```

Define Datastore For Point Cloud Data

Construct a Datastore that is configured for reading in the point clouds from the /objects folder in Sydney Urban Objects, along with associated object labels.

```
ds = sydneyUrbanObjectsClassificationDatastore(sydneyUrbanObjectsPath);
```

Read and display the first observation from the Datastore.

```
data = preview(ds)
```

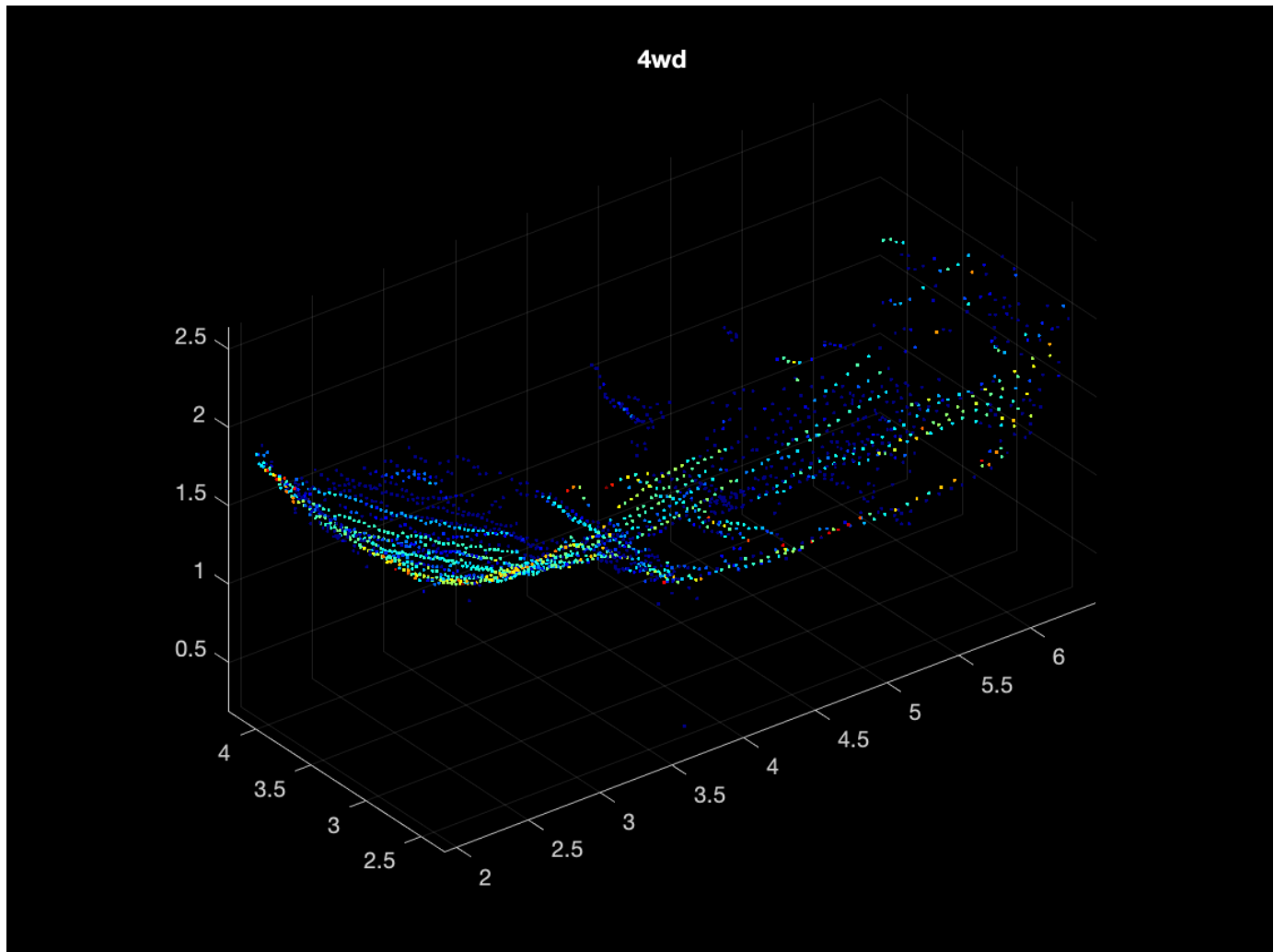
```
data=1x2 cell array
    {1x1 pointCloud}    {[4wd]}
```

```
disp(data)
```

```
    {1x1 pointCloud}    {[4wd]}
```

The output of the read and preview methods of the Datastore is a cell array in which the first column is a pointCloud object and the second column is the associated class label. A pointCloud object can be visualized using the pcshow function.

```
figure
pcshow(data{1})
title(string(data{2}))
```



References

[1] Alastair Quadros, James Underwood, Bertrand Douillard; 2013. Sydney Urban Objects Dataset.

Supporting Functions

```
function datasetPath = downloadSydneyUrbanObjects(dataLoc)
% This function downloads the syntax urban objects tar archive to tempdir
% provides as output the location of where the data was saved.

if nargin == 0
    dataLoc = pwd();
end

dataLoc = string(dataLoc);

url = "http://www.acfr.usyd.edu.au/papers/data/";
name = "sydney-urban-objects-dataset.tar.gz";

if ~exist(fullfile(dataLoc, 'sydney-urban-objects-dataset'), 'dir')
    disp('Downloading Sydney Urban Objects Dataset...');
```

```

        untar(fullfile(url,name),dataLoc);
    end

    datasetPath = dataLoc.append('sydney-urban-objects-dataset');

    end

    function ds = sydneyUrbanObjectsClassificationDatastore(datapath,folds)
    % sydneyUrbanObjectsClassificationDatastore Datastore with point clouds and
    % associated categorical labels for Sydney Urban Objects dataset.
    %
    % ds = sydneyUrbanObjectsDatastore(datapath) constructs a datastore that
    % represents point clouds and associated categories for the Sydney Urban
    % Objects dataset. The input, datapath, is a string or char array which
    % represents the path to the root directory of the Sydney Urban Objects
    % Dataset.
    %
    % ds = sydneyUrbanObjectsDatastore(___,folds) optionally allows
    % specification of desired folds that you wish to be included in the
    % output ds. For example, [1 2 4] specifies that you want the first,
    % second, and fourth folds of the Dataset. Default: [1 2 3 4].

    if nargin < 2
        folds = 1:4;
    end

    datapath = string(datapath);
    path = fullfile(datapath,'objects',filesep);

    % For now, include all folds in Datastore
    foldNames{1} = importdata(fullfile(datapath,'folds','fold0.txt'));
    foldNames{2} = importdata(fullfile(datapath,'folds','fold1.txt'));
    foldNames{3} = importdata(fullfile(datapath,'folds','fold2.txt'));
    foldNames{4} = importdata(fullfile(datapath,'folds','fold3.txt'));
    names = foldNames(folds);
    names = vertcat(names{:});

    fullfilenames = append(path,names);
    ds = fileDatastore(fullfilenames,'ReadFcn',@extractTrainingData,'FileExtensions','.bin');

    end

    function dataOut = extractTrainingData(fname)

    [pointData,intensity] = readbin(fname);

    [~,name] = fileparts(fname);
    name = string(name);
    name = extractBefore(name, '.');

    labelNames = ["4wd","bench","bicycle","biker",...
        "building","bus","car","cyclist","excavator","pedestrian","pillar",...
        "pole","post","scooter","ticket_machine","traffic_lights","traffic_sign",...
        "trailer","trash","tree","truck","trunk","umbrella","ute","van","vegetation"];

    label = categorical(name,labelNames);

    dataOut = {pointCloud(pointData,'Intensity',intensity),label};

```

```
end

function [pointData,intensity] = readbin(fname)
% readbin Read point and intensity data from Sydney Urban Object binary
% files.

% names = ['t','intensity','id',...
%         'x','y','z',...
%         'azimuth','range','pid']
%
% formats = ['int64', 'uint8', 'uint8',...
%           'float32', 'float32', 'float32',...
%           'float32', 'float32', 'int32']

fid = fopen(fname, 'r');
c = onCleanup(@() fclose(fid));

fseek(fid,10,-1); % Move to the first X point location 10 bytes from beginning
X = fread(fid,inf,'single',30);
fseek(fid,14,-1);
Y = fread(fid,inf,'single',30);
fseek(fid,18,-1);
Z = fread(fid,inf,'single',30);

fseek(fid,8,-1);
intensity = fread(fid,inf,'uint8',33);

pointData = [X,Y,Z];

end
```

Encode Point Cloud Data For Deep Learning

When using convolutional neural networks with point cloud data, certain core operations like convolution require input data that is regularly sampled spatially. The irregular spatial sampling of point cloud and lidar data must be transformed into some regularly sampled structure at some point in the preprocessing pipeline. There are many different approaches to how point cloud data is transformed into a dense, gridded structure [1 on page 5-0][2 on page 5-0][3 on page 5-0]. This example demonstrates a simple approach known as voxelization.

Voxelization of Point Cloud Data

Start by defining a datastore for working with the Sydney Urban Objects Dataset.

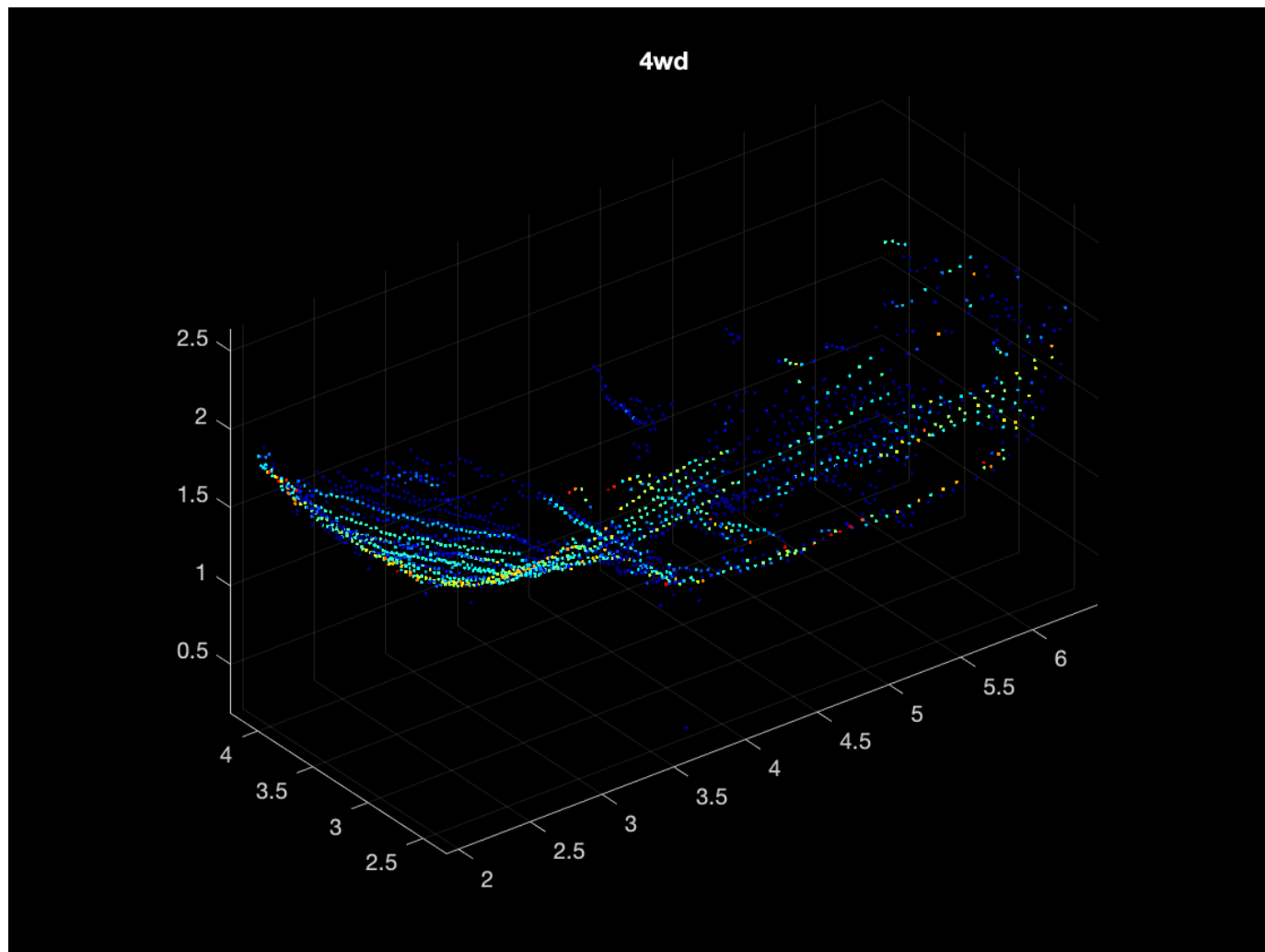
```
dataPath = downloadSydneyUrbanObjects(tempdir);  
ds = sydneyUrbanObjectsClassificationDatastore(dataPath);
```

Obtain sample output data from datastore.

```
data = preview(ds);  
disp(data)  
  
    {1×1 pointCloud}    {[4wd]}
```

View sample output data from datastore

```
figure  
ptCloud = data{1};  
pcshow(ptCloud);  
label = string(data{2});  
title(label);
```



Use the `pcbin` function to define a desired regular 3-D gridding of the coordinate system of an input `pointCloud` object. Use `pcbin` to also return an output cell array that contains spatial bin locations for each point in the input `pointCloud`. In this case, the input `pointCloud` is binned in a `[32,32,32]` size output grid that spans the `XLimits`, `YLimits`, and `ZLimits` of the input `pointCloud`.

```
outputGridSize = [32,32,32];
bins = pcbin(data{1},outputGridSize);
```

Each cell in `bins` contains the indices of the points in `ptCloud.Location` that fall in a particular point location. The MATLAB function `cellfun` can be used to define common encodings of point cloud data using `bins` as input.

```
occupancyGrid = cellfun(@(c) ~isempty(c),bins);
```

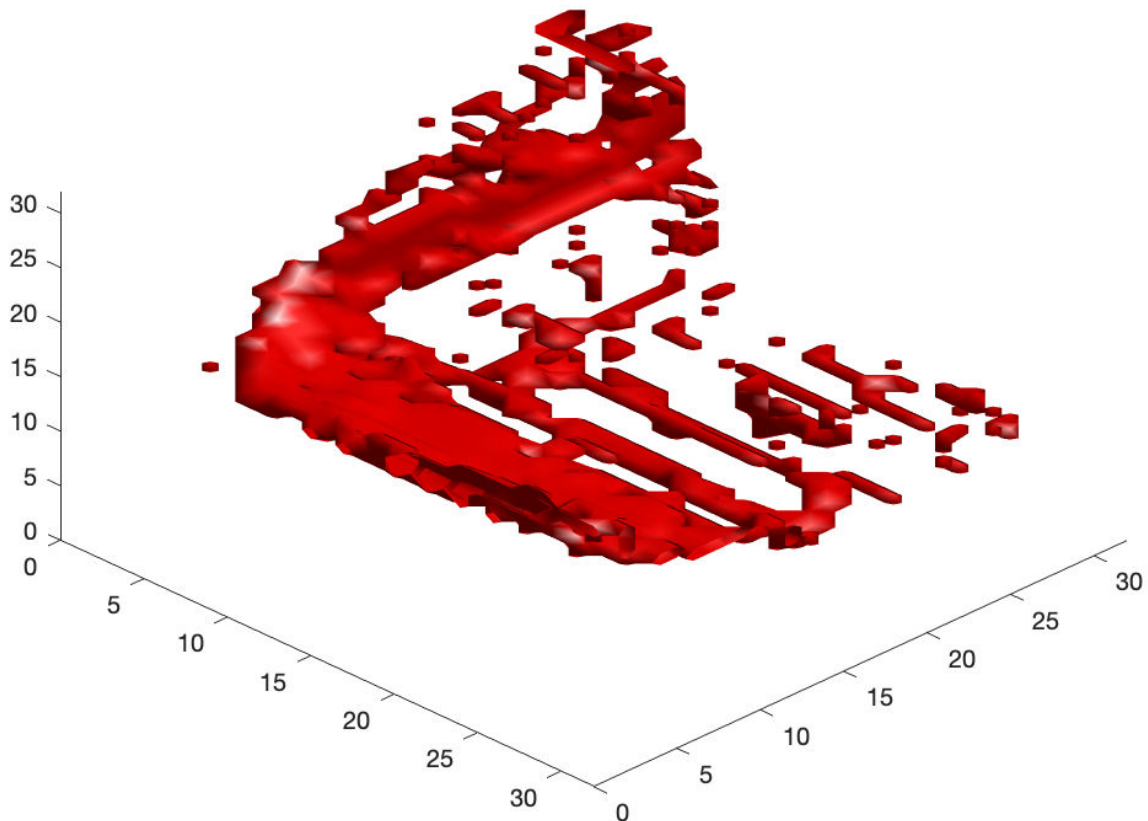
Define a 3-D occupancy grid which is true for grid locations that are occupied by at least one point and false otherwise.

```
figure;
p = patch(isosurface(occupancyGrid,0.5));
view(45,45);
```

```

p.FaceColor = 'red';
p.EdgeColor = 'none';
camlight;
lighting phong

```



Transform Datastore to Apply Point Cloud Encoding to Entire Dataset

Use the `transform` function of datastore to apply a simple occupancy grid encoding to every observation in an input datastore. The `formOccupancyGrid` function, which is included in the supporting functions section, uses the exact same approach shown above with `pcbin`.

```

dsTransformed = transform(ds,@formOccupancyGrid);
exampleOutputData = preview(dsTransformed);
disp(exampleOutputData);

```

```

{32×32×32 logical}    {[4wd]}

```

The resulting datastore, `dsTransformed`, can be passed to deep learning interfaces including `trainNetwork` and `DataLoader` for use in training deep neural networks.

References

- [1] Maturana, D. and Scherer, S., VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition, IROS 2015.
- [2] AH Lang, S Vora, H Caesar, L Zhou, J Yang, O Beijbom, PointPillars: Fast Encoders for Object Detection from Point Clouds, CVPR 2019
- [3] Charles R. Qi, Hao Su, Kaichun Mo, Leonidas J. Guibas, PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation, CVPR 2017

Supporting Functions

```
function datasetPath = downloadSydneyUrbanObjects(dataLoc)

if nargin == 0
    dataLoc = pwd();
end

dataLoc = string(dataLoc);

url = "http://www.acfr.usyd.edu.au/papers/data/";
name = "sydney-urban-objects-dataset.tar.gz";

if ~exist(fullfile(dataLoc,'sydney-urban-objects-dataset'),'dir')
    disp('Downloading Sydney Urban Objects Dataset...');
    untar(fullfile(url,name),dataLoc);
end

datasetPath = dataLoc.append('sydney-urban-objects-dataset');

end

function ds = sydneyUrbanObjectsClassificationDatastore(datapath,folds)
% sydneyUrbanObjectsClassificationDatastore Datastore with point clouds and
% associated categorical labels for Sydney Urban Objects dataset.
%
% ds = sydneyUrbanObjectsDatastore(datapath) constructs a datastore that
% represents point clouds and associated categories for the Sydney Urban
% Objects dataset. The input, datapath, is a string or char array which
% represents the path to the root directory of the Sydney Urban Objects
% Dataset.
%
% ds = sydneyUrbanObjectsDatastore(___,folds) optionally allows
% specification of desired folds that you wish to be included in the
% output ds. For example, [1 2 4] specifies that you want the first,
% second, and fourth folds of the Dataset. Default: [1 2 3 4].

if nargin < 2
    folds = 1:4;
end

datapath = string(datapath);
path = fullfile(datapath,'objects',filesep);

% For now, include all folds in Datastore
foldNames{1} = importdata(fullfile(datapath,'folds','fold0.txt'));
foldNames{2} = importdata(fullfile(datapath,'folds','fold1.txt'));
```



```

foldNames{3} = importdata(fullfile(datapath,'folds','fold2.txt'));
foldNames{4} = importdata(fullfile(datapath,'folds','fold3.txt'));
names = foldNames(folds);
names = vertcat(names{:});

fullfilenames = append(path,names);
ds = fileDatastore(fullfileNames,'ReadFcn',@extractTrainingData,'FileExtensions','.bin');

end

function dataOut = extractTrainingData(fname)

[pointData,intensity] = readbin(fname);

[~,name] = fileparts(fname);
name = string(name);
name = extractBefore(name, '.');

labelNames = ["4wd","bench","bicycle","biker",...
    "building","bus","car","cyclist","excavator","pedestrian","pillar",...
    "pole","post","scooter","ticket_machine","traffic_lights","traffic_sign",...
    "trailer","trash","tree","truck","trunk","umbrella","ute","van","vegetation"];

label = categorical(name,labelNames);

dataOut = {pointCloud(pointData,'Intensity',intensity),label};

end

function [pointData,intensity] = readbin(fname)
% readbin Read point and intensity data from Sydney Urban Object binary
% files.

% names = ['t','intensity','id',...
%         'x','y','z',...
%         'azimuth','range','pid']
%
% formats = ['int64','uint8','uint8',...
%           'float32','float32','float32',...
%           'float32','float32','int32']

fid = fopen(fname, 'r');
c = onCleanup(@() fclose(fid));

fseek(fid,10,-1); % Move to the first X point location 10 bytes from beginning
X = fread(fid,inf,'single',30);
fseek(fid,14,-1);
Y = fread(fid,inf,'single',30);
fseek(fid,18,-1);
Z = fread(fid,inf,'single',30);

fseek(fid,8,-1);
intensity = fread(fid,inf,'uint8',33);

pointData = [X,Y,Z];

end

```

```
function dataOut = formOccupancyGrid(data)

grid = pccbin(data{1},[32 32 32]);
occupancyGrid = cellfun(@(c) ~isempty(c),grid);
label = data{2};
dataOut = {occupancyGrid,label};

end
```

Build a Map from Lidar Data

This example shows how to process 3-D lidar data from a sensor mounted on a vehicle to progressively build a map, with assistance from inertial measurement unit (IMU) readings. Such a map can facilitate path planning for vehicle navigation or can be used for localization. For evaluating the generated map, this example also shows how to compare the trajectory of the vehicle against global positioning system (GPS) recording.

Overview

High Definition (HD) maps are mapping services that provide precise geometry of roads up to a few centimeters in accuracy. This level of accuracy makes HD maps suitable for automated driving workflows such as localization and navigation. Such HD maps are generated by building a map from 3-D lidar scans, in conjunction with high-precision GPS and or IMU sensors and can be used to localize a vehicle within a few centimeters. This example implements a subset of features required to build such a system.

In this example, you learn how to:

- Load, explore and visualize recorded driving data
- Build a map using lidar scans
- Improve the map using IMU readings

Load and Explore Recorded Driving Data

The data used in this example is from this GitHub® repository, and represents approximately 100 seconds of lidar, GPS and IMU data. The data is saved in the form of MAT-files, each containing a `timetable`. Download the MAT-files from the repository and load them into the MATLAB® workspace.

Note: This download can take a few minutes.

```
baseDownloadURL = 'https://github.com/mathworks/udacity-self-driving-data-subset/raw/master/drive';
dataFolder      = fullfile(tempdir, 'drive_segment_11_18_16', filesep);
options         = weboptions('Timeout', Inf);

lidarFileName = dataFolder + "lidarPointClouds.mat";
imuFileName   = dataFolder + "imuOrientations.mat";
gpsFileName   = dataFolder + "gpsSequence.mat";

folderExists = exist(dataFolder, 'dir');
matfilesExist = exist(lidarFileName, 'file') && exist(imuFileName, 'file') ...
    && exist(gpsFileName, 'file');

if ~folderExists
    mkdir(dataFolder);
end

if ~matfilesExist
    disp('Downloading lidarPointClouds.mat (613 MB)...')
    websave(lidarFileName, baseDownloadURL + "lidarPointClouds.mat", options);

    disp('Downloading imuOrientations.mat (1.2 MB)...')
    websave(imuFileName, baseDownloadURL + "imuOrientations.mat", options);
end
```

```

disp('Downloading gpsSequence.mat (3 KB)...')
websave(gpsFileName, baseDownloadURL + "gpsSequence.mat", options);
end

```

```

Downloading lidarPointClouds.mat (613 MB)...
Downloading imuOrientations.mat (1.2 MB)...
Downloading gpsSequence.mat (3 KB)...

```

First, load the point cloud data saved from a Velodyne® HDL32E lidar. Each scan of lidar data is stored as a 3-D point cloud using the `pointCloud` object. This object internally organizes the data using a K-d tree data structure for faster search. The timestamp associated with each lidar scan is recorded in the `Time` variable of the timetable.

```

% Load lidar data from MAT-file
data = load(lidarFileName);
lidarPointClouds = data.lidarPointClouds;

% Display first few rows of lidar data
head(lidarPointClouds)

```

ans =

8×1 timetable

Time	PointCloud
23:46:10.5115	[1×1 pointCloud]
23:46:10.6115	[1×1 pointCloud]
23:46:10.7116	[1×1 pointCloud]
23:46:10.8117	[1×1 pointCloud]
23:46:10.9118	[1×1 pointCloud]
23:46:11.0119	[1×1 pointCloud]
23:46:11.1120	[1×1 pointCloud]
23:46:11.2120	[1×1 pointCloud]

Load the GPS data from the MAT-file. The `Latitude`, `Longitude`, and `Altitude` variables of the `timetable` are used to store the geographic coordinates recorded by the GPS device on the vehicle.

```

% Load GPS sequence from MAT-file
data = load(gpsFileName);
gpsSequence = data.gpsSequence;

% Display first few rows of GPS data
head(gpsSequence)

```

ans =

8×3 timetable

Time	Latitude	Longitude	Altitude
23:46:11.4563	37.4	-122.11	-42.5
23:46:12.4563	37.4	-122.11	-42.5

```

23:46:13.4565    37.4    -122.11    -42.5
23:46:14.4455    37.4    -122.11    -42.5
23:46:15.4455    37.4    -122.11    -42.5
23:46:16.4567    37.4    -122.11    -42.5
23:46:17.4573    37.4    -122.11    -42.5
23:46:18.4656    37.4    -122.11    -42.5

```

Load the IMU data from the MAT-file. An IMU typically consists of individual sensors that report information about the motion of the vehicle. They combine multiple sensors, including accelerometers, gyroscopes and magnetometers. The `Orientation` variable stores the reported orientation of the IMU sensor. These readings are reported as quaternions. Each reading is specified as a 1-by-4 vector containing the four quaternion parts. Convert the 1-by-4 vector to a quaternion (Automated Driving Toolbox) object.

```

% Load IMU recordings from MAT-file
data = load(imuFileName);
imuOrientations = data.imuOrientations;

% Convert IMU recordings to quaternion type
imuOrientations = convertvars(imuOrientations, 'Orientation', 'quaternion');

% Display first few rows of IMU data
head(imuOrientations)

```

```
ans =
```

```

8x1 timetable

      Time      Orientation
-----
23:46:11.4570 [1x1 quaternion]
23:46:11.4605 [1x1 quaternion]
23:46:11.4620 [1x1 quaternion]
23:46:11.4655 [1x1 quaternion]
23:46:11.4670 [1x1 quaternion]
23:46:11.4705 [1x1 quaternion]
23:46:11.4720 [1x1 quaternion]
23:46:11.4755 [1x1 quaternion]

```

To understand how the sensor readings come in, for each sensor, compute the approximate frame duration.

```

lidarFrameDuration = median(diff(lidarPointClouds.Time));
gpsFrameDuration   = median(diff(gpsSequence.Time));
imuFrameDuration   = median(diff(imuOrientations.Time));

% Adjust display format to seconds
lidarFrameDuration.Format = 's';
gpsFrameDuration.Format   = 's';
imuFrameDuration.Format   = 's';

% Compute frame rates
lidarRate = 1/seconds(lidarFrameDuration);
gpsRate   = 1/seconds(gpsFrameDuration);

```

```
imuRate = 1/seconds(imuFrameDuration);

% Display frame durations and rates
fprintf('Lidar: %s, %3.1f Hz\n', char(lidarFrameDuration), lidarRate);
fprintf('GPS : %s, %3.1f Hz\n', char(gpsFrameDuration), gpsRate);
fprintf('IMU : %s, %3.1f Hz\n', char(imuFrameDuration), imuRate);

Lidar: 0.10008 sec, 10.0 Hz
GPS : 1.0001 sec, 1.0 Hz
IMU : 0.002493 sec, 401.1 Hz
```

The GPS sensor is the slowest, running at a rate close to 1 Hz. The lidar is next slowest, running at a rate close to 10 Hz, followed by the IMU at a rate of almost 400 Hz.

Visualize Driving Data

To understand what the scene contains, visualize the recorded data using streaming players. To visualize the GPS readings, use `geoplayer` (Automated Driving Toolbox). To visualize lidar readings using `pcplayer`.

```
% Create a geoplayer to visualize streaming geographic coordinates
latCenter = gpsSequence.Latitude(1);
lonCenter = gpsSequence.Longitude(1);
zoomLevel = 17;

gpsPlayer = geoplayer(latCenter, lonCenter, zoomLevel);

% Plot the full route
plotRoute(gpsPlayer, gpsSequence.Latitude, gpsSequence.Longitude);

% Determine limits for the player
xlims = [-45 45]; % meters
ylims = [-45 45];
zlims = [-10 20];

% Create a pcplayer to visualize streaming point clouds from lidar sensor
lidarPlayer = pcplayer(xlims, ylims, zlims);

% Customize player axes labels
xlabel(lidarPlayer.Axes, 'X (m)')
ylabel(lidarPlayer.Axes, 'Y (m)')
zlabel(lidarPlayer.Axes, 'Z (m)')

title(lidarPlayer.Axes, 'Lidar Sensor Data')

% Align players on screen
helperAlignPlayers({gpsPlayer, lidarPlayer});

% Outer loop over GPS readings (slower signal)
for g = 1 : height(gpsSequence)-1

    % Extract geographic coordinates from timetable
    latitude = gpsSequence.Latitude(g);
    longitude = gpsSequence.Longitude(g);

    % Update current position in GPS display
    plotPosition(gpsPlayer, latitude, longitude);
```

```
% Compute the time span between the current and next GPS reading
timeSpan = timerange(gpsSequence.Time(g), gpsSequence.Time(g+1));

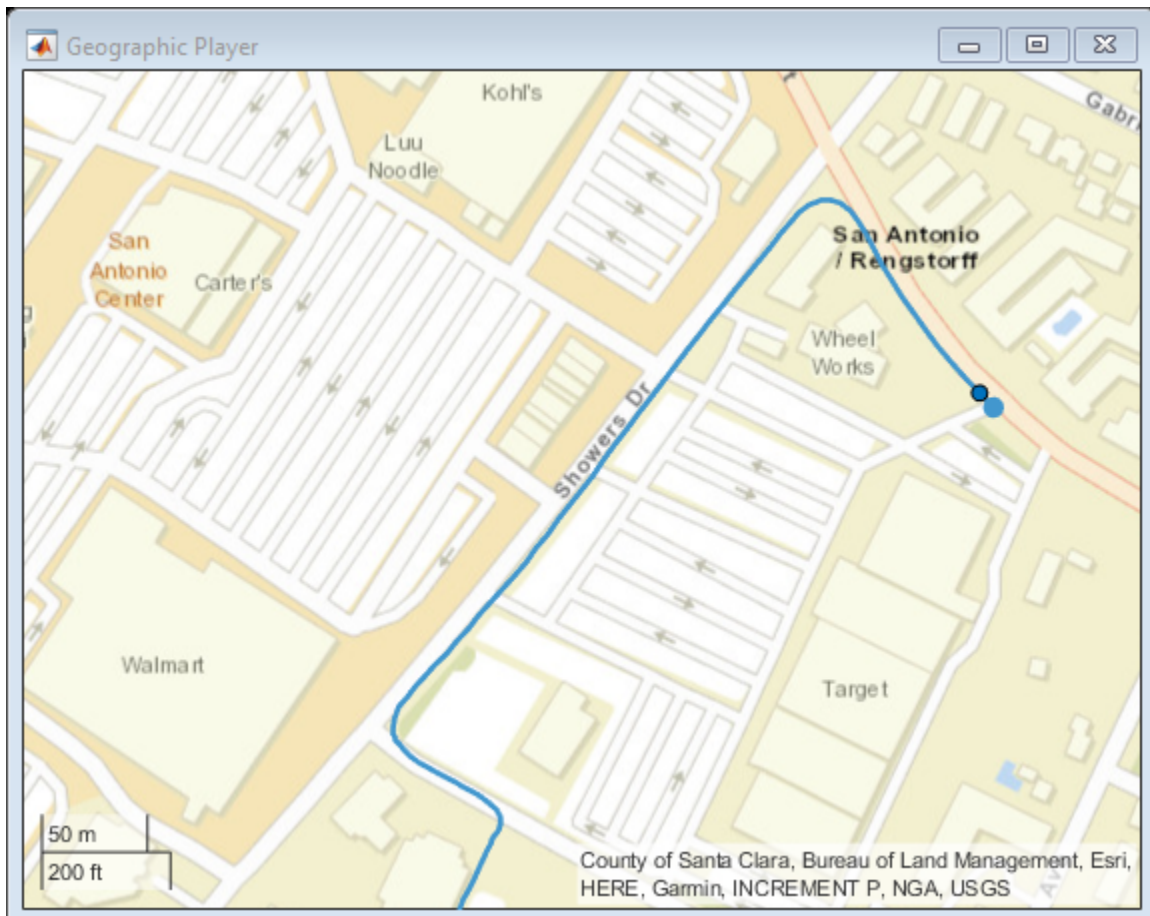
% Extract the lidar frames recorded during this time span
lidarFrames = lidarPointClouds(timeSpan, :);

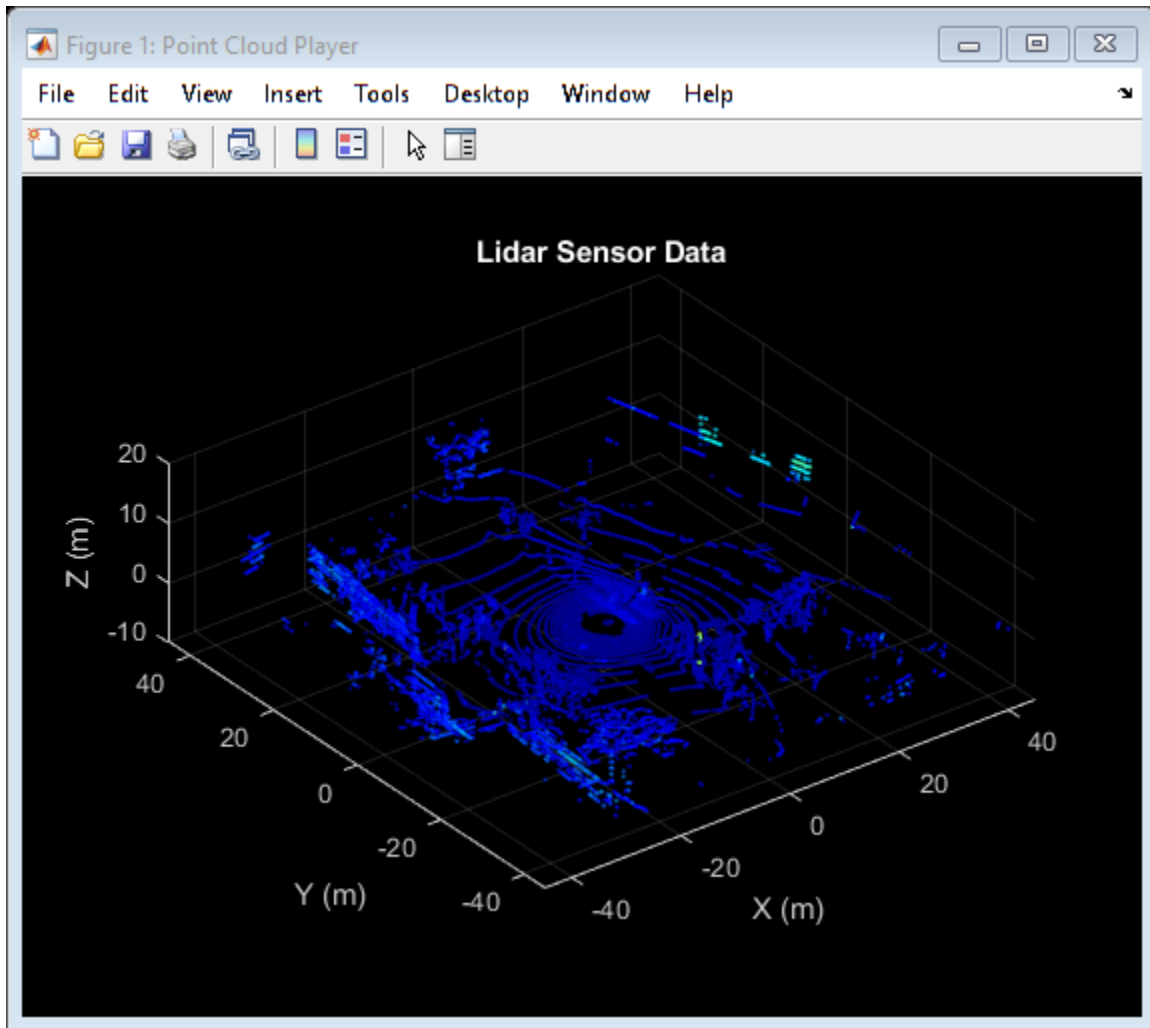
% Inner loop over lidar readings (faster signal)
for l = 1 : height(lidarFrames)

    % Extract point cloud
    ptCloud = lidarFrames.PointCloud(l);

    % Update lidar display
    view(lidarPlayer, ptCloud);

    % Pause to slow down the display
    pause(0.01)
end
end
```





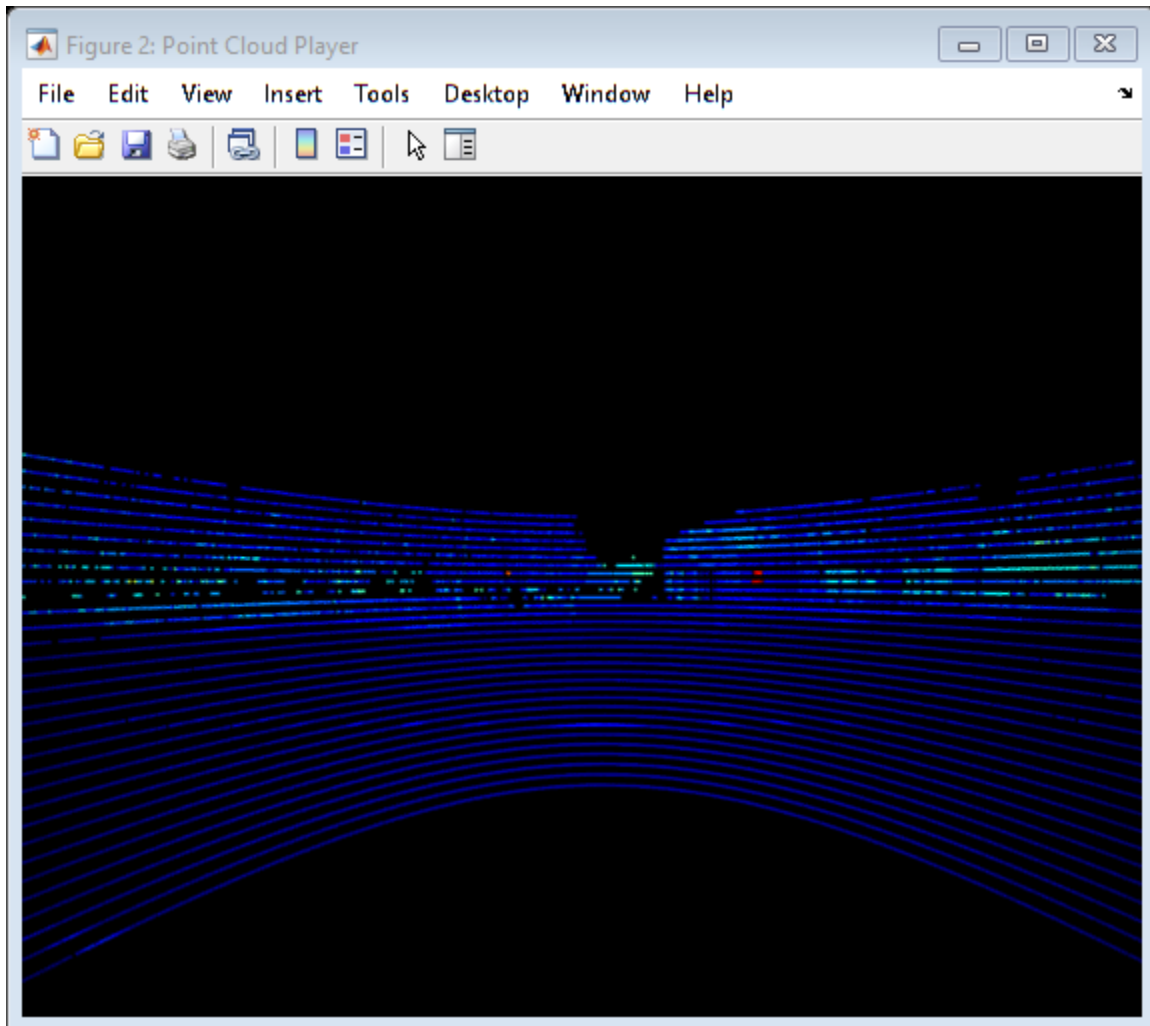
Use Recorded Lidar Data to Build a Map

Lidars are powerful sensors that can be used for perception in challenging environments where other sensors are not useful. They provide a detailed, full 360 degree view of the environment of the vehicle.

```
% Hide players  
hide(gpsPlayer)  
hide(lidarPlayer)
```

```
% Select a frame of lidar data to demonstrate registration workflow  
frameNum = 600;  
ptCloud = lidarPointClouds.PointCloud(frameNum);
```

```
% Display and rotate ego view to show lidar data  
helperVisualizeEgoView(ptCloud);
```

Lidars can be used to build centimeter-accurate HD maps, including HD maps of entire cities. These maps can later be used for in-vehicle localization. A typical approach to build such a map is to align successive lidar scans obtained from the moving vehicle and combine them into a single large point cloud. The rest of this example explores this approach to building a map.

- 1 **Align lidar scans:** Align successive lidar scans using a point cloud registration technique like the iterative closest point (ICP) algorithm or the normal-distributions transform (NDT) algorithm. See `pregistericp` and `pregisterndt` for more details about each algorithm. This example uses NDT, because it is typically more accurate, especially when considering rotations. The `pregisterndt` function returns the rigid transformation that aligns the moving point cloud with respect to the reference point cloud. By successively composing these transformations, each point cloud is transformed back to the reference frame of the first point cloud.
- 2 **Combine aligned scans:** Once a new point cloud scan is registered and transformed back to the reference frame of the first point cloud, the point cloud can be merged with the first point cloud using `pcmerge`.

Start by taking two point clouds corresponding to nearby lidar scans. To speed up processing, and accumulate enough motion between scans, use every tenth scan.

```
skipFrames = 10;
frameNum    = 100;

fixed  = lidarPointClouds.PointCloud(frameNum);
moving = lidarPointClouds.PointCloud(frameNum + skipFrames);
```

Prior to registration, process the point cloud so as to retain structures in the point cloud that are distinctive. These pre-processing steps include the following: * Detect and remove the ground plane * Detect and remove ego-vehicle

These steps are described in more detail in the “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox) example. In this example, the `helperProcessPointCloud` helper function accomplishes these steps.

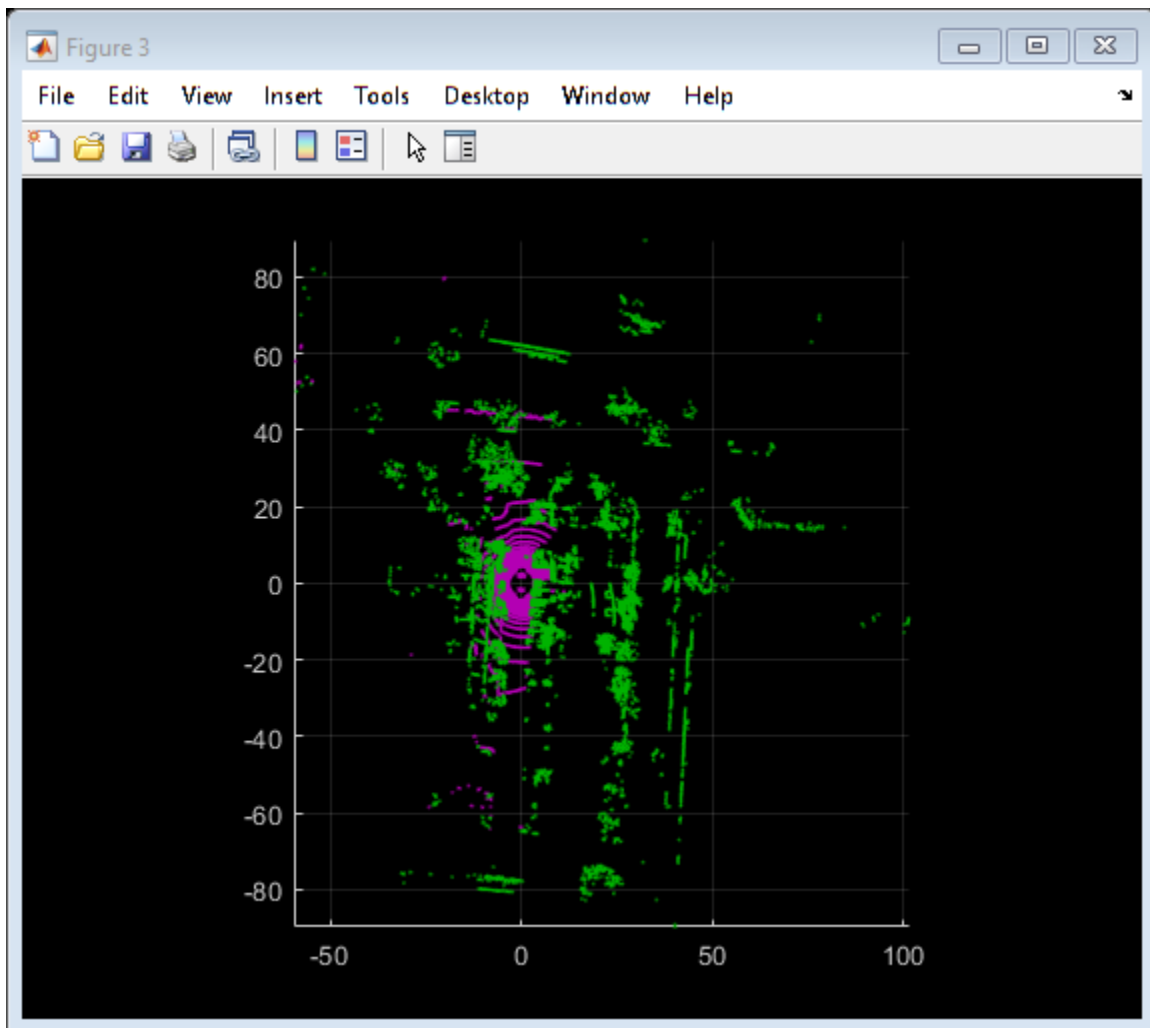
```
fixedProcessed  = helperProcessPointCloud(fixed);
movingProcessed = helperProcessPointCloud(moving);
```

Display the raw and processed point clouds in top-view. Magenta points were removed during processing. These points correspond to the ground plane and ego vehicle.

```
hFigFixed = figure;
pcshowpair(fixed, fixedProcessed)
view(2);                                     % Adjust view to show top-view

helperMakeFigurePublishFriendly(hFigFixed);

% Downsample the point clouds prior to registration. Downsampling improves
% both registration accuracy and algorithm speed.
downsamplePercent = 0.1;
fixedDownsampled  = pcdsample(fixedProcessed, 'random', downsamplePercent);
movingDownsampled = pcdsample(movingProcessed, 'random', downsamplePercent);
```



After preprocessing the point clouds, register them using NDT. Visualize the alignment before and after registration.

```
regGridStep = 5;
tform = pcregisterndt(movingDownsampled, fixedDownsampled, regGridStep);

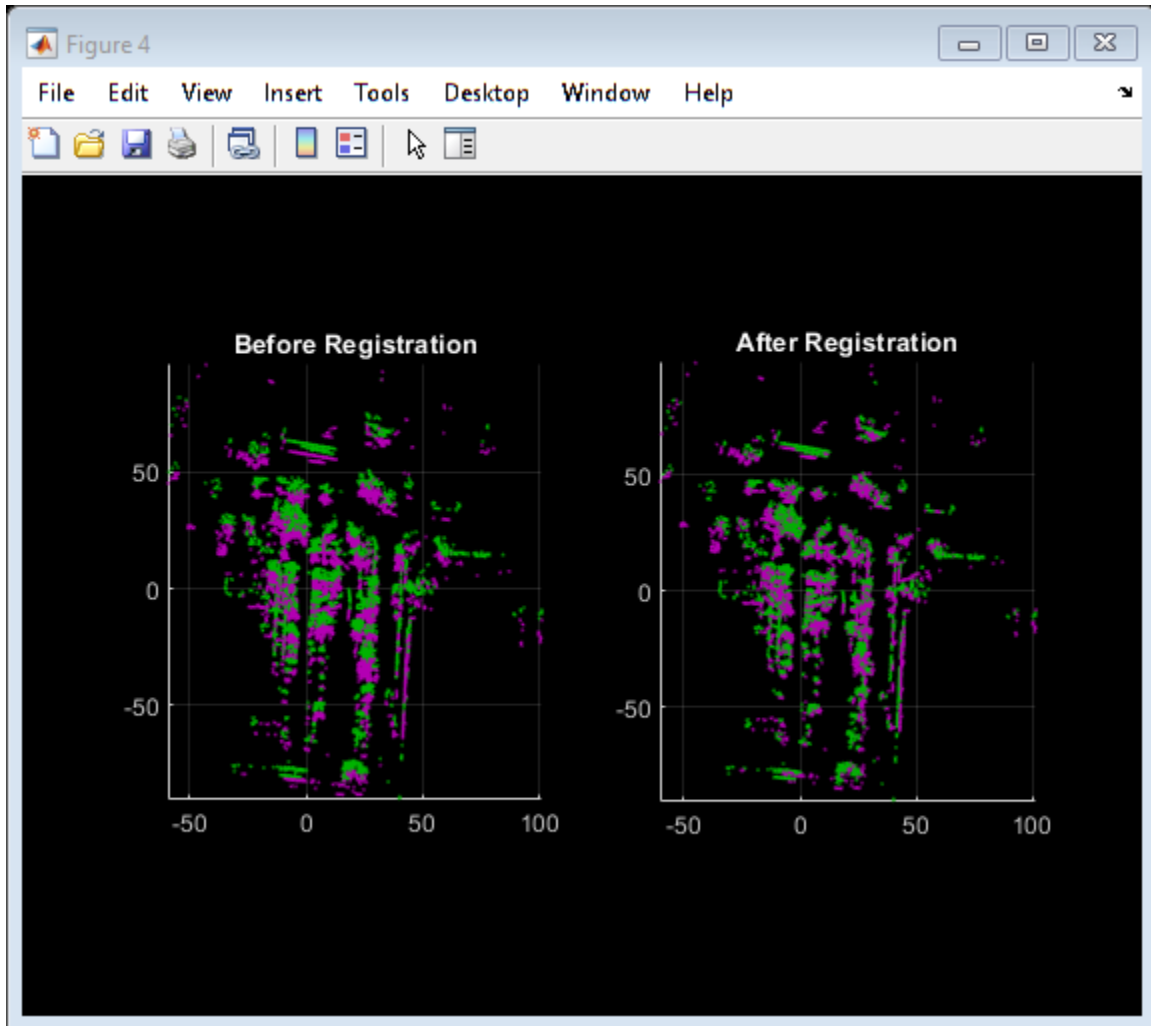
movingReg = pctransform(movingProcessed, tform);

% Visualize alignment in top-view before and after registration
hFigAlign = figure;

subplot(121)
pcshowpair(movingProcessed, fixedProcessed)
title('Before Registration')
view(2)

subplot(122)
pcshowpair(movingReg, fixedProcessed)
title('After Registration')
view(2)
```

```
helperMakeFigurePublishFriendly(hFigAlign);
```



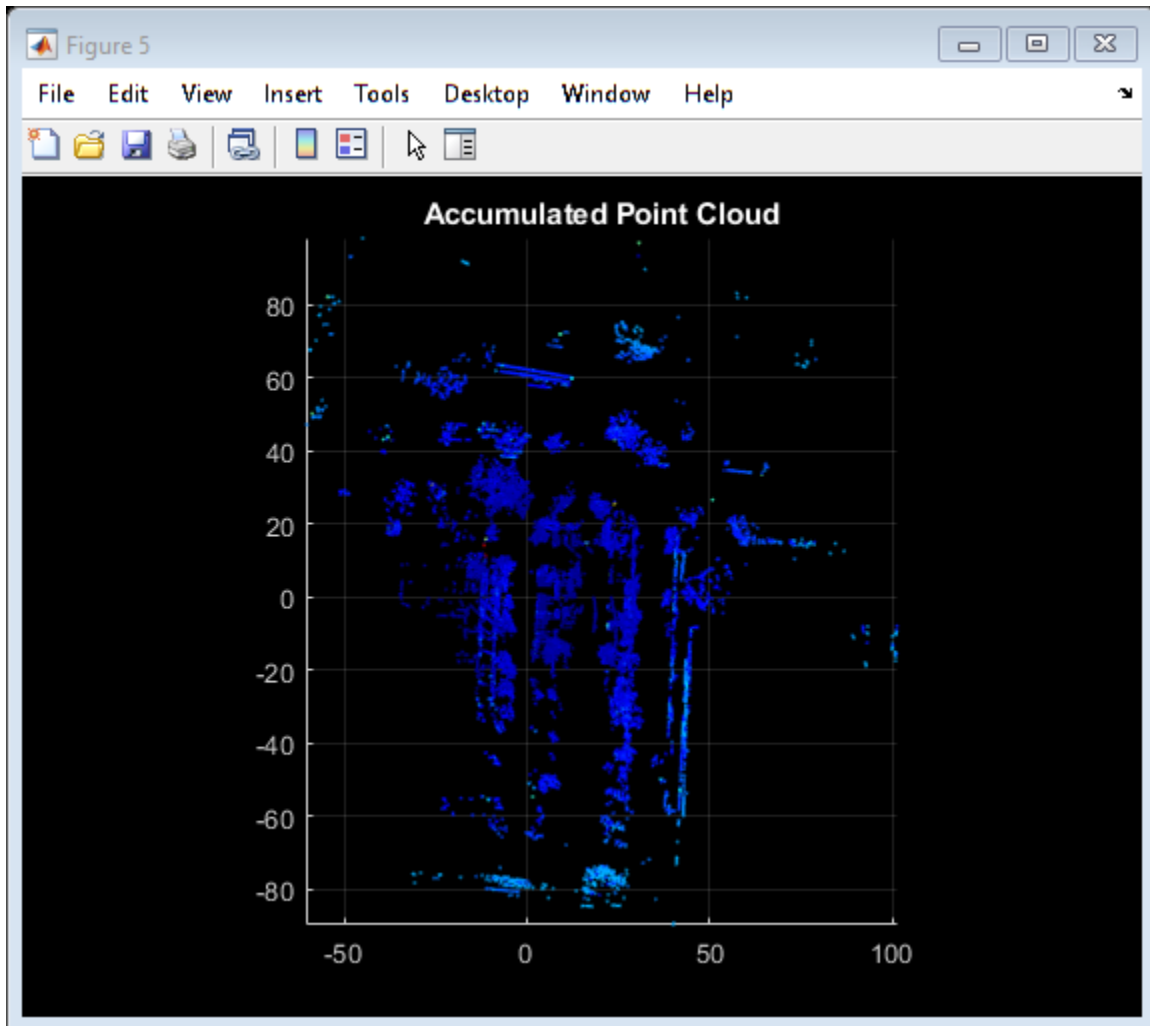
Notice that the point clouds are well-aligned after registration. Even though the point clouds are closely aligned, the alignment is still not perfect.

Next, merge the point clouds using `pcmerge`.

```
mergeGridStep = 0.5;
ptCloudAccum = pcmerge(fixedProcessed, movingReg, mergeGridStep);

hFigAccum = figure;
pcshow(ptCloudAccum)
title('Accumulated Point Cloud')
view(2)

helperMakeFigurePublishFriendly(hFigAccum);
```



Now that the processing pipeline for a single pair of point clouds is well-understood, put this together in a loop over the entire sequence of recorded data. The `helperLidarMapBuilder` class puts all this together. The `updateMap` method of the class takes in a new point cloud and goes through the steps detailed previously:

- Processing the point cloud by removing the ground plane and ego vehicle, using the `processPointCloud` method.
- Downsampling the point cloud.
- Estimating the rigid transformation required to merge the previous point cloud with the current point cloud.
- Transforming the point cloud back to the first frame.
- Merging the point cloud with the accumulated point cloud map.

Additionally, the `updateMap` method also accepts an initial transformation estimate, which is used to initialize the registration. A good initialization can significantly improve results of registration. Conversely, a poor initialization can adversely affect registration. Providing a good initialization can also improve the execution time of the algorithm.

A common approach to providing an initial estimate for registration is to use a constant velocity assumption. Use the transformation from the previous iteration as the initial estimate.

The `updateDisplay` method additionally creates and updates a 2-D top-view streaming point cloud display.

```
% Create a map builder object
mapBuilder = helperLidarMapBuilder('DownsamplePercent', downsamplePercent);

% Set random number seed
rng(0);

closeDisplay = false;
numFrames    = height(lidarPointClouds);

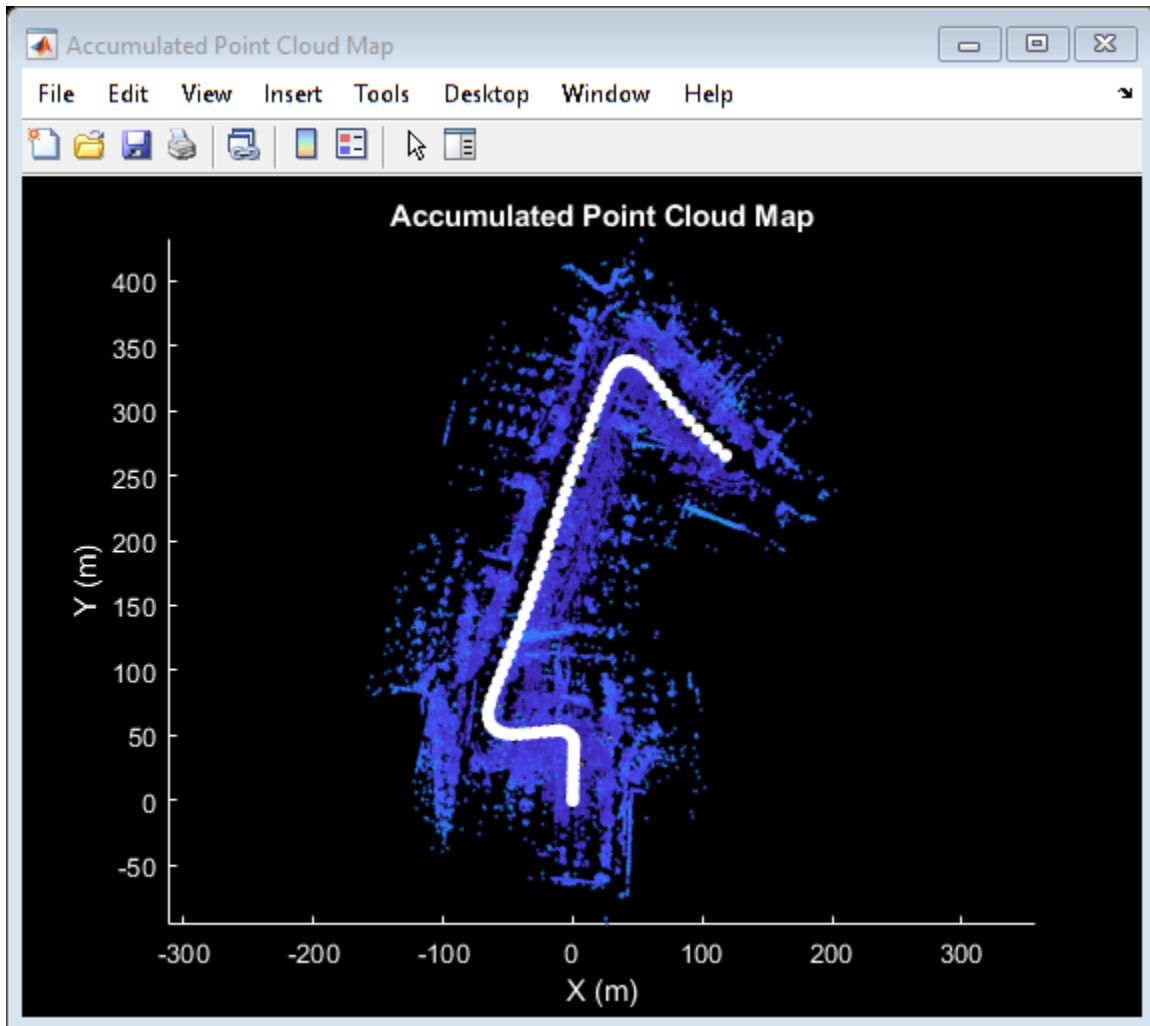
tform = rigid3d;
for n = 1 : skipFrames : numFrames - skipFrames

    % Get the nth point cloud
    ptCloud = lidarPointClouds.PointCloud(n);

    % Use transformation from previous iteration as initial estimate for
    % current iteration of point cloud registration. (constant velocity)
    initTform = tform;

    % Update map using the point cloud
    tform = updateMap(mapBuilder, ptCloud, initTform);

    % Update map display
    updateDisplay(mapBuilder, closeDisplay);
end
```



Point cloud registration alone builds a map of the environment traversed by the vehicle. While the map may appear locally consistent, it might have developed significant drift over the entire sequence.

Use the recorded GPS readings as a ground truth trajectory, to visually evaluate the quality of the built map. First convert the GPS readings (latitude, longitude, altitude) to a local coordinate system. Select a local coordinate system that coincides with the origin of the first point cloud in the sequence. This conversion is computed using two transformations:

- 1 Convert the GPS coordinates to local Cartesian East-North-Up coordinates using the `latlon2local` (Automated Driving Toolbox) function. The GPS location from the start of the trajectory is used as the reference point and defines the origin of the local x,y,z coordinate system.
- 2 Rotate the Cartesian coordinates so that the local coordinate system is aligned with the first lidar sensor coordinates. Since the exact mounting configuration of the lidar and GPS on the vehicle are not known, they are estimated.

```
% Select reference point as first GPS reading
origin = [gpsSequence.Latitude(1), gpsSequence.Longitude(1), gpsSequence.Altitude(1)];
```

```
% Convert GPS readings to a local East-North-Up coordinate system
```

```
[xEast, yNorth, zUp] = latlon2local(gpsSequence.Latitude, gpsSequence.Longitude, ...
    gpsSequence.Altitude, origin);

% Estimate rough orientation at start of trajectory to align local ENU
% system with lidar coordinate system
theta = median(atan2d(yNorth(1:15), xEast(1:15)));

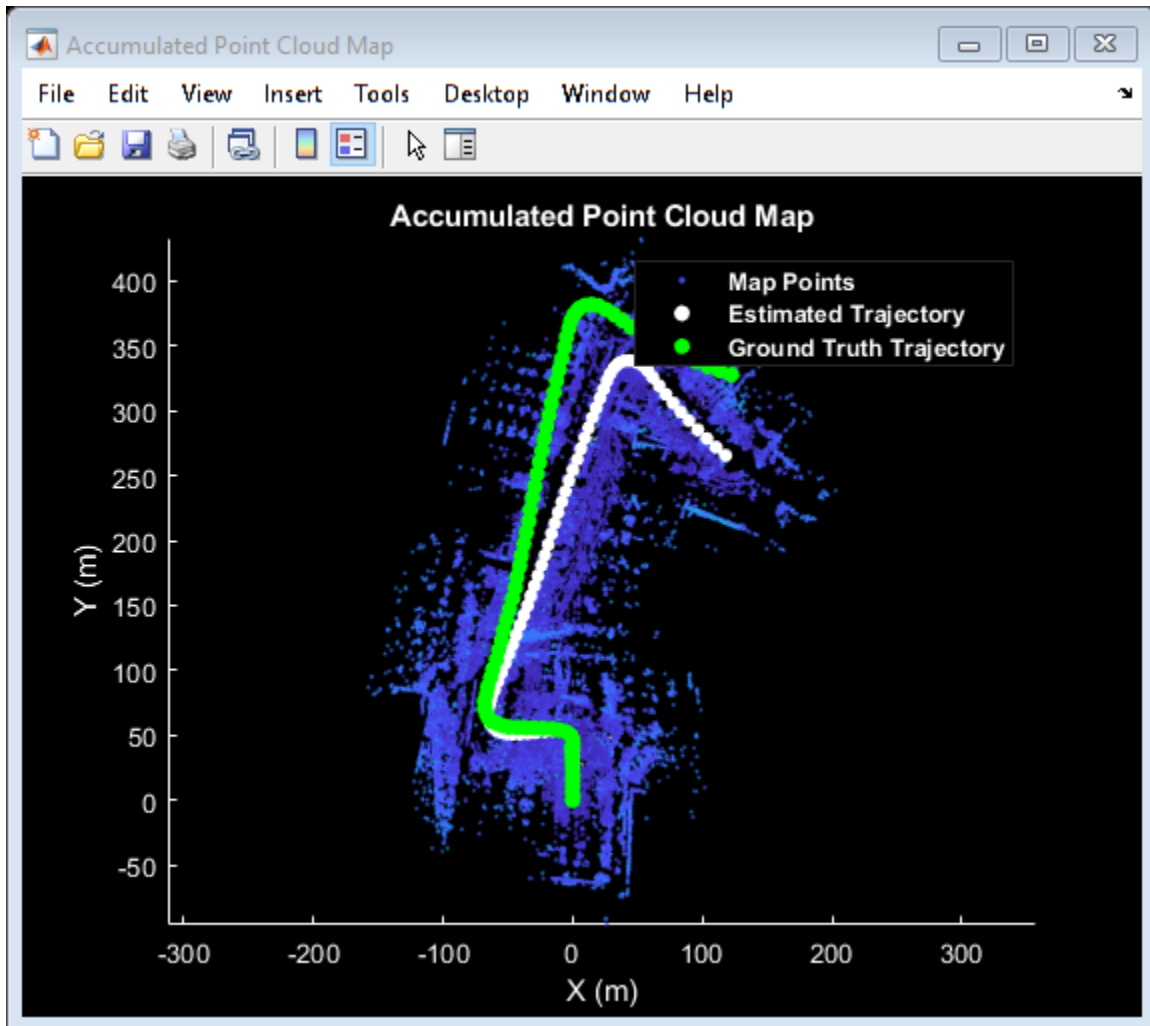
R = [ cosd(90-theta) sind(90-theta) 0;
      -sind(90-theta) cosd(90-theta) 0;
      0 0 1];

% Rotate ENU coordinates to align with lidar coordinate system
groundTruthTrajectory = [xEast, yNorth, zUp] * R;

Superimpose the ground truth trajectory on the built map.

hold(mapBuilder.Axes, 'on')
scatter(mapBuilder.Axes, groundTruthTrajectory(:,1), groundTruthTrajectory(:,2), ...
    'green','filled');

helperAddLegend(mapBuilder.Axes, ...
    {'Map Points', 'Estimated Trajectory', 'Ground Truth Trajectory'});
```

After the initial turn, the estimated trajectory veers off the ground truth trajectory significantly. The trajectory estimated using point cloud registration alone can drift for a number of reasons:

- Noisy scans from the sensor without sufficient overlap
- Absence of strong enough features, for example, near long roads
- Inaccurate initial transformation, especially when rotation is significant.

```
% Close map display
updateDisplay(mapBuilder, true);
```

Use IMU Orientation to Improve Built Map

An IMU is an electronic device mounted on a platform. IMUs contain multiple sensors that report various information about the motion of the vehicle. Typical IMUs incorporate accelerometers, gyroscopes, and magnetometers. An IMU can provide a reliable measure of orientation.

Use the IMU readings to provide a better initial estimate for registration. The IMU-reported sensor readings used in this example have already been filtered on the device.

```
% Reset the map builder to clear previously built map
reset(mapBuilder);
```

```
% Set random number seed
rng(0);

initTform = rigid3d;
for n = 1 : skipFrames : numFrames - skipFrames

    % Get the nth point cloud
    ptCloud = lidarPointClouds.PointCloud(n);

    if n > 1
        % Since IMU sensor reports readings at a much faster rate, gather
        % IMU readings reported since the last lidar scan.
        prevTime = lidarPointClouds.Time(n - skipFrames);
        currTime = lidarPointClouds.Time(n);
        timeSinceScan = timerange(prevTime, currTime);

        imuReadings = imuOrientations(timeSinceScan, 'Orientation');

        % Form an initial estimate using IMU readings
        initTform = helperComputeInitialEstimateFromIMU(imuReadings, tform);
    end

    % Update map using the point cloud
    tform = updateMap(mapBuilder, ptCloud, initTform);

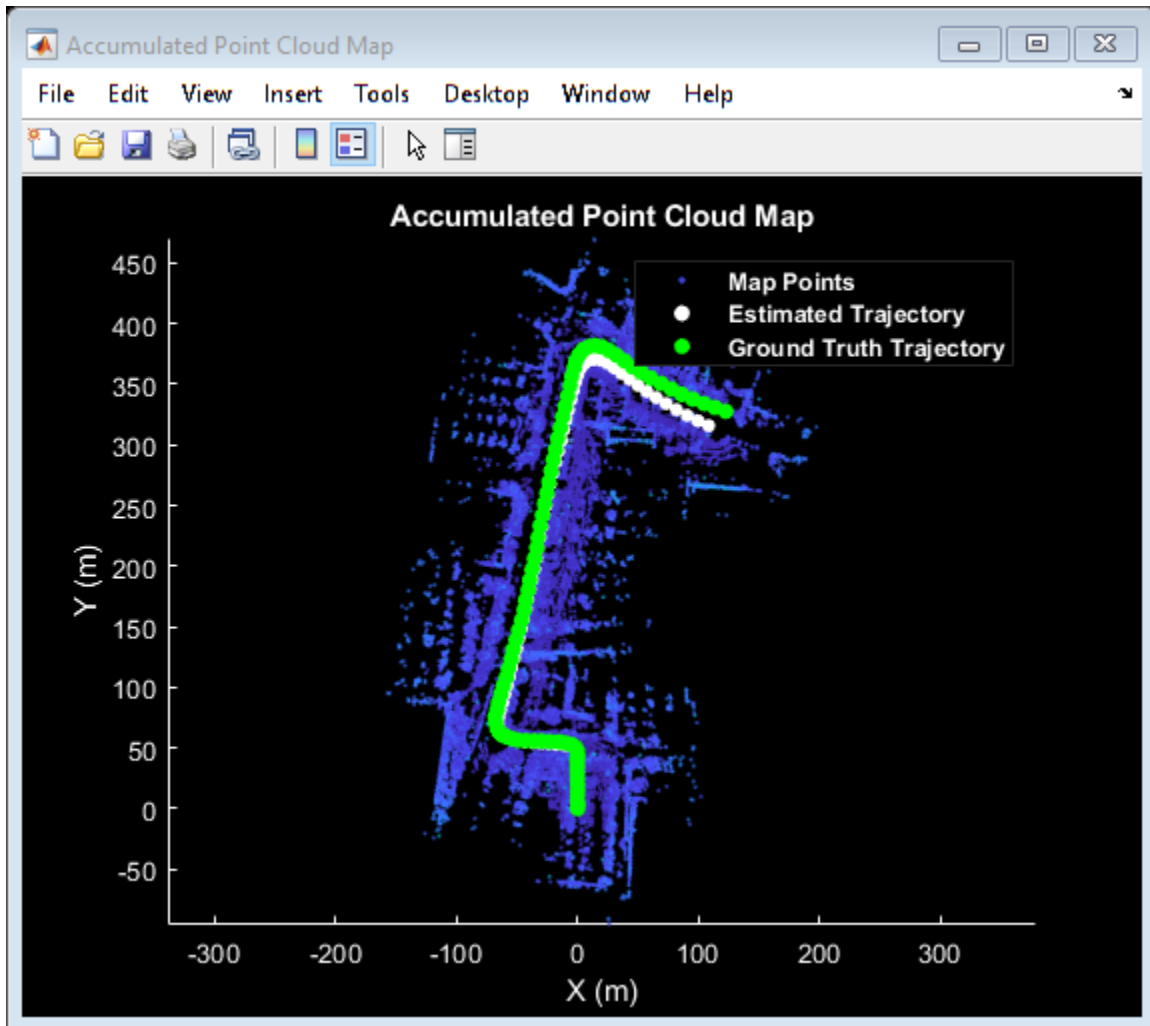
    % Update map display
    updateDisplay(mapBuilder, closeDisplay);
end

% Superimpose ground truth trajectory on new map
hold(mapBuilder.Axes, 'on')
scatter(mapBuilder.Axes, groundTruthTrajectory(:,1), groundTruthTrajectory(:,2), ...
        'green','filled');

helperAddLegend(mapBuilder.Axes, ...
    {'Map Points', 'Estimated Trajectory', 'Ground Truth Trajectory'});

% Capture snapshot for publishing
snapnow;

% Close open figures
close([hFigFixed, hFigAlign, hFigAccum]);
updateDisplay(mapBuilder, true);
```



Using the orientation estimate from IMU significantly improved registration, leading to a much closer trajectory with smaller drift.

Supporting Functions

helperAlignPlayers aligns a cell array of streaming players so they are arranged from left to right on the screen.

```
function helperAlignPlayers(players)
    validateattributes(players, {'cell'}, {'vector'});

    hasAxes = cellfun(@(p)isprop(p,'Axes'),players,'UniformOutput', true);
    if ~all(hasAxes)
        error('Expected all viewers to have an Axes property');
    end

    screenSize = get(groot, 'ScreenSize');
    screenMargin = [50, 100];

    playerSizes = cellfun(@getPlayerSize, players, 'UniformOutput', false);
```

```

playerSizes = cell2mat(playerSizes);
maxHeightInSet = max(playerSizes(1:3:end));

% Arrange players vertically so that the tallest player is 100 pixels from
% the top.
location = round([screenMargin(1), screenSize(4)-screenMargin(2)-maxHeightInSet]);
for n = 1 : numel(players)
    player = players{n};

    hFig = ancestor(player.Axes, 'figure');
    hFig.OuterPosition(1:2) = location;

    % Set up next location by going right
    location = location + [50+hFig.OuterPosition(3), 0];
end

function sz = getPlayerSize(viewer)

    % Get the parent figure container
    h = ancestor(viewer.Axes, 'figure');

    sz = h.OuterPosition(3:4);
end
end

```

helperVisualizeEgoView visualizes point cloud data in the ego perspective by rotating about the center.

```

function player = helperVisualizeEgoView(ptCloud)

% Create a pcplayer object
xlimits = ptCloud.XLimits;
ylimits = ptCloud.YLimits;
zlimits = ptCloud.ZLimits;

player = pcplayer(xlimits, ylimits, zlimits);

% Turn off axes lines
axis(player.Axes, 'off');

% Set up camera to show ego view
camproj(player.Axes, 'perspective');
camva(player.Axes, 90);
campos(player.Axes, [0 0 0]);
camtarget(player.Axes, [-1 0 0]);

% Set up a transformation to rotate by 5 degrees
theta = 5;
R = [ cosd(theta) sind(theta) 0 0
      -sind(theta) cosd(theta) 0 0
        0          0          1 0
        0          0          0 1];
rotateByTheta = rigid3d(R);

for n = 0 : theta : 359
    % Rotate point cloud by theta
    ptCloud = pctransform(ptCloud, rotateByTheta);
end

```

```

    % Display point cloud
    view(player, ptCloud);

    pause(0.05)
end
end

helperProcessPointCloud processes a point cloud by removing points belonging to the ground
plane or ego vehicle.

function ptCloudProcessed = helperProcessPointCloud(ptCloud)

% Check if the point cloud is organized
isOrganized = ~ismatrix(ptCloud.Location);

% If the point cloud is organized, use range-based flood fill algorithm
% (segmentGroundFromLidarData). Otherwise, use plane fitting.
groundSegmentationMethods = ["planefit", "rangefloodfill"];
method = groundSegmentationMethods(isOrganized+1);

if method == "planefit"
    % Segment ground as the dominant plane, with reference normal vector
    % pointing in positive z-direction, using pcfiteplane. For organized
    % point clouds, consider using segmentGroundFromLidarData instead.
    maxDistance = 0.4; % meters
    maxAngDistance = 5; % degrees
    refVector = [0, 0, 1]; % z-direction

    [~,groundIndices] = pcfiteplane(ptCloud, maxDistance, refVector, maxAngDistance);
elseif method == "rangefloodfill"
    % Segment ground using range-based flood fill.
    groundIndices = segmentGroundFromLidarData(ptCloud);
else
    error("Expected method to be 'planefit' or 'rangefloodfill'")
end

% Segment ego vehicle as points within a given radius of sensor
sensorLocation = [0, 0, 0];
radius = 3.5;

egoIndices = findNeighborsInRadius(ptCloud, sensorLocation, radius);

% Remove points belonging to ground or ego vehicle
ptsToKeep = true(ptCloud.Count, 1);
ptsToKeep(groundIndices) = false;
ptsToKeep(egoIndices) = false;

% If the point cloud is organized, retain organized structure
if isOrganized
    ptCloudProcessed = select(ptCloud, find(ptsToKeep), 'OutputSize', 'full');
else
    ptCloudProcessed = select(ptCloud, find(ptsToKeep));
end
end

```

helperComputeInitialEstimateFromIMU estimates an initial transformation for NDT using IMU orientation readings and previously estimated transformation.

```

function tform = helperComputeInitialEstimateFromIMU(imuReadings, prevTform)

% Initialize transformation using previously estimated transform
tform = prevTform;

% If no IMU readings are available, return
if height(imuReadings) <= 1
    return;
end

% IMU orientation readings are reported as quaternions representing the
% rotational offset to the body frame. Compute the orientation change
% between the first and last reported IMU orientations during the interval
% of the lidar scan.
q1 = imuReadings.Orientation(1);
q2 = imuReadings.Orientation(end);

% Compute rotational offset between first and last IMU reading by
% - Rotating from q2 frame to body frame
% - Rotating from body frame to q1 frame
q = q1 * conj(q2);

% Convert to Euler angles
yawPitchRoll = euler(q, 'ZYX', 'point');

% Discard pitch and roll angle estimates. Use only heading angle estimate
% from IMU orientation.
yawPitchRoll(2:3) = 0;

% Convert back to rotation matrix
q = quaternion(yawPitchRoll, 'euler', 'ZYX', 'point');
R = rotmat(q, 'point');

% Use computed rotation
tform.T(1:3, 1:3) = R;
end

```

helperAddLegend adds a legend to the axes.

```

function helperAddLegend(hAx, labels)

% Add a legend to the axes
hLegend = legend(hAx, labels{:});

% Set text color and font weight
hLegend.TextColor = [1 1 1];
hLegend.FontWeight = 'bold';
end

```

helperMakeFigurePublishFriendly adjusts figures so that screenshot captured by publish is correct.

```

function helperMakeFigurePublishFriendly(hFig)

if ~isempty(hFig) && isvalid(hFig)
    hFig.HandleVisibility = 'callback';
end

```

end

See Also

Functions

`pcmerge` | `pcregistericp` | `pcregisterndt`

Objects

`geoplayer` | `pcplayer` | `pointCloud`

More About

- “Build a Map from Lidar Data Using SLAM” (Automated Driving Toolbox)
- “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox)

External Websites

- Udacity Self-Driving Car Data Subset (MathWorks GitHub repository)

Build a Map from Lidar Data Using SLAM

This example shows how to process 3-D lidar data from a sensor mounted on a vehicle to progressively build a map and estimate the trajectory of a vehicle using simultaneous localization and mapping (SLAM). In addition to 3-D lidar data, an inertial navigation sensor (INS) is also used to help build the map. Maps built this way can facilitate path planning for vehicle navigation or can be used for localization.

Overview

The “Build a Map from Lidar Data” (Automated Driving Toolbox) example uses 3-D lidar data and IMU readings to progressively build a map of the environment traversed by a vehicle. While this approach builds a locally consistent map, it is suitable only for mapping small areas. Over longer sequences, the drift accumulates into a significant error. To overcome this limitation, this example recognizes previously visited places and tries to correct for the accumulated drift using the graph SLAM approach.

Load and Explore Recorded Data

The data used in this example is part of the Velodyne SLAM Dataset, and represents close to 6 minutes of recorded data. Download the data to a temporary directory.

Note: This download can take a few minutes.

```
baseDownloadURL = 'https://www.mrt.kit.edu/z/publ/download/velodyneslam/data/scenario1.zip';
dataFolder      = fullfile(tempdir, 'kit_velodyneslam_data_scenario1', filesep);
options         = weboptions('Timeout', Inf);

zipFileName    = dataFolder + "scenario1.zip";

folderExists = exist(dataFolder, 'dir');
if ~folderExists
    % Create a folder in a temporary directory to save the downloaded zip
    % file.
    mkdir(dataFolder);

    disp('Downloading scenario1.zip (153 MB) ...')
    websave(zipFileName, baseDownloadURL, options);

    % Unzip downloaded file
    unzip(zipFileName, dataFolder);
end
```

```
Downloading scenario1.zip (153 MB) ...
```

Use the `helperReadDataset` function to read data from the created folder in the form of a `timetable`. The point clouds captured by the lidar are stored in the form of PNG image files. Extract the list of point cloud file names in the `pointCloudTable` variable. To read the point cloud data from the image file, use the `helperReadPointCloudFromFile` function. This function takes an image file name and returns a `pointCloud` object. The INS readings are read directly from a configuration file and stored in the `insDataTable` variable.

```
datasetTable = helperReadDataset(dataFolder);

pointCloudTable = datasetTable(:, 1);
insDataTable     = datasetTable(:, 2:end);
```


Read the first point cloud and display it at the MATLAB® command prompt

```
ptCloud = helperReadPointCloudFromFile(pointCloudTable.PointCloudFileName{1});
disp(ptCloud)
```

pointCloud with properties:

```
Location: [64x870x3 single]
Count: 55680
XLimits: [-78.4980 77.7062]
YLimits: [-76.8795 74.7502]
ZLimits: [-2.4839 2.6836]
Color: []
Normal: []
Intensity: []
```

Display the first INS reading. The timetable holds Heading, Pitch, Roll, X, Y, and Z information from the INS.

```
disp(insDataTable(1, :))
```

Timestamps	Heading	Pitch	Roll	X	Y	Z
13-Jun-2010 06:27:31	1.9154	0.007438	-0.019888	-2889.1	-2183.9	116.47

Visualize the point clouds using `pcplayer`, a streaming point cloud display. The vehicle traverses a path consisting of two loops. In the first loop, the vehicle makes a series of turns and returns to the starting point. In the second loop, the vehicle makes a series of turns along another route and again returns to the starting point.

```
% Specify limits of the player
xlimits = [-45 45]; % meters
ylimits = [-45 45];
zlimits = [-10 20];

% Create a streaming point cloud display object
lidarPlayer = pcplayer(xlimits, ylimits, zlimits);

% Customize player axes labels
xlabel(lidarPlayer.Axes, 'X (m)')
ylabel(lidarPlayer.Axes, 'Y (m)')
zlabel(lidarPlayer.Axes, 'Z (m)')

title(lidarPlayer.Axes, 'Lidar Sensor Data')

% Skip every other frame since this is a long sequence
skipFrames = 2;
numFrames = height(pointCloudTable);
for n = 1 : skipFrames : numFrames

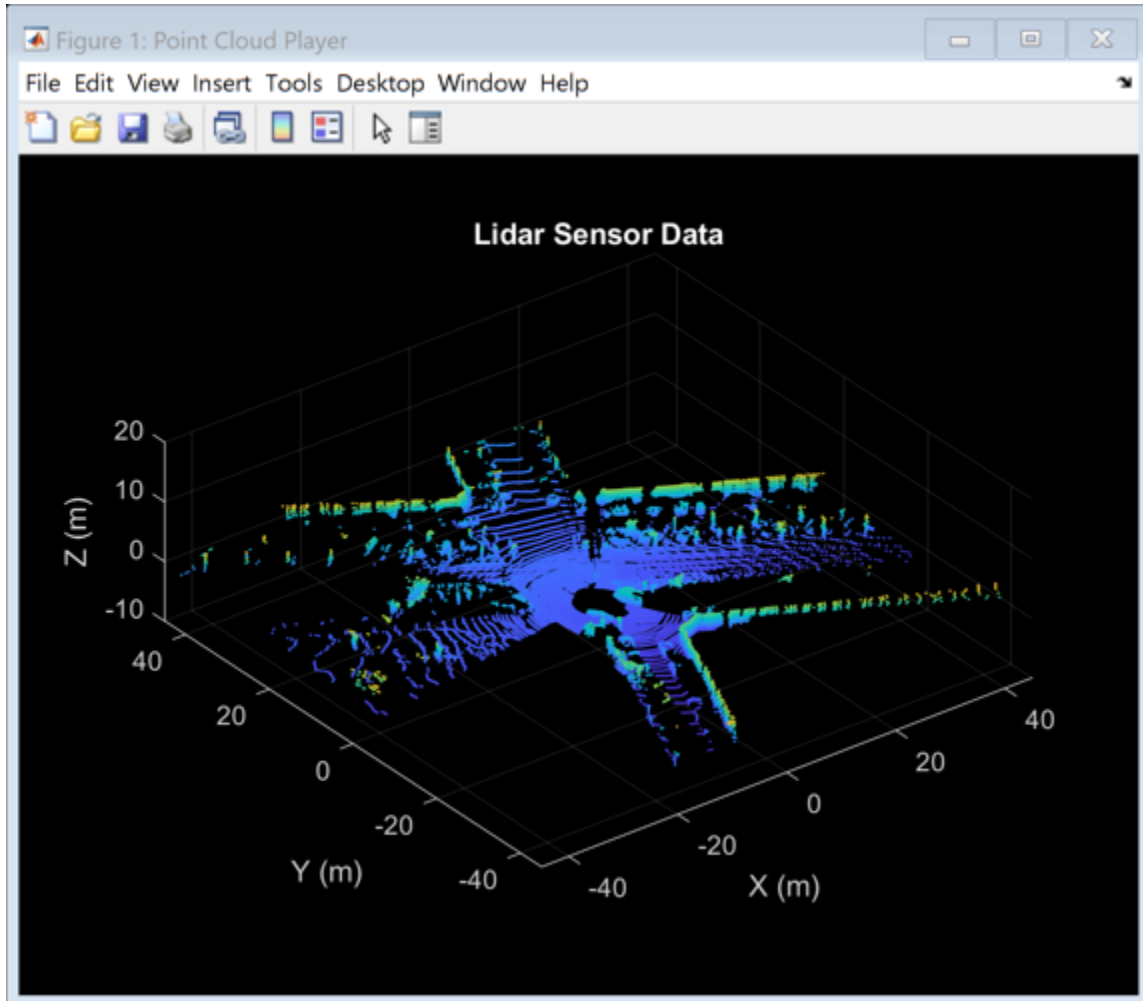
    % Read a point cloud
    fileName = pointCloudTable.PointCloudFileName{n};
    ptCloud = helperReadPointCloudFromFile(fileName);

    % Visualize point cloud
```

```

view(lidarPlayer, ptCloud);
pause(0.01)
end

```



Build a Map Using Odometry

First, use the approach explained in the “Build a Map from Lidar Data” (Automated Driving Toolbox) example to build a map. The approach consists of the following steps:

- **Align lidar scans:** Align successive lidar scans using a point cloud registration technique. This example uses `pcregisterndt` for registering scans. By successively composing these transformations, each point cloud is transformed back to the reference frame of the first point cloud.
- **Combine aligned scans:** Generate a map by combining all the transformed point clouds.

This approach of incrementally building a map and estimating the trajectory of the vehicle is called *odometry*.

Use a `pcviewset` object to store and manage data across multiple views. A view set consists of a set of connected views.

- Each view stores information associated with a single view. This information includes the absolute pose of the view, the point cloud sensor data captured at that view, and a unique identifier for the view. Add views to the view set using `addView`.
- To establish a connection between views use `addConnection`. A connection stores information like the relative transformation between the connecting views, the uncertainty involved in computing this measurement (represented as an information matrix) and the associated view identifiers.
- Use the `plot` method to visualize the connections established by the view set. These connections can be used to visualize the path traversed by the vehicle.

```
hide(lidarPlayer)

% Set random seed to ensure reproducibility
rng(0);

% Create an empty view set
vSet = pcviewset;

% Create a figure for view set display
hFigBefore = figure('Name', 'View Set Display');
hAxBefore = axes(hFigBefore);

% Initialize point cloud processing parameters
downsamplePercent = 0.1;
regGridSize       = 3;

% Initialize transformations
absTform = rigid3d; % Absolute transformation to reference frame
relTform = rigid3d; % Relative transformation between successive scans

viewId = 1;
skipFrames = 5;
numFrames = height(pointCloudTable);
displayRate = 100; % Update display every 100 frames
for n = 1 : skipFrames : numFrames

    % Read point cloud
    fileName = pointCloudTable.PointCloudFileName{n};
    ptCloudOrig = helperReadPointCloudFromFile(fileName);

    % Process point cloud
    % - Segment and remove ground plane
    % - Segment and remove ego vehicle
    ptCloud = helperProcessPointCloud(ptCloudOrig);

    % Downsample the processed point cloud
    ptCloud = pcdownsample(ptCloud, "random", downsamplePercent);

    firstFrame = (n==1);
    if firstFrame
        % Add first point cloud scan as a view to the view set
        vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

        viewId = viewId + 1;
        ptCloudPrev = ptCloud;
        continue;
    end
end
```

```
end

% Use INS to estimate an initial transformation for registration
initTform = helperComputeInitialEstimateFromINS(relTform, ...
    insDataTable(n-skipFrames:n, :));

% Compute rigid transformation that registers current point cloud with
% previous point cloud
relTform = pcregisterndt(ptCloud, ptCloudPrev, regGridSize, ...
    "InitialTransform", initTform);

% Update absolute transformation to reference frame (first point cloud)
absTform = rigid3d( relTform.T * absTform.T );

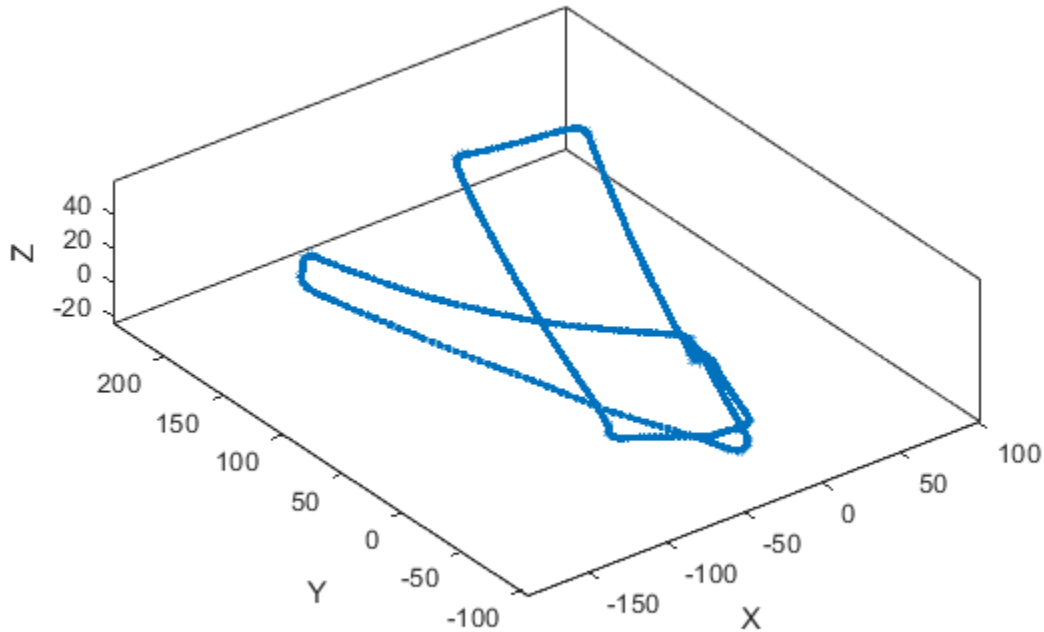
% Add current point cloud scan as a view to the view set
vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

% Add a connection from the previous view to the current view, representing
% the relative transformation between them
vSet = addConnection(vSet, viewId-1, viewId, relTform);

viewId = viewId + 1;

ptCloudPrev = ptCloud;
initTform = relTform;

if n>1 && mod(n, displayRate) == 1
    plot(vSet, "Parent", hAxBefore);
    drawnow update
end
end
```



The view set object `vSet`, now holds views and connections. In the Views table of `vSet`, the `AbsolutePose` variable specifies the absolute pose of each view with respect to the first view. In the Connections table of `vSet`, the `RelativePose` variable specifies relative constraints between the connected views, the `InformationMatrix` variable specifies, for each edge, the uncertainty associated with a connection.

```
% Display the first few views and connections
head(vSet.Views)
head(vSet.Connections)
```

ans =

8×3 table

ViewId	AbsolutePose	PointCloud
1	[1×1 rigid3d]	[1×1 pointCloud]
2	[1×1 rigid3d]	[1×1 pointCloud]
3	[1×1 rigid3d]	[1×1 pointCloud]
4	[1×1 rigid3d]	[1×1 pointCloud]
5	[1×1 rigid3d]	[1×1 pointCloud]
6	[1×1 rigid3d]	[1×1 pointCloud]
7	[1×1 rigid3d]	[1×1 pointCloud]
8	[1×1 rigid3d]	[1×1 pointCloud]

```
ans =
```

```
8×4 table
```

ViewId1	ViewId2	RelativePose	InformationMatrix
1	2	[1×1 rigid3d]	{6×6 double}
2	3	[1×1 rigid3d]	{6×6 double}
3	4	[1×1 rigid3d]	{6×6 double}
4	5	[1×1 rigid3d]	{6×6 double}
5	6	[1×1 rigid3d]	{6×6 double}
6	7	[1×1 rigid3d]	{6×6 double}
7	8	[1×1 rigid3d]	{6×6 double}
8	9	[1×1 rigid3d]	{6×6 double}

Now, build a point cloud map using the created view set. Align the view absolute poses with the point clouds in the view set using `pcalign`. Specify a grid size to control the resolution of the map. The map is returned as a `pointCloud` object.

```
ptClouds = vSet.Views.PointCloud;
absPoses = vSet.Views.AbsolutePose;
mapGridSize = 0.2;
ptCloudMap = pcalign(ptClouds, absPoses, mapGridSize);
```

Notice that the path traversed using this approach drifts over time. While the path along the first loop back to the starting point seems reasonable, the second loop drifts significantly from the starting point. The accumulated drift results in the second loop terminating several meters away from the starting point.

A map built using odometry alone is inaccurate. Display the built point cloud map with the traversed path. Notice that the map and traversed path for the second loop are not consistent with the first loop.

```
hold(hAxBefore, 'on');
pcshow(ptCloudMap);
hold(hAxBefore, 'off');

close(hAxBefore.Parent)
```

Correct Drift Using Pose Graph Optimization

Graph SLAM is a widely used technique for resolving the drift in odometry. The graph SLAM approach incrementally creates a graph, where nodes correspond to vehicle poses and edges represent sensor measurements constraining the connected poses. Such a graph is called a *pose graph*. The pose graph contains edges that encode contradictory information, due to noise or inaccuracies in measurement. The nodes in the constructed graph are then optimized to find the set of vehicle poses that optimally explain the measurements. This technique is called *pose graph optimization*.

To create a pose graph from a view set, you can use the `createPoseGraph` function. This function creates a node for each view, and an edge for each connection in the view set. To optimize the pose graph, you can use the `optimizePoseGraph` (Navigation Toolbox) function.

A key aspect contributing to the effectiveness of graph SLAM in correcting drift is the accurate detection of loops, that is, places that have been previously visited. This is called *loop closure detection* or *place recognition*. Adding edges to the pose graph corresponding to loop closures provides a contradictory measurement for the connected node poses, which can be resolved during pose graph optimization.

Loop closures can be detected using descriptors that characterize the local environment visible to the Lidar sensor. The *Scan Context* descriptor [1] is one such descriptor that can be computed from a point cloud using the `scanContextDescriptor` function. This example uses the `helperLoopClosureDetector` class to detect loop closures. This class uses scan context descriptors for describing individual point clouds, and a two-phase descriptor search algorithm.

- **Descriptor Computation:** A scan context descriptor and a ring key are extracted from each point cloud using the `scanContextDescriptor` function.
- **Descriptor Matching:** In the first phase, the ring key is used to find potential loop candidates. In the second phase, a nearest neighbor search is performed using the scan context feature descriptors among the potential loop candidates. The `scanContextDistance` function is used to compute the distance between two scan context descriptors. See the `helperFeatureSearcher` class for more details.

```
% Set random seed to ensure reproducibility
rng(0);

% Create an empty view set
vSet = pcviewset;

% Create a loop closure detector
matchThresh = 0.08;
loopDetector = helperLoopClosureDetector('MatchThreshold', matchThresh);

% Create a figure for view set display
hFigBefore = figure('Name', 'View Set Display');
hAxBefore = axes(hFigBefore);

% Initialize transformations
absTform = rigid3d; % Absolute transformation to reference frame
relTform = rigid3d; % Relative transformation between successive scans

maxTolerableRMSE = 3; % Maximum allowed RMSE for a loop closure candidate to be accepted

viewId = 1;
for n = 1 : skipFrames : numFrames

    % Read point cloud
    fileName = pointCloudTable.PointCloudFileName{n};
    ptCloudOrig = helperReadPointCloudFromFile(fileName);

    % Process point cloud
    % - Segment and remove ground plane
    % - Segment and remove ego vehicle
    ptCloud = helperProcessPointCloud(ptCloudOrig);

    % Downsample the processed point cloud
    ptCloud = pcdownsampling(ptCloud, "random", downsamplingPercent);

    firstFrame = (n==1);
```

```

if firstFrame
    % Add first point cloud scan as a view to the view set
    vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

    viewId = viewId + 1;
    ptCloudPrev = ptCloud;
    continue;
end

% Use INS to estimate an initial transformation for registration
initTform = helperComputeInitialEstimateFromINS(relTform, ...
    insDataTable(n-skipFrames:n, :));

% Compute rigid transformation that registers current point cloud with
% previous point cloud
relTform = pcregisterndt(ptCloud, ptCloudPrev, regGridSize, ...
    "InitialTransform", initTform);

% Update absolute transformation to reference frame (first point cloud)
absTform = rigid3d( relTform.T * absTform.T );

% Add current point cloud scan as a view to the view set
vSet = addView(vSet, viewId, absTform, "PointCloud", ptCloudOrig);

% Add a connection from the previous view to the current view representing
% the relative transformation between them
vSet = addConnection(vSet, viewId-1, viewId, relTform);

% Detect loop closure candidates
[loopFound, loopViewId] = detectLoop(loopDetector, ptCloudOrig);

% A loop candidate was found
if loopFound
    loopViewId = loopViewId(1);

    % Retrieve point cloud from view set
    ptCloudOrig = vSet.Views.PointCloud( find(vSet.Views.ViewId == loopViewId, 1) );

    % Process point cloud
    ptCloudOld = helperProcessPointCloud(ptCloudOrig);

    % Downsample point cloud
    ptCloudOld = pcdsample(ptCloudOld, "random", downsamplePercent);

    % Use registration to estimate the relative pose
    [relTform, ~, rmse] = pcregisterndt(ptCloud, ptCloudOld, ...
        regGridSize, "MaxIterations", 50);

    acceptLoopClosure = rmse <= maxTolerableRMSE;
    if acceptLoopClosure
        % For simplicity, use a constant, small information matrix for
        % loop closure edges
        infoMat = 0.01 * eye(6);

        % Add a connection corresponding to a loop closure
        vSet = addConnection(vSet, loopViewId, viewId, relTform, infoMat);
    end
end
end

```



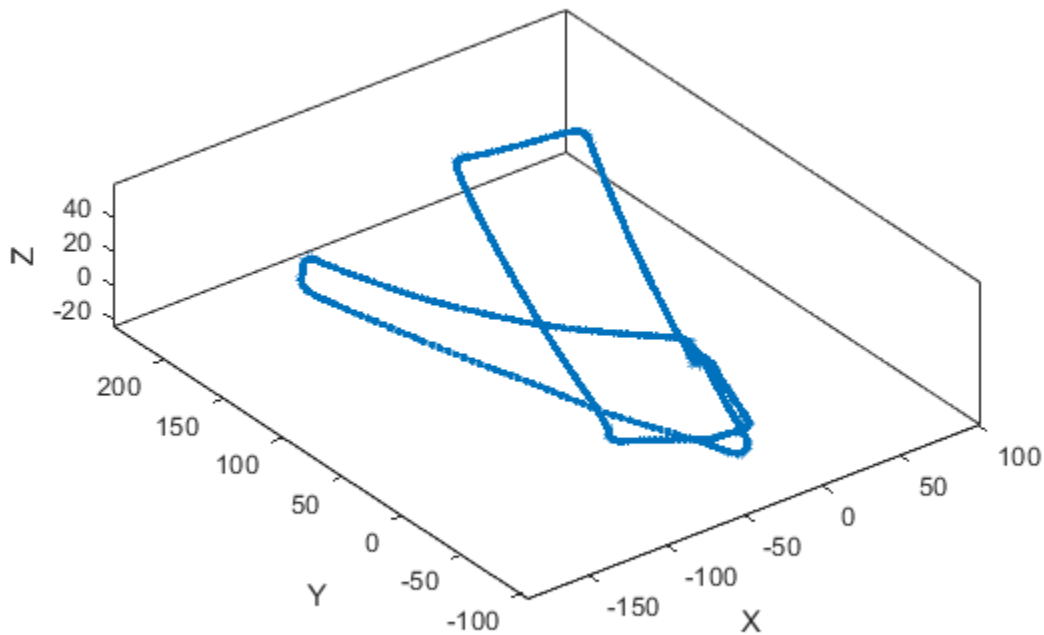
```

viewId = viewId + 1;

ptCloudPrev = ptCloud;
initTform = relTform;

if n>1 && mod(n, displayRate) == 1
    hG = plot(vSet, "Parent", hAxBefore);
    drawnow update
end
end

```



Create a pose graph from the view set by using the `createPoseGraph` method. The pose graph is a **digraph** object with:

- Nodes containing the absolute pose of each view
- Edges containing the relative pose constraints of each connection

```

G = createPoseGraph(vSet);
disp(G)

```

digraph with properties:

```

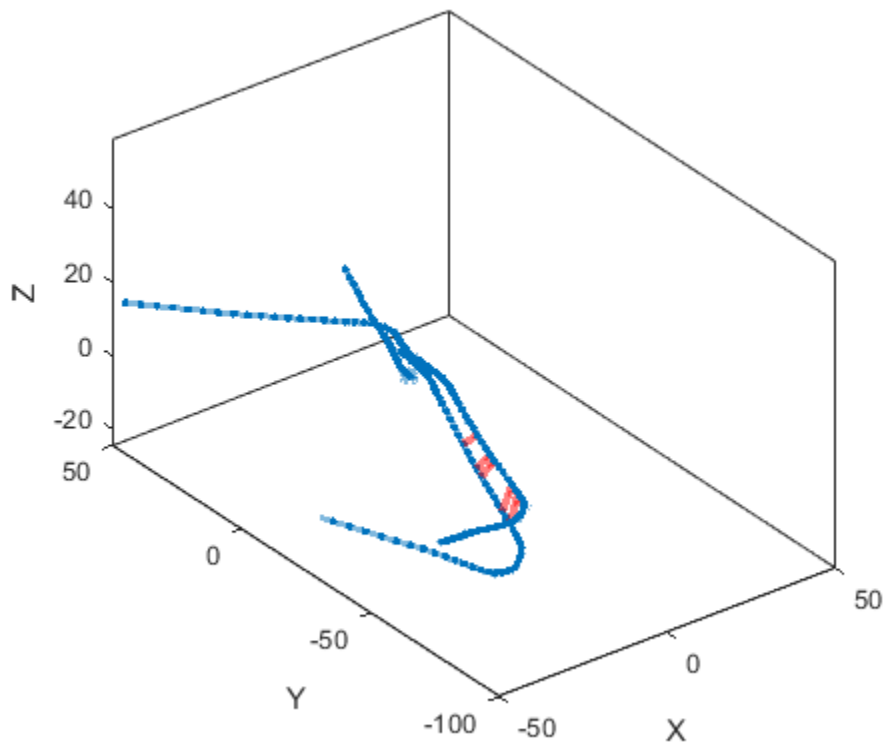
Edges: [507×3 table]
Nodes: [503×2 table]

```

In addition to the odometry connections between successive views, the view set now includes loop closure connections. For example, notice the new connections between the second loop traversal and the first loop traversal. These are loop closure connections. These can be identified as edges in the graph whose end nodes are not consecutive.

```
% Update axes limits to focus on loop closure connections
xlim(hAxBefore, [-50 50]);
ylim(hAxBefore, [-100 50]);

% Find and highlight loop closure connections
loopEdgeIds = find(abs(diff(G.Edges.EndNodes, 1, 2)) > 1);
highlight(hG, 'Edges', loopEdgeIds, 'EdgeColor', 'red', 'LineWidth', 3)
```

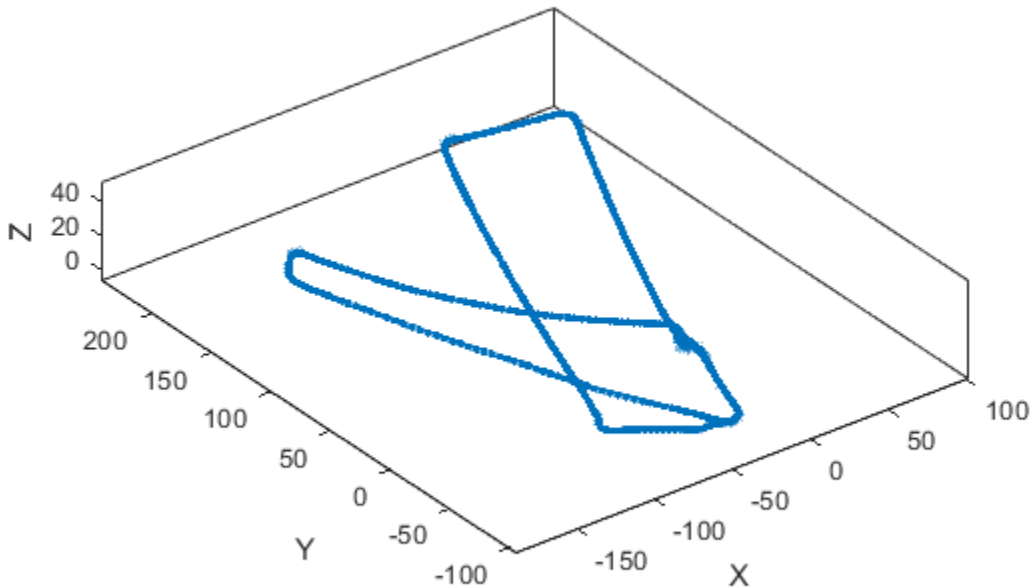


Optimize the pose graph using `optimizePoseGraph`.

```
optimG = optimizePoseGraph(G, 'g2o-levenberg-marquardt');
vSetOptim = updateView(vSet, optimG.Nodes);
```

Display the view set with optimized poses. Notice that the detected loops are now merged, resulting in a more accurate trajectory.

```
plot(vSetOptim, 'Parent', hAxBefore)
```



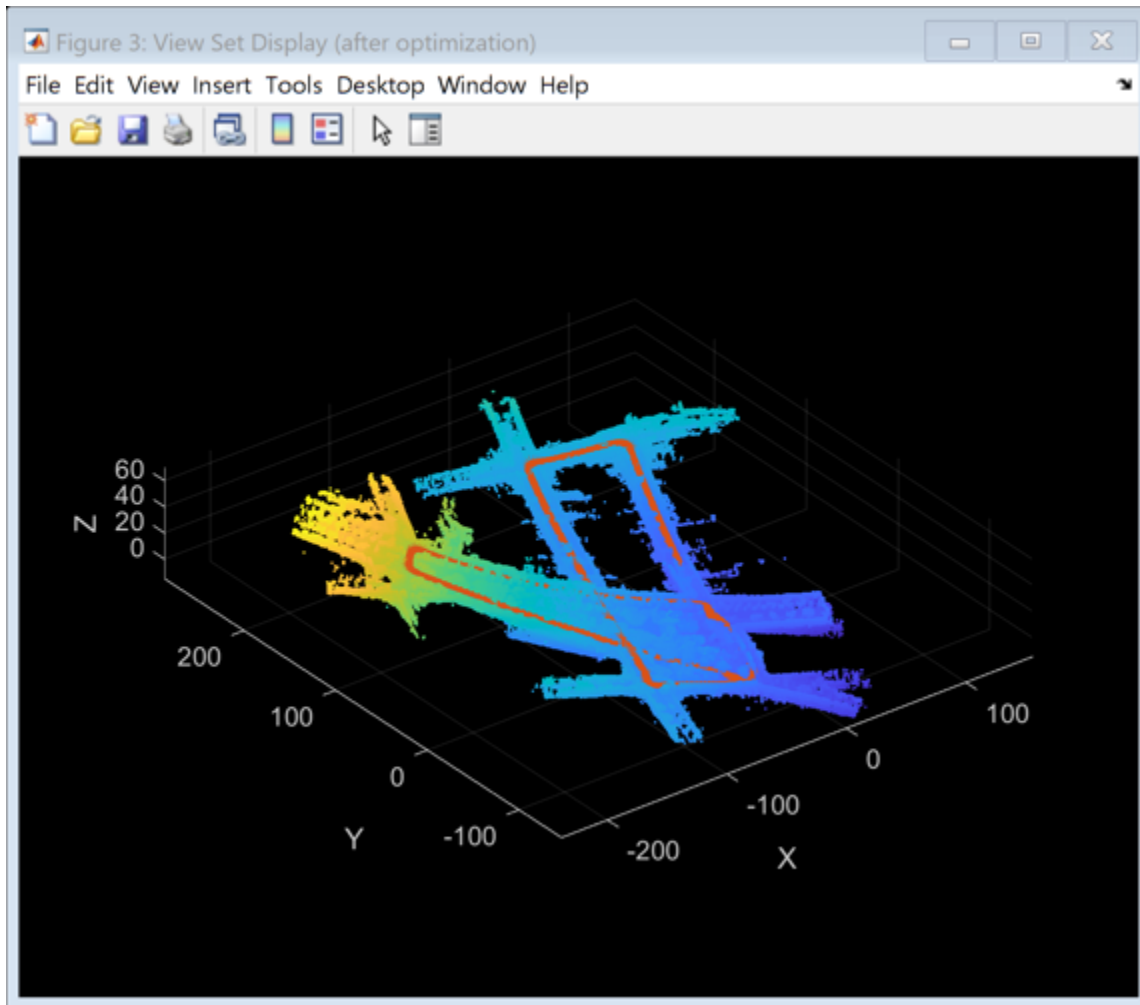
The absolute poses in the optimized view set can now be used to build a more accurate map. Use the `pcalign` function to align the view set point clouds with the optimized view set absolute poses into a single point cloud map. Specify a grid size to control the resolution of the created point cloud map.

```
mapGridSize = 0.2;
ptClouds = vSetOptim.Views.PointCloud;
absPoses = vSetOptim.Views.AbsolutePose;
ptCloudMap = pcalign(ptClouds, absPoses, mapGridSize);

hFigAfter = figure('Name', 'View Set Display (after optimization)');
hAxAfter = axes(hFigAfter);
pcshow(ptCloudMap, 'Parent', hAxAfter);

% Overlay view set display
hold on
plot(vSetOptim, 'Parent', hAxAfter);

helperMakeFigurePublishFriendly(hFigAfter);
```



While accuracy can still be improved, this point cloud map is significantly more accurate.

References

- 1 G. Kim and A. Kim, "Scan Context: Egocentric Spatial Descriptor for Place Recognition Within 3D Point Cloud Map," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid, 2018, pp. 4802-4809.

Supporting Functions and Classes

helperReadDataset reads data from specified folder into a timetable.

```
function datasetTable = helperReadDataset(dataFolder)
%helperReadDataset Read Velodyne SLAM Dataset data into a timetable
% datasetTable = helperReadDataset(dataFolder) reads data from the
% folder specified in dataFolder into a timetable. The function
% expects data from the Velodyne SLAM Dataset.
%
% See also fileDatastore, helperReadINSConfigFile.

% Point clouds are stored as PNG files in the scenario1 folder
pointCloudFilePattern = fullfile(dataFolder, 'scenario1', 'scan*.png');
```

```

% Create a file datastore to read in files in the right order
fileDS = fileDatastore(pointCloudFilePattern, 'ReadFcn', ...
    @helperReadPointCloudFromFile);

% Extract the file list from the datastore
pointCloudFiles = fileDS.Files;

imuConfigFile = fullfile(dataFolder, 'scenario1', 'imu.cfg');
insDataTable = helperReadINSConfigFile(imuConfigFile);

% Delete the bad row from the INS config file
insDataTable(1447, :) = [];

% Remove columns that will not be used
datasetTable = removevars(insDataTable, ...
    {'Num_Satellites', 'Latitude', 'Longitude', 'Altitude', 'Omega_Heading', ...
    'Omega_Pitch', 'Omega_Roll', 'V_X', 'V_Y', 'V_ZDown'});

datasetTable = addvars(datasetTable, pointCloudFiles, 'Before', 1, ...
    'NewVariableNames', "PointCloudFileName");
end

```

helperProcessPointCloud processes a point cloud by removing points belonging to the ground plane and the ego vehicle.

```

function ptCloud = helperProcessPointCloud(ptCloudIn, method)
%helperProcessPointCloud Process pointCloud to remove ground and ego vehicle
% ptCloud = helperProcessPointCloud(ptCloudIn, method) processes
% ptCloudIn by removing the ground plane and the ego vehicle.
% method can be "planefit" or "rangefloodfill".
%
% See also pcfiteplane, pointCloud/findNeighborsInRadius.

arguments
    ptCloudIn (1,1) pointCloud
    method     string     {mustBeMember(method, ["planefit","rangefloodfill"])} = "rangefloodfill"
end

isOrganized = ~ismatrix(ptCloudIn.Location);

if (method=="rangefloodfill" && isOrganized)
    % Segment ground using floodfill on range image
    groundFixedIdx = segmentGroundFromLidarData(ptCloudIn, ...
        "ElevationAngleDelta", 11);
else
    % Segment ground as the dominant plane with reference normal
    % vector pointing in positive z-direction
    maxDistance = 0.4;
    maxAngularDistance = 5;
    referenceVector = [0 0 1];

    [~, groundFixedIdx] = pcfiteplane(ptCloudIn, maxDistance, ...
        referenceVector, maxAngularDistance);
end

if isOrganized
    groundFixed = false(size(ptCloudIn.Location,1),size(ptCloudIn.Location,2));

```

```

else
    groundFixed = false(ptCloudIn.Count, 1);
end
groundFixed(groundFixedIdx) = true;

% Segment ego vehicle as points within a given radius of sensor
sensorLocation = [0 0 0];
radius = 3.5;
egoFixedIdx = findNeighborsInRadius(ptCloudIn, sensorLocation, radius);

if isOrganized
    egoFixed = false(size(ptCloudIn.Location,1),size(ptCloudIn.Location,2));
else
    egoFixed = false(ptCloudIn.Count, 1);
end
egoFixed(egoFixedIdx) = true;

% Retain subset of point cloud without ground and ego vehicle
if isOrganized
    indices = ~groundFixed & ~egoFixed;
else
    indices = find(~groundFixed & ~egoFixed);
end

ptCloud = select(ptCloudIn, indices);
end

```

helperComputeInitialEstimateFromINS estimates an initial transformation for registration from INS readings.

```

function initTform = helperComputeInitialEstimateFromINS(initTform, insData)

% If no INS readings are available, return
if isempty(insData)
    return;
end

% The INS readings are provided with X pointing to the front, Y to the left
% and Z up. Translation below accounts for transformation into the lidar
% frame.
insToLidarOffset = [0 -0.79 -1.73]; % See DATAFORMAT.txt
Tnow = [-insData.Y(end), insData.X(end), insData.Z(end)].' + insToLidarOffset';
Tbef = [-insData.Y(1) , insData.X(1) , insData.Z(1)].' + insToLidarOffset';

% Since the vehicle is expected to move along the ground, changes in roll
% and pitch are minimal. Ignore changes in roll and pitch, use heading only.
Rnow = rotmat(Quaternion([insData.Heading(end) 0 0], 'euler', 'ZYX', 'point'), 'point');
Rbef = rotmat(Quaternion([insData.Heading(1) 0 0], 'euler', 'ZYX', 'point'), 'point');

T = [Rbef Tbef;0 0 0 1] \ [Rnow Tnow;0 0 0 1];

initTform = rigid3d(T. ');
end

```

helperMakeFigurePublishFriendly adjusts figures so that screenshot captured by publish is correct.

```
function helperMakeFigurePublishFriendly(hFig)
if ~isempty(hFig) && isValid(hFig)
    hFig.HandleVisibility = 'callback';
end
end
```

helperFeatureSearcher creates an object that can be used to search for closest feature matches.

helperLoopClosureDetector creates an object that can be used to detect loop closures using scan context feature descriptors.

See Also

Functions

[createPoseGraph](#) | [optimizePoses](#) | [pcregisterndt](#) | [pcshow](#)

Objects

[pcviewset](#) | [pointCloud](#) | [rigid3d](#)

More About

- “Build a Map from Lidar Data” (Automated Driving Toolbox)
- “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox)

External Websites

- [Velodyne SLAM Dataset](#)

3-D Point Cloud Registration and Stitching

This example shows how to combine multiple point clouds to reconstruct a 3-D scene using Iterative Closest Point (ICP) algorithm.

Overview

This example stitches together a collection of point clouds that was captured with Kinect to construct a larger 3-D view of the scene. The example applies ICP to two successive point clouds. This type of reconstruction can be used to develop 3-D models of objects or build 3-D world maps for simultaneous localization and mapping (SLAM).

Register Two Point Clouds

```
dataFile = fullfile(toolboxdir('vision'), 'visiondata', 'livingRoom.mat');
load(dataFile);

% Extract two consecutive point clouds and use the first point cloud as
% reference.
ptCloudRef = livingRoomData{1};
ptCloudCurrent = livingRoomData{2};
```

The quality of registration depends on data noise and initial settings of the ICP algorithm. You can apply preprocessing steps to filter the noise or set initial property values appropriate for your data. Here, preprocess the data by downsampling with a box grid filter and set the size of grid filter to be 10cm. The grid filter divides the point cloud space into cubes. Points within each cube are combined into a single output point by averaging their X,Y,Z coordinates.

```
gridSize = 0.1;
fixed = pcdsample(ptCloudRef, 'gridAverage', gridSize);
moving = pcdsample(ptCloudCurrent, 'gridAverage', gridSize);

% Note that the downsampling step does not only speed up the registration,
% but can also improve the accuracy.
```

To align the two point clouds, we use the ICP algorithm to estimate the 3-D rigid transformation on the downsampled data. We use the first point cloud as the reference and then apply the estimated transformation to the original second point cloud. We need to merge the scene point cloud with the aligned point cloud to process the overlapped points.

Begin by finding the rigid transformation for aligning the second point cloud with the first point cloud. Use it to transform the second point cloud to the reference coordinate system defined by the first point cloud.

```
tform = pregistericp(moving, fixed, 'Metric', 'pointToPlane', 'Extrapolate', true);
ptCloudAligned = pctransform(ptCloudCurrent, tform);
```

We can now create the world scene with the registered data. The overlapped region is filtered using a 1.5cm box grid filter. Increase the merge size to reduce the storage requirement of the resulting scene point cloud, and decrease the merge size to increase the scene resolution.

```
mergeSize = 0.015;
ptCloudScene = pcmerge(ptCloudRef, ptCloudAligned, mergeSize);

% Visualize the input images.
figure
```



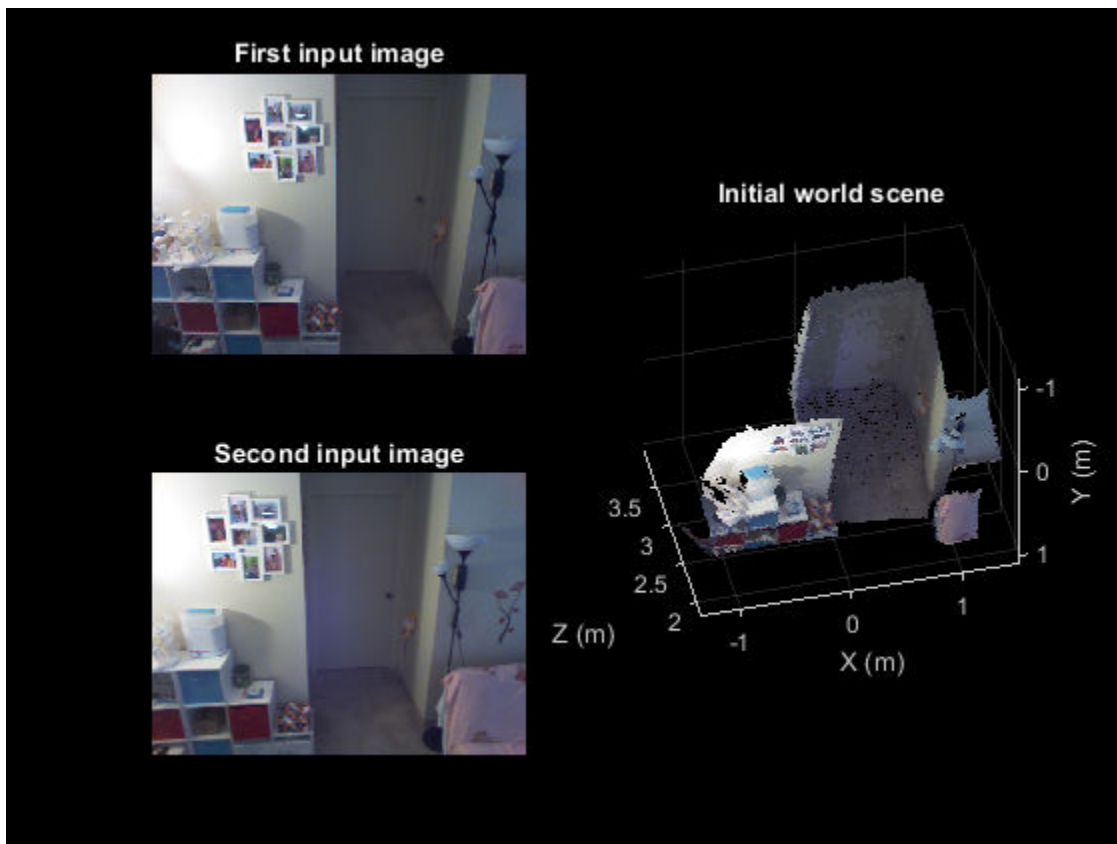
```

subplot(2,2,1)
imshow(ptCloudRef.Color)
title('First input image','Color','w')
drawnow

subplot(2,2,3)
imshow(ptCloudCurrent.Color)
title('Second input image','Color','w')
drawnow

% Visualize the world scene.
subplot(2,2,[2,4])
pcshow(ptCloudScene, 'VerticalAxis','Y', 'VerticalAxisDir', 'Down')
title('Initial world scene')
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Z (m)')

```



```
drawnow
```

Stitch a Sequence of Point Clouds

To compose a larger 3-D scene, repeat the same procedure as above to process a sequence of point clouds. Use the first point cloud to establish the reference coordinate system. Transform each point cloud to the reference coordinate system. This transformation is a multiplication of pairwise transformations.

```
% Store the transformation object that accumulates the transformation.
accumTform = tform;

figure
hAxes = pcshow(ptCloudScene, 'VerticalAxis','Y', 'VerticalAxisDir', 'Down');
title('Updated world scene')
% Set the axes property for faster rendering
hAxes.CameraViewAngleMode = 'auto';
hScatter = hAxes.Children;

for i = 3:length(livingRoomData)
    ptCloudCurrent = livingRoomData{i};

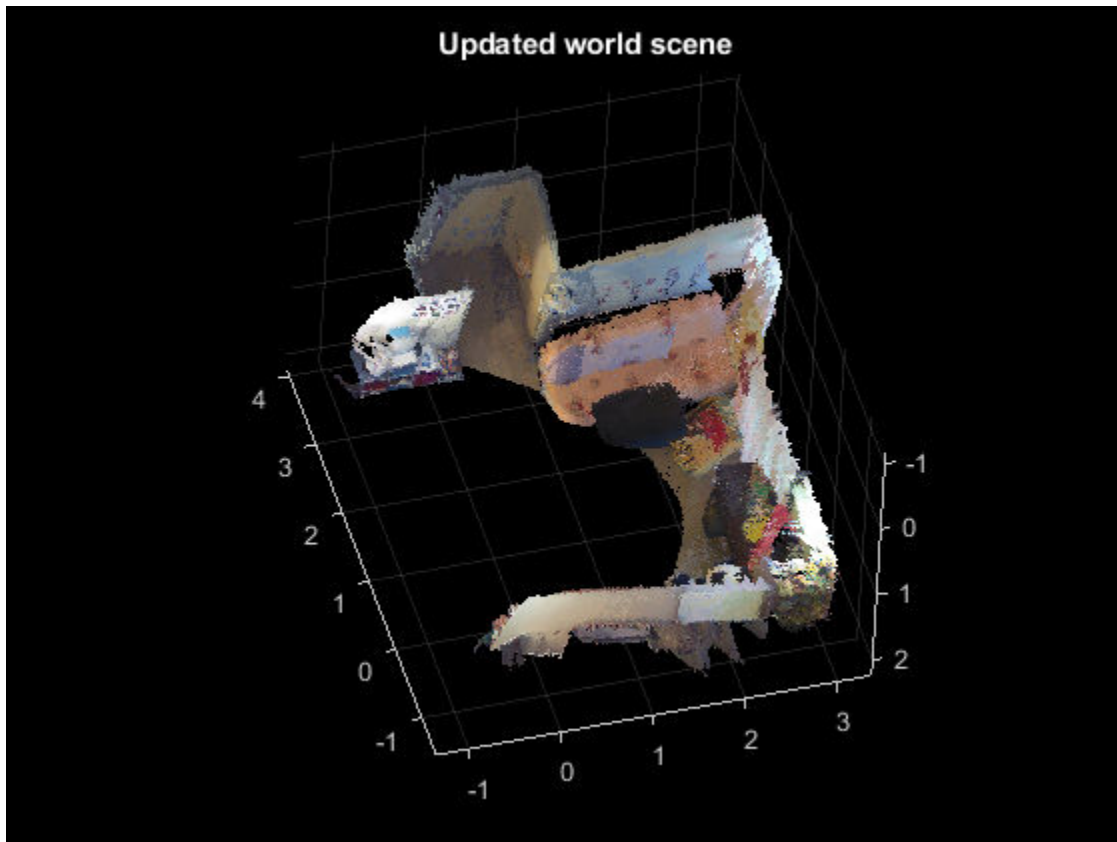
    % Use previous moving point cloud as reference.
    fixed = moving;
    moving = pcdsample(ptCloudCurrent, 'gridAverage', gridSize);

    % Apply ICP registration.
    tform = pcregistericp(moving, fixed, 'Metric','pointToPlane','Extrapolate', true);

    % Transform the current point cloud to the reference coordinate system
    % defined by the first point cloud.
    accumTform = affine3d(tform.T * accumTform.T);
    ptCloudAligned = pctransform(ptCloudCurrent, accumTform);

    % Update the world scene.
    ptCloudScene = pcmerge(ptCloudScene, ptCloudAligned, mergeSize);

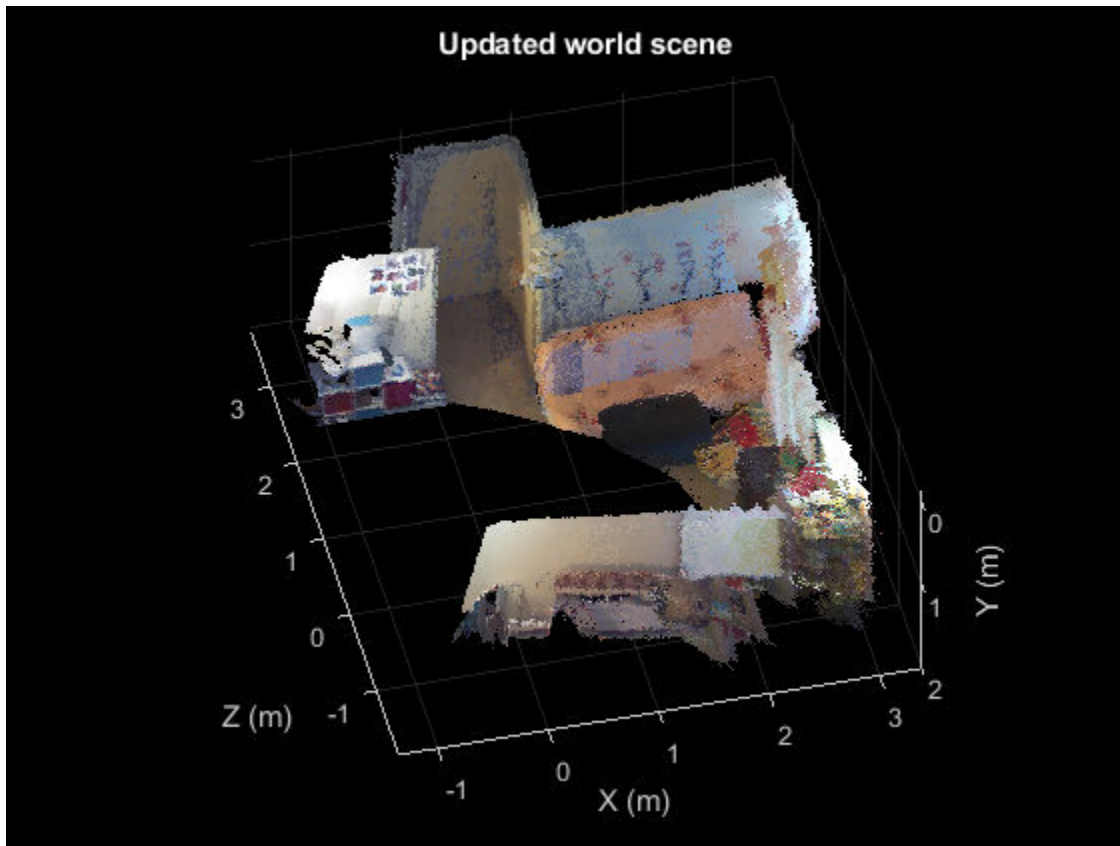
    % Visualize the world scene.
    hScatter.XData = ptCloudScene.Location(:,1);
    hScatter.YData = ptCloudScene.Location(:,2);
    hScatter.ZData = ptCloudScene.Location(:,3);
    hScatter.CData = ptCloudScene.Color;
    drawnow('limitrate')
end
```



```

% During the recording, the Kinect was pointing downward. To visualize the
% result more easily, let's transform the data so that the ground plane is
% parallel to the X-Z plane.
angle = -pi/10;
A = [1,0,0,0;...
     0, cos(angle), sin(angle), 0; ...
     0, -sin(angle), cos(angle), 0; ...
     0 0 0 1];
ptCloudScene = pctransform(ptCloudScene, affine3d(A));
pcshow(ptCloudScene, 'VerticalAxis','Y', 'VerticalAxisDir', 'Down', ...
       'Parent', hAxes)
title('Updated world scene')
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Z (m)')

```



Computer Vision with Simulink Examples

- “Multicore Simulation of Video Processing System” on page 6-2
- “Concentricity Inspection” on page 6-7
- “Object Counting” on page 6-9
- “Video Focus Assessment” on page 6-11
- “Video Compression” on page 6-13
- “Barcode Recognition” on page 6-15
- “Motion Detection” on page 6-17
- “Pattern Matching” on page 6-19
- “Scene Change Detection” on page 6-22
- “Surveillance Recording” on page 6-24
- “Traffic Warning Sign Recognition” on page 6-26
- “Abandoned Object Detection” on page 6-29
- “Color-based Road Tracking” on page 6-32
- “Detect and Track Face” on page 6-36
- “Lane Departure Warning System” on page 6-43
- “Tracking Cars Using Foreground Detection” on page 6-47
- “Tracking Cars Using Optical Flow” on page 6-50
- “Tracking Based on Color” on page 6-52
- “Video Mosaicking” on page 6-54
- “Video Stabilization” on page 6-59
- “Periodic Noise Reduction” on page 6-61
- “Rotation Correction” on page 6-63
- “Barcode Recognition Using Live Video Acquisition” on page 6-66
- “Edge Detection Using Live Video Acquisition” on page 6-68
- “Noise Removal and Image Sharpening” on page 6-73

Multicore Simulation of Video Processing System

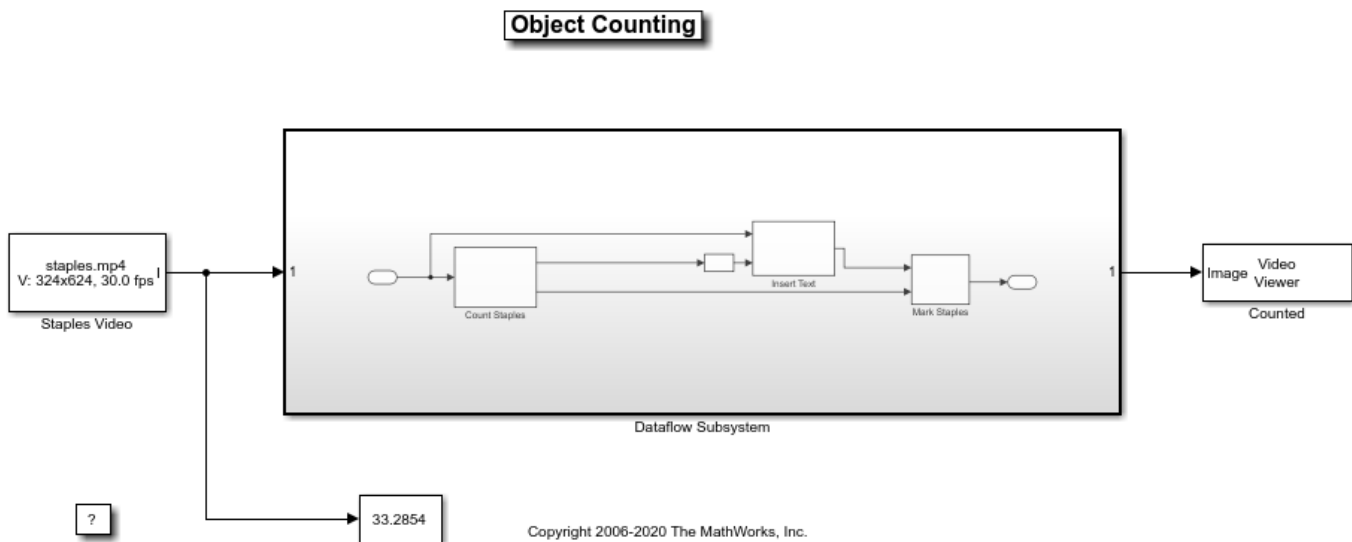
This example shows how to run a video processing system on multiple cores using dataflow execution domain in Simulink®.

Introduction

Dataflow execution domain allows you to make use of multiple cores in the design of computationally intensive systems. This example shows how dataflow as the execution domain of a subsystem improves simulation performance of the model. To learn more about dataflow and how to run Simulink models using multiple threads, see “Multicore Execution using Dataflow Domain” (DSP System Toolbox).

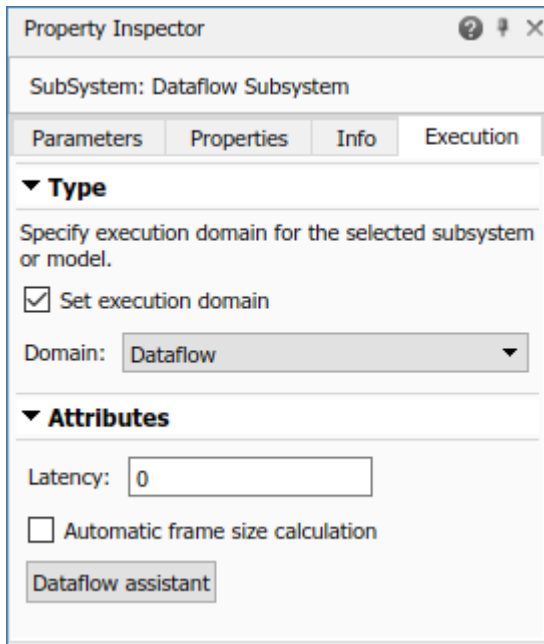
Object Counting in Video

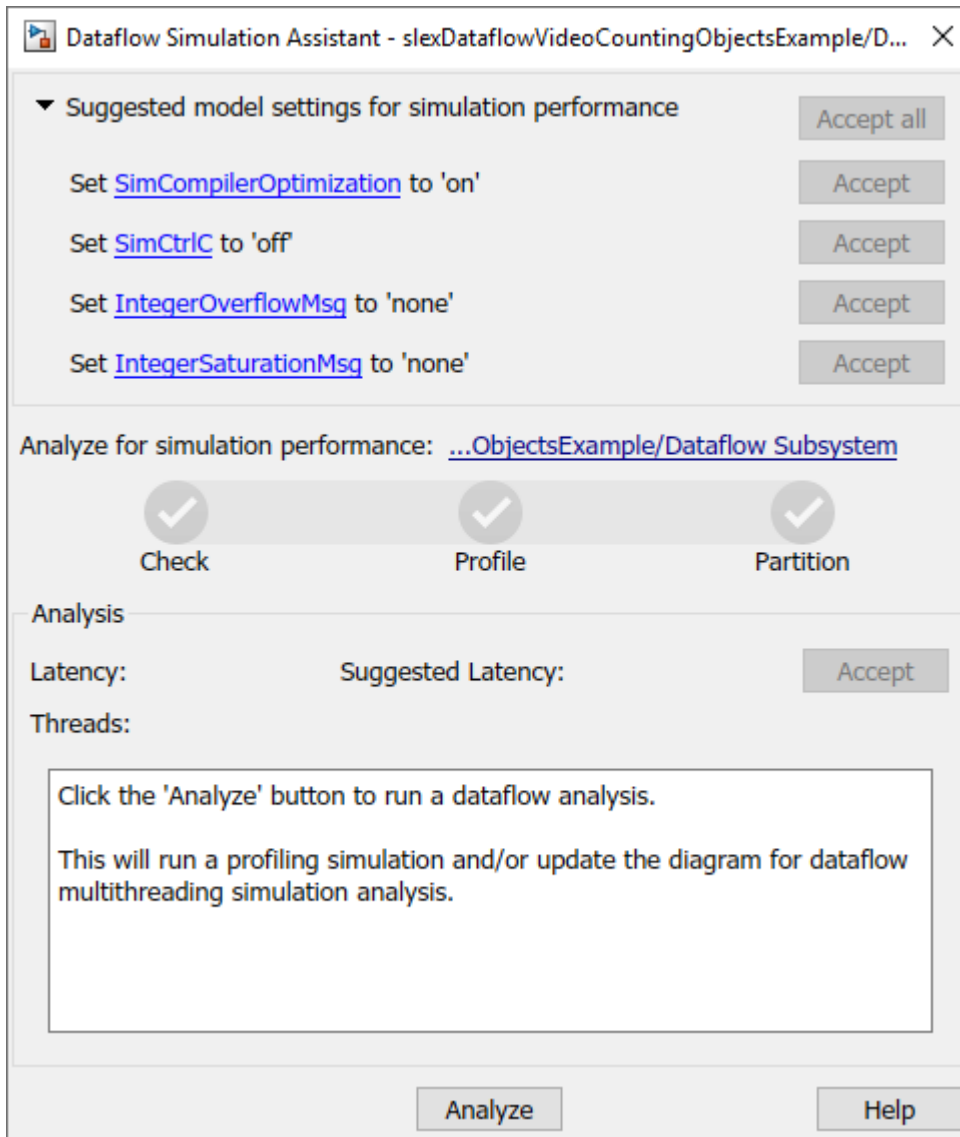
This example shows how to use basic morphological operators to extract information from a video stream. In this case the model counts the number of staples in each video frame. The model uses the Top-hat block to remove uneven illumination and then the Autothreshold block to convert it into a binary image. The Blob Analysis block is then used to count the number of staples and compute the centroid of each staple. The Draw markers and insert text block are used to mark the staples and write the number of staples found on the video frame.



Setting up the Dataflow Subsystem

This example uses dataflow domain in Simulink to make use of multiple cores on your desktop to improve simulation performance. The Domain parameter of the Dataflow Subsystem in this model is set as **Dataflow**. You can view this by selecting the subsystem and then selecting **View>Property Inspector**. Dataflow domains automatically partition your model and simulate the system using multiple threads for better simulation performance. Once you set the Domain parameter to Dataflow, you can use Dataflow Simulation Assistant to analyze your model to get better performance. You can open Dataflow Simulation Assistant, by clicking on the **Dataflow assistant** button below the **Automatic frame size calculation** parameter in Property Inspector.



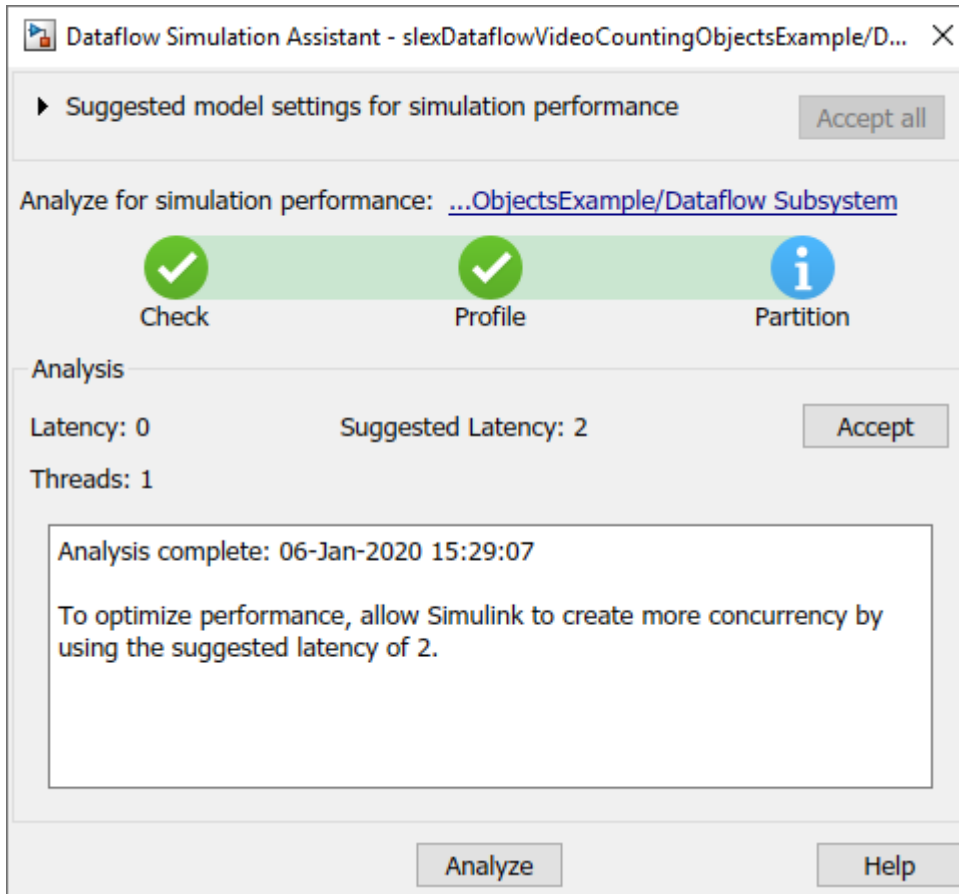


Analyzing Concurrency in Dataflow Subsystem

The Dataflow Simulation Assistant suggests changing model settings for optimal simulation performance. To accept the proposed model settings, next to **Suggested model settings for simulation performance**, click **Accept all**. Alternatively, you can expand the section to change the settings individually. In this example the model settings are already optimal. In the Dataflow Simulation Assistant, click the **Analyze** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Dataflow Simulation Assistant shows how many threads the dataflow subsystem will use during simulation.

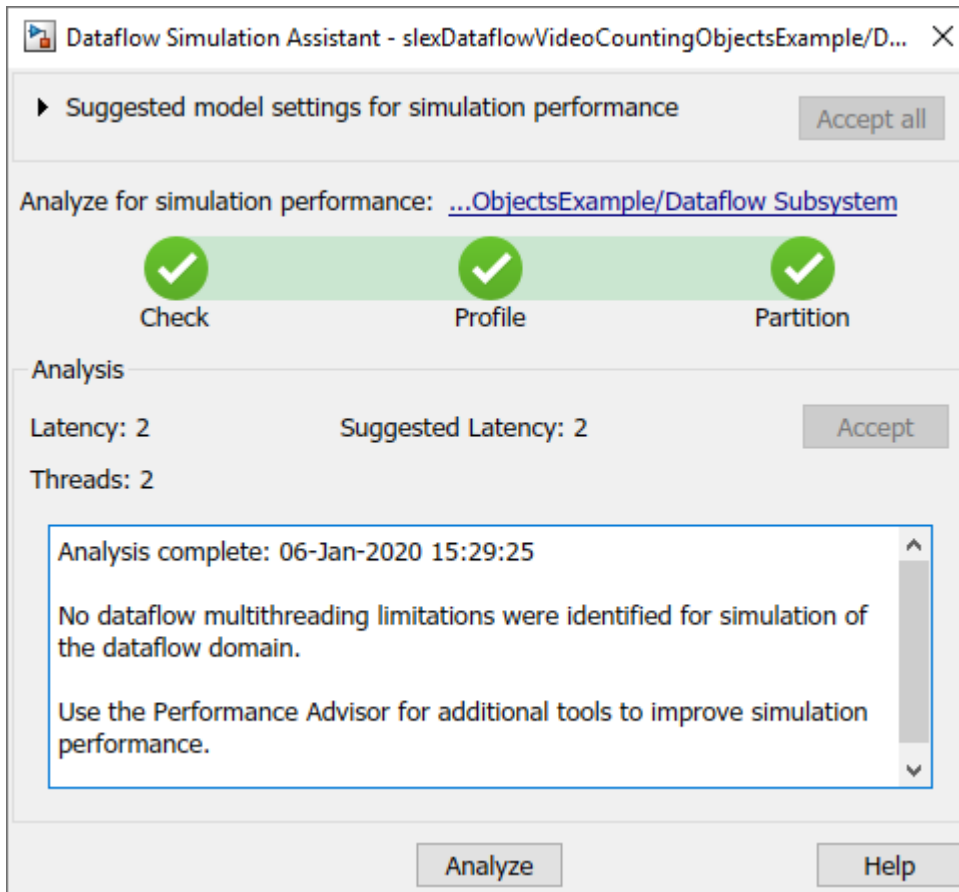
After analyzing the model, the assistant shows one thread because the data dependency between the blocks in the model prevents blocks from being executed concurrently. By pipelining the data dependent blocks, the Dataflow Subsystem can increase concurrency for higher data throughput. The Dataflow Simulation Assistant shows the recommended number of pipeline delays as Suggested Latency. The suggested latency value is computed to give the best performance.

The following diagram shows the Dataflow Simulation Assistant where the Dataflow Subsystem currently specifies a latency value of zero, and the suggested latency for the system is two.



Click the **Accept** button next to **Suggested Latency** in the Dataflow Simulation Assistant to use the recommended latency for the Dataflow Subsystem. This value can also be entered directly in the Property Inspector for "Latency" parameter. Simulink shows the latency parameter value using Z^{-n} tags at the output ports of the dataflow subsystem.

Dataflow Simulation Assistant now shows the number of threads as 2 meaning that the blocks inside the dataflow subsystem simulate in parallel using 2 threads.



Multicore Simulation Performance

We measure the performance improvement of using dataflow domain by comparing the execution time taken for running model with and without using dataflow. Execution time is measured using the `sim` command, which returns the simulation execution time of the model. While measuring the execution time the Video Viewer block is commented to measure the time taken primarily for the Dataflow Subsystem. These numbers and analysis were published on a Windows desktop computer with Intel® Xeon® CPU W-2133 @ 3.6 GHz 6 Cores 12 Threads processor.

```
Simulation execution time for multithreaded model = 7.74s
Simulation execution time for single-threaded model = 11.37s
Actual speedup with dataflow: 1.5x
```

Summary

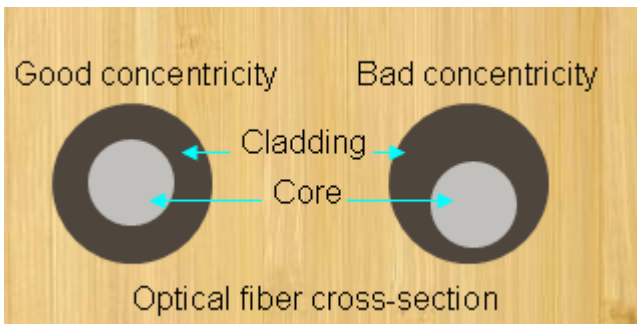
This example shows how multithreading using dataflow domain can improve performance in a video processing model using multiple cores on the desktop.

Concentricity Inspection

This example shows how to inspect the concentricity of both the core and the cladding in a cross-section of optical fiber. Concentricity is a measure of how centered the core is within the cladding.

First, the example uses the Blob Analysis block to determine the centroid of the cladding. It uses this centroid to find a point on the cladding's outer boundary. Using this as a starting point, the Trace Boundaries block defines the cladding's outer boundary. Then the example uses these boundary points to compute the cladding's center and radius using a least-square, circle-fitting algorithm. If the distance between the cladding's centroid and the center of its outer boundary is within a certain tolerance, the fiber optic cable is in acceptable condition.

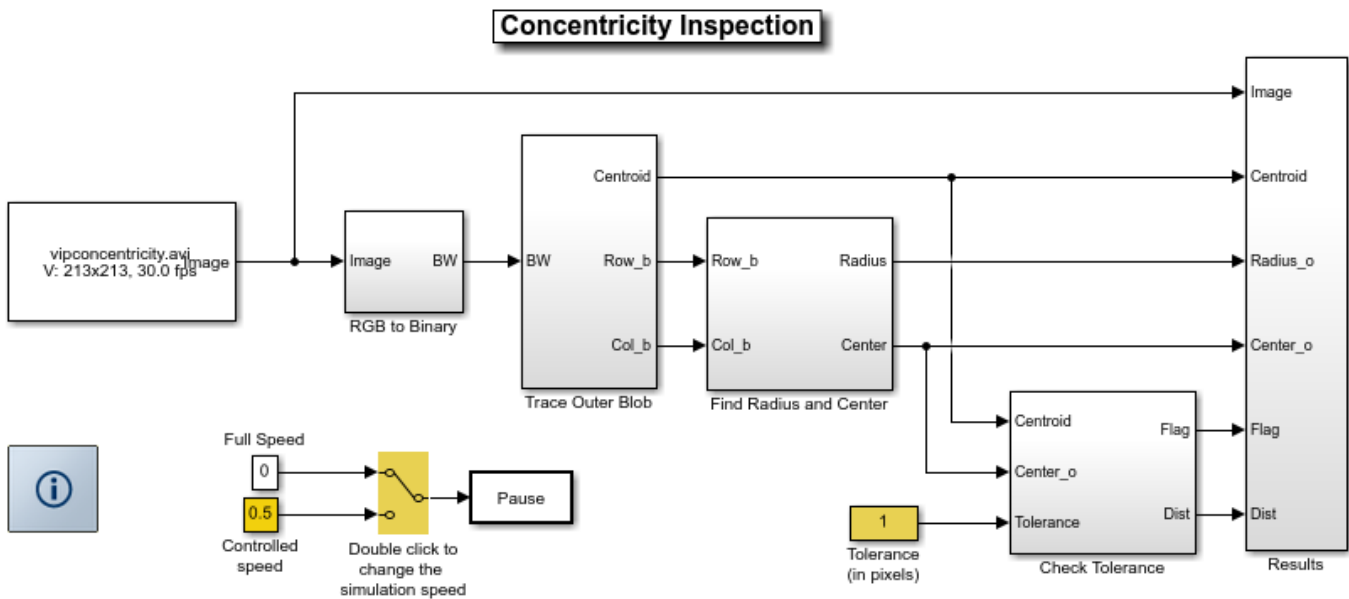
The following figure shows examples of optical fibers with good and bad concentricity:



Example Model

The following figure shows the Concentricity Inspection example model:

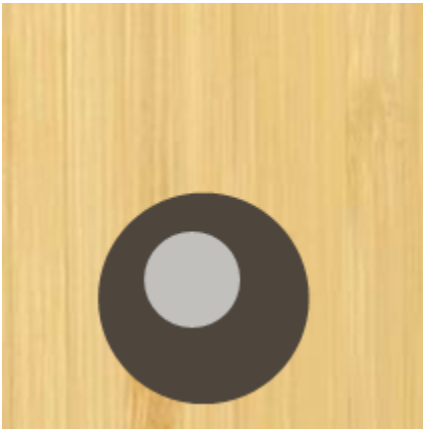
```
open_system('vipconcentricity');
```



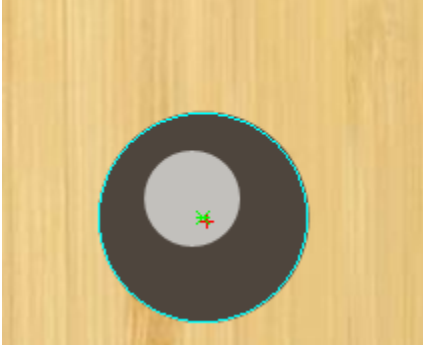
Concentricity Inspection Results

In the Results window, you can see that the example marked the cladding's centroid with a red '+'. It marked the center of the cladding's outer boundary with a green '*'. When the distance between these two markers is within an acceptable tolerance, the example labels the cross-section of fiber optic cable "Concentricity: Good". Otherwise, it labels it "Concentricity: Bad". The example also displays the distance, in pixels, between the cladding's centroid and the center of the cladding's outer boundary.

```
close_system('vipconcentricity');  
sim('vipconcentricity', 0.0333333);  
  
set(allchild(0), 'Visible', 'off');  
  
captureVideoViewerFrame('vipconcentricity/Results/Original');  
captureVideoViewerFrame('vipconcentricity/Results/Results');
```



Concentricity: Bad
Distance in pixels: 2



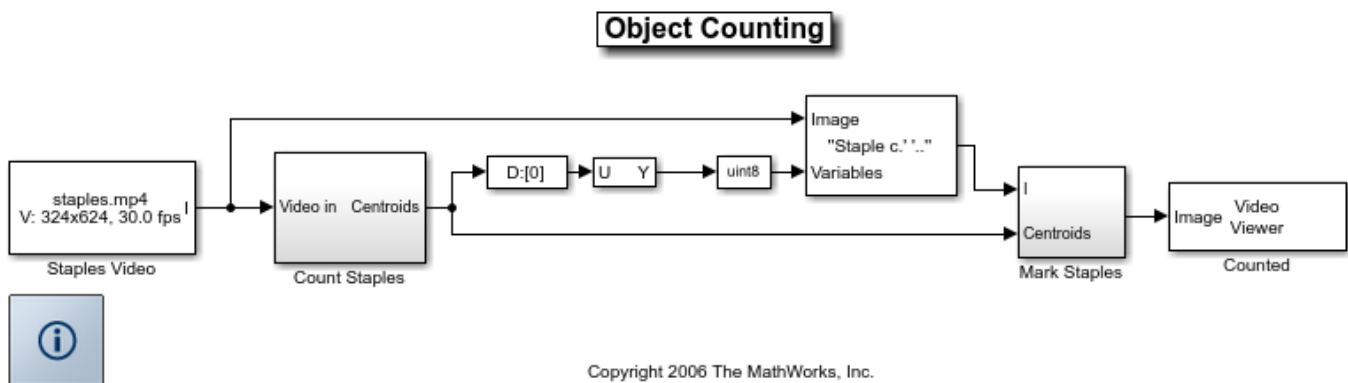
```
close_system('vipconcentricity', 0);
```

Object Counting

This example shows how to use basic morphological operators to extract information from a video stream. In this case, the model counts the number of staples in each video frame. Note that the focus and lighting change in each video frame.

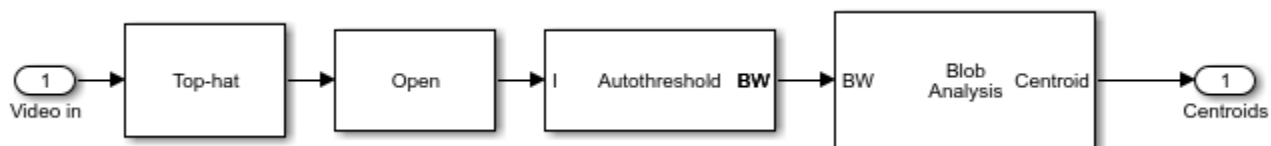
Example Model

The following figure shows the Object Counting model.



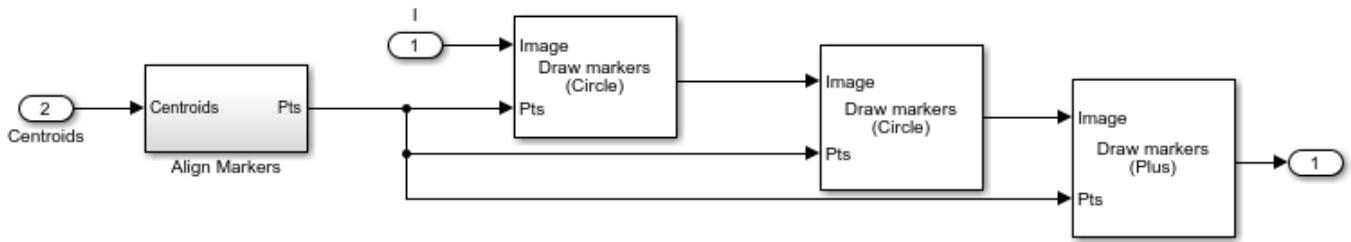
Count Staples Subsystem

The model uses the Top-hat block to remove uneven illumination and the Opening block to widen the gaps between the staples. Due to changes in overall lighting intensity, the model cannot apply a single threshold value to all of the video frames. Instead, it uses the Autothreshold block to compute a threshold for each frame. Once the model applies the threshold to separate the staples, it uses the Blob Analysis block to count the number of staples in each frame and to calculate the centroid of each staple. The model passes the total number of staples in each frame to the Insert Text block in the main model. This block embeds this information on each video frame.



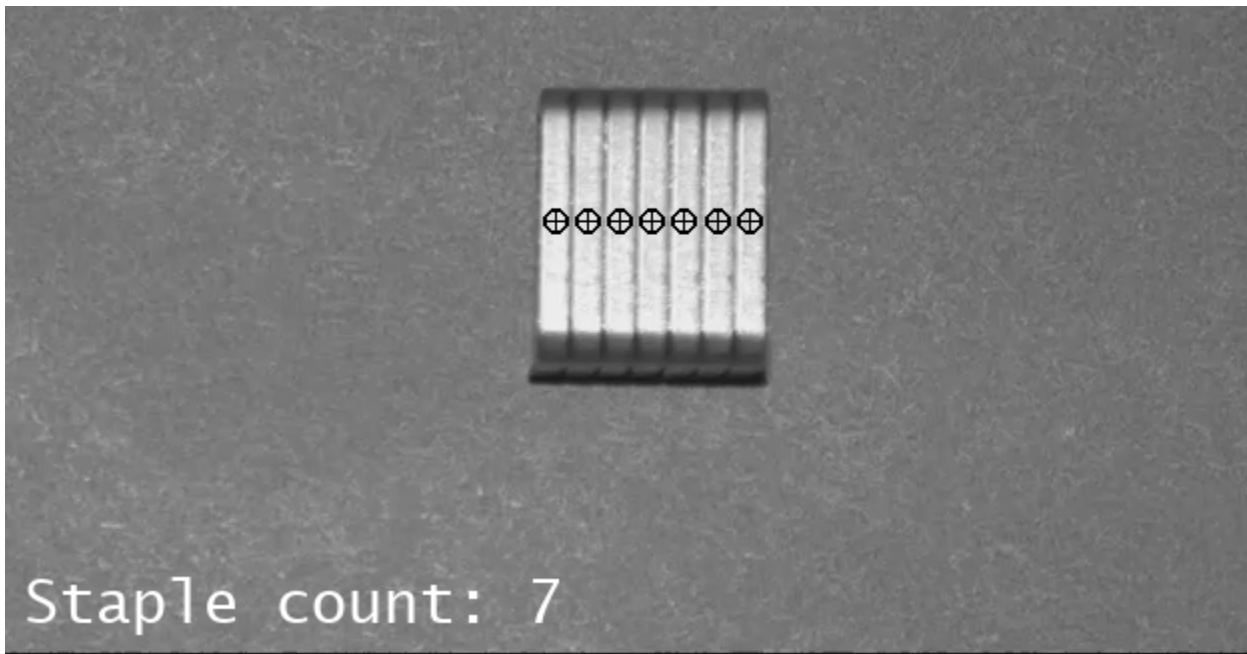
Mark Staples Subsystem

The model passes the centroid information to a series of Draw Markers blocks, which mark the centroids of each staple.



Object Counting Results

The Counted window displays one frame of the original video and the segmented staples in that frame. The number of staples is displayed in the lower left corner.

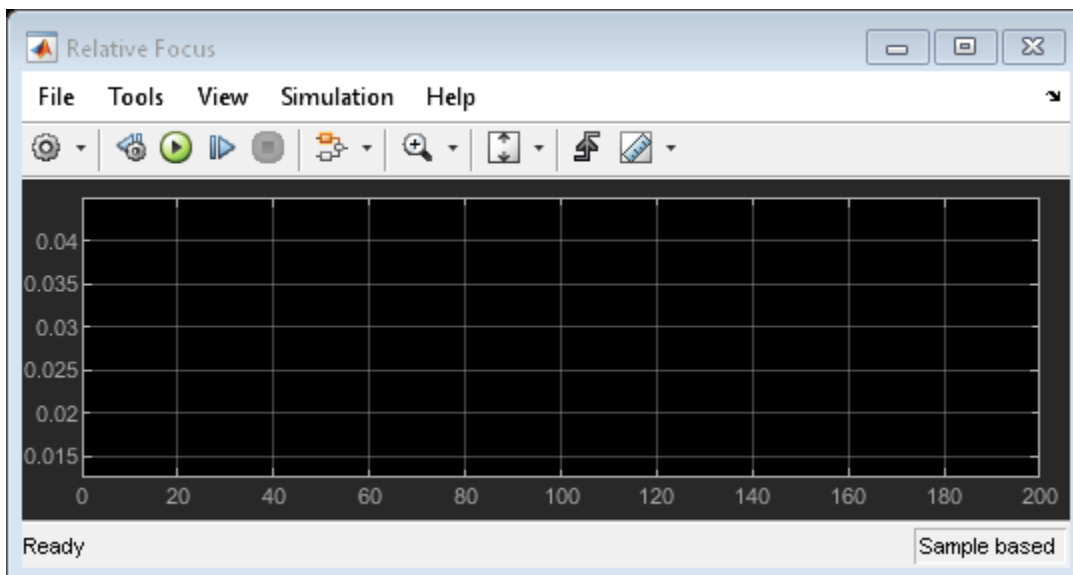
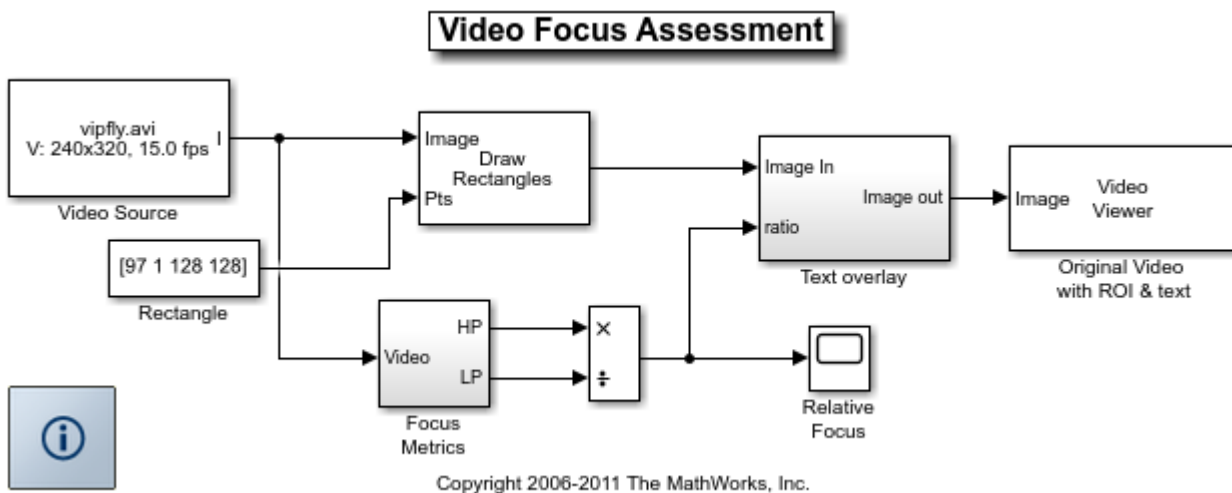


Video Focus Assessment

This example shows how to determine whether video frames are in focus by using the ratio of the high spatial frequency content to the low spatial frequency content within a region of interest (ROI). When this ratio is high, the video is in focus. When this ratio is low, the video is out of focus.

Example Model

The following figure shows the Video Focus Assessment model:

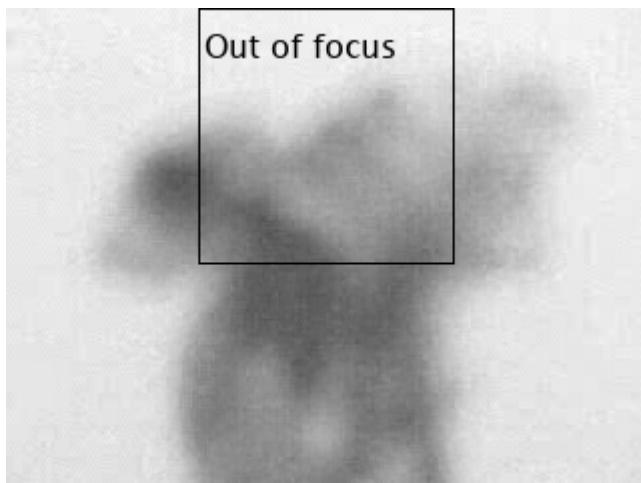
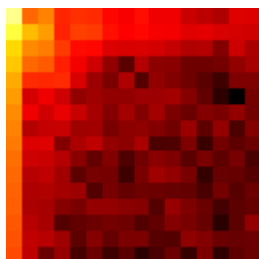


Video Focus Assessment Results

This example shows a video sequence that is moving in and out of focus. The model uses the Draw Shapes block to highlight an ROI on the video frames and the Insert Text block to indicate whether or not the video is in focus.

The Relative Focus window displays a plot of the ratio of the high spatial frequency content to the low spatial frequency content within the ROI. This ratio is an indication of the relative focus adjustment of the video camera. When this ratio is high, the video is in focus. When this ratio is low, the video is out of focus. Although it is possible to judge the relative focus of a camera with respect to the video using 2-D filters, the approach used in this example enables you to see the relationship between the high spatial frequency content of the video and its focus.

The FFT Data window shows the 2-D FFT data within the ROI.

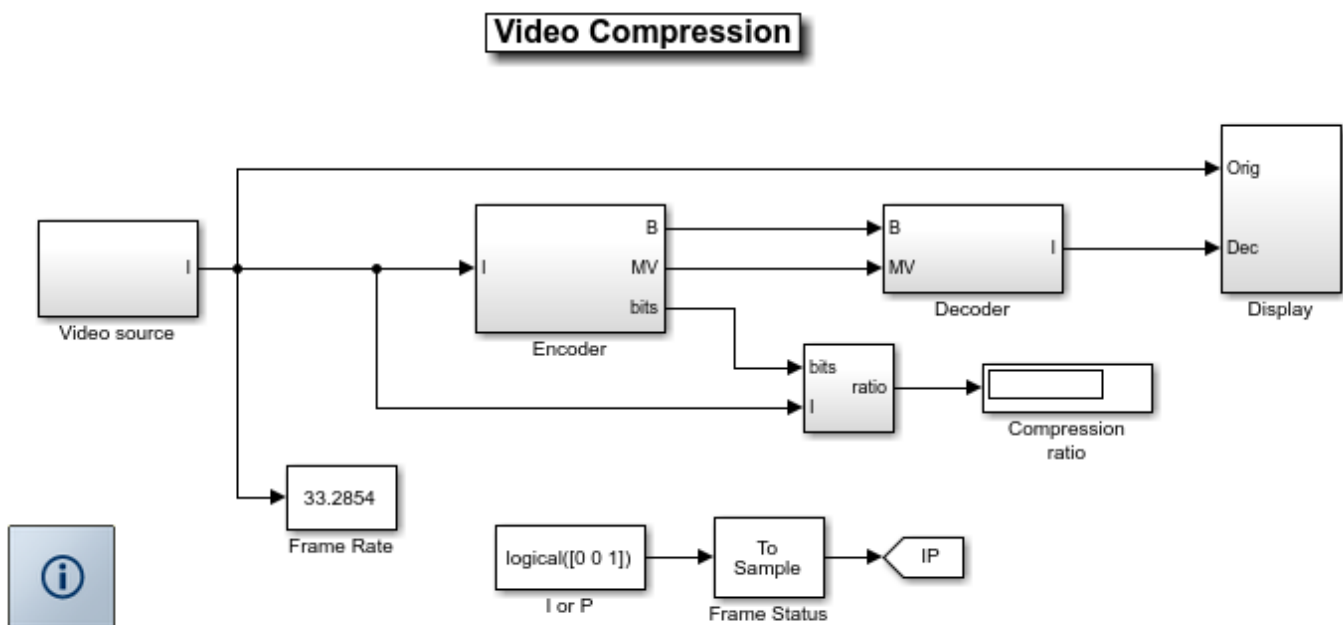


Video Compression

This example shows how to compress a video using motion compensation and discrete cosine transform (DCT) techniques. The example calculates motion vectors between successive frames and uses them to reduce redundant information. Then it divides each frame into submatrices and applies the discrete cosine transform to each submatrix. Finally, the example applies a quantization technique to achieve further compression. The Decoder subsystem performs the inverse process to recover the original video.

Example Model

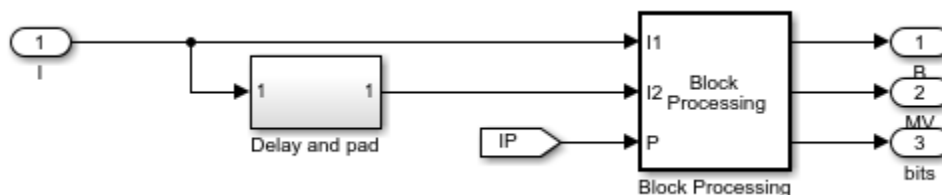
The following figure shows the Video Compression model:



Copyright 2005-2016 The MathWorks, Inc.

Encoder Subsystem

The Block Processing block sends 16-by-16 submatrices of each video frame to the Block Processing block's subsystem for processing. Within this subsystem, the model applies a motion compensation technique and the DCT to the video stream. By discarding many high-frequency coefficients in the DCT output, the example reduces the bit rate of the input video.



Video Compression Results

The Decoded window shows the compressed video stream. You can see that the compressed video is not as clear as the original video, shown in the Original window, but it still contains many of its features.



Available Example Versions

Intensity version of this example:

vipcodec.slx

Color version of this example:

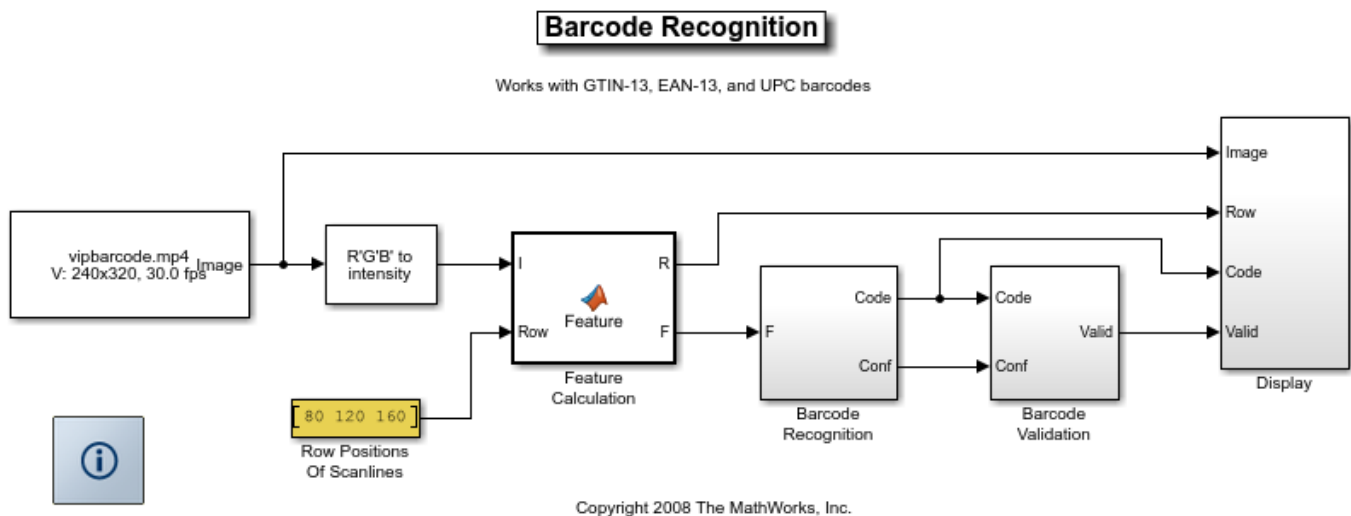
vipcodec_color.slx

Barcode Recognition

This example shows how to create an image processing system which can recognize and interpret a GTIN-13 barcode. The GTIN-13 barcode, formally known as EAN-13, is an international barcode standard. It is a superset of the widely used UPC standard.

Example Model

The following figure shows the Barcode Recognition model:



The GTIN-13 Barcode

GTIN is the acronym for Global Trade Item Number, a family of product identification numbers that encompasses the various versions of the EAN barcodes and provides a unified worldwide numbering system. The GTIN-13 (EAN/UCC-13) barcode encodes a 13-digit number.

Algorithm

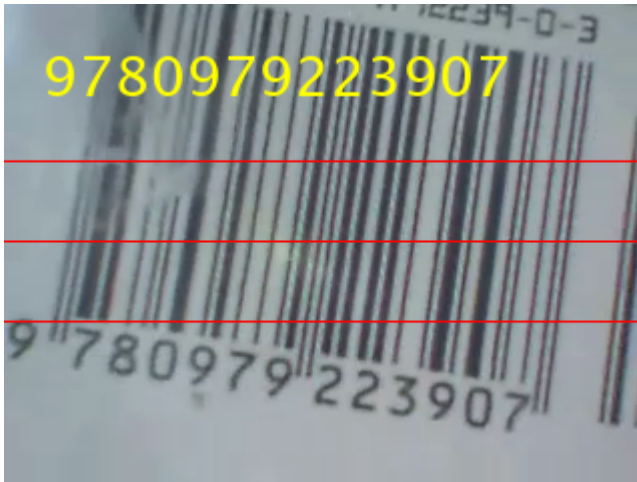
The barcode recognition example performs a search on selected rows of the input image, called scanlines. Prior to recognition, each pixel of the scanline is preprocessed by transforming it into a feature value. The feature value of a pixel is set to a 1, if the pixel is considered black, -1 if it is considered white, and a value between -1 and 1 otherwise. Once all pixels are transformed, the scanline sequences are analyzed. The example identifies the sequence and location of the guard patterns [1] and symbols. The symbols are upsampled and compared with the codebook to determine the corresponding code.

To compensate for various barcode orientations, the example analyzes from left-to-right and from right-to-left and chooses the better match. If the checksum is correct and a matching score against the codebook is higher than a set threshold, the code is considered valid and is displayed.

You can change the number and location of the scanlines by changing the value of the "Row Positions Of Scanlines" parameter.

Results

The scanlines that have been used to detect barcodes are displayed in red. When a GTIN-13 is correctly recognized and verified, the code is displayed at the top of the image.



Available Example Versions

Example using stored video data: vipbarcoderecognition.slx (platform independent)

Example using live video acquisition: viplivebarcoderecognition_win.slx (Windows® only)

References

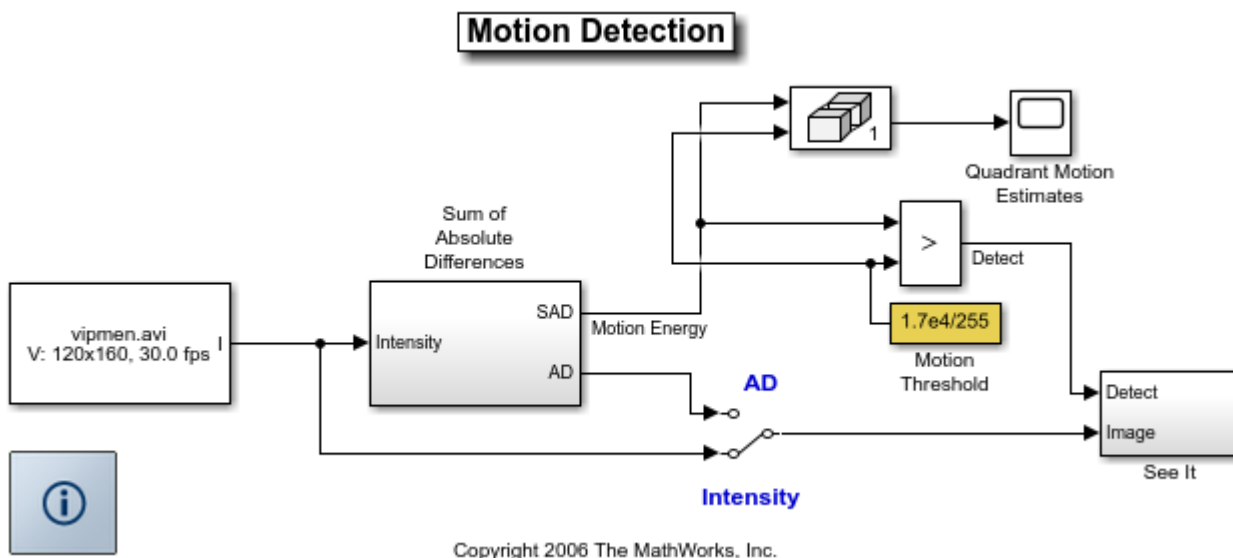
[1] T. Pavlidis, J. Swartz, and Y.P. Wang, *Fundamentals of bar code information theory*, Computer, pp. 74-86, vol. 23, no. 4, Apr 1990.

Motion Detection

This example shows how to use sum of absolute differences (SAD) method for detecting motion in a video sequence. This example applies SAD independently to four quadrants of a video sequence. If motion is detected in a quadrant, the example highlights the quadrant in red.

Example Model

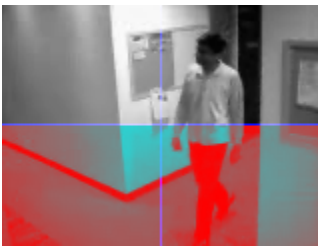
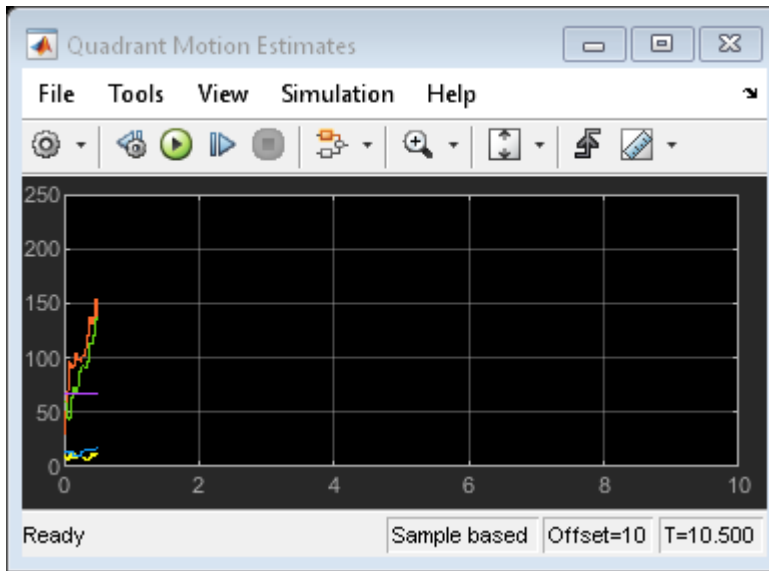
The following figure shows the Motion Detection example model:



Motion Detection Results

If you double-click the Switch block so that the signal is connected to the SAD side, the Video Viewer block displays the SAD values, which represent the absolute value of the difference between the current and previous image. When these SAD values exceed a threshold value, the example highlights the quadrant in red.

Note that the difference image itself may be viewed, in place of the original intensity image, along with the red motion highlighting, which indicates how the SAD metric works.



Pattern Matching

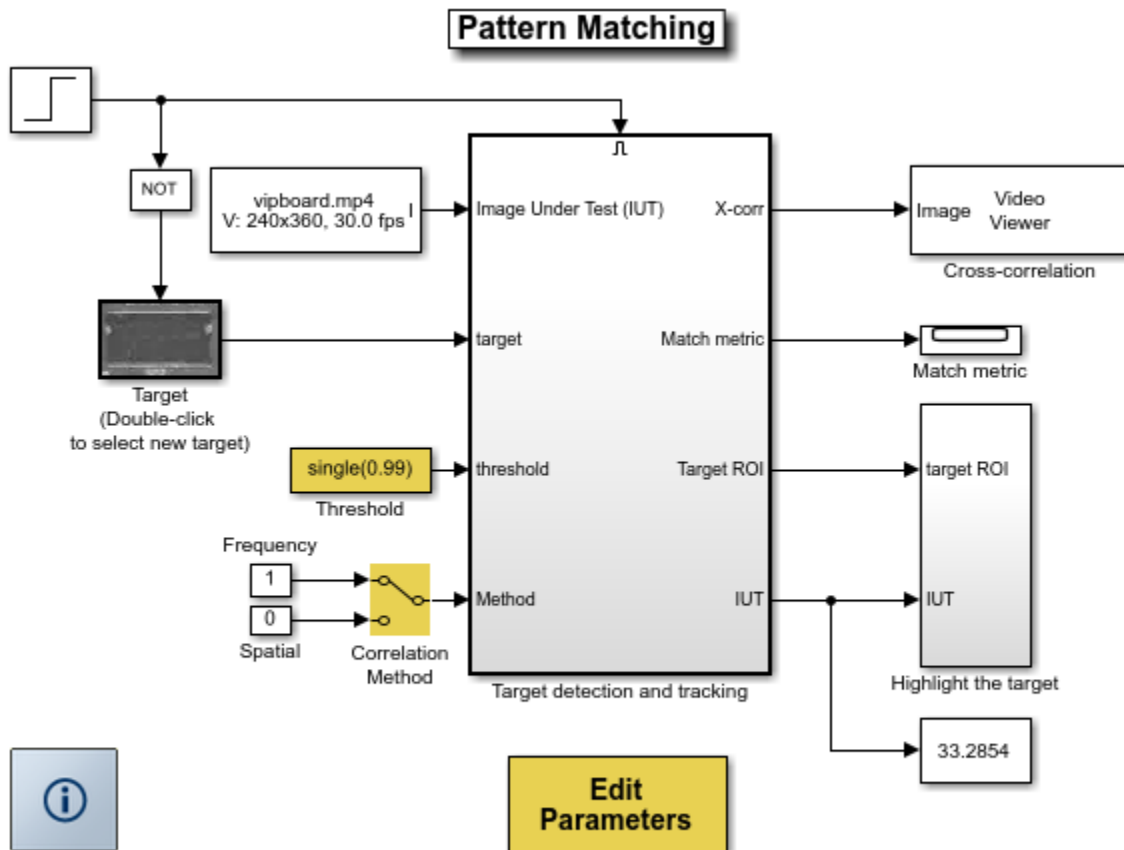
This example shows how to use the 2-D normalized cross-correlation for pattern matching and target tracking.

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

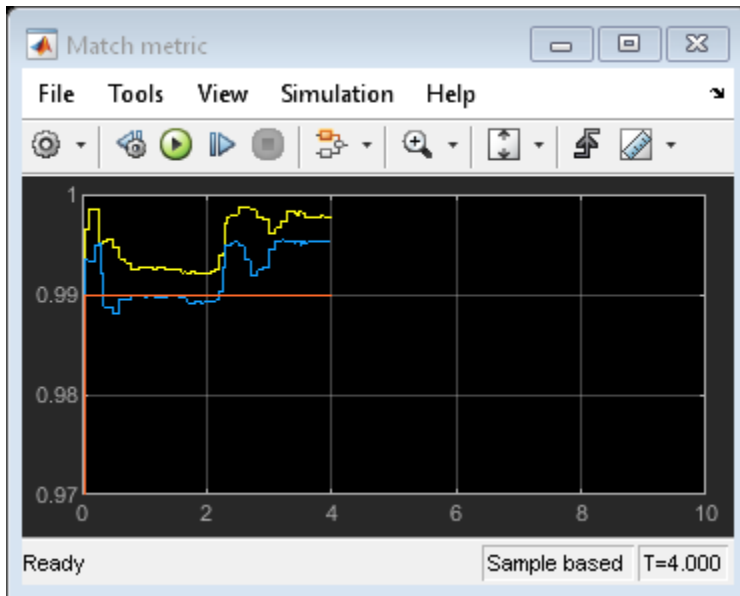
The following figure shows the Pattern Matching model:



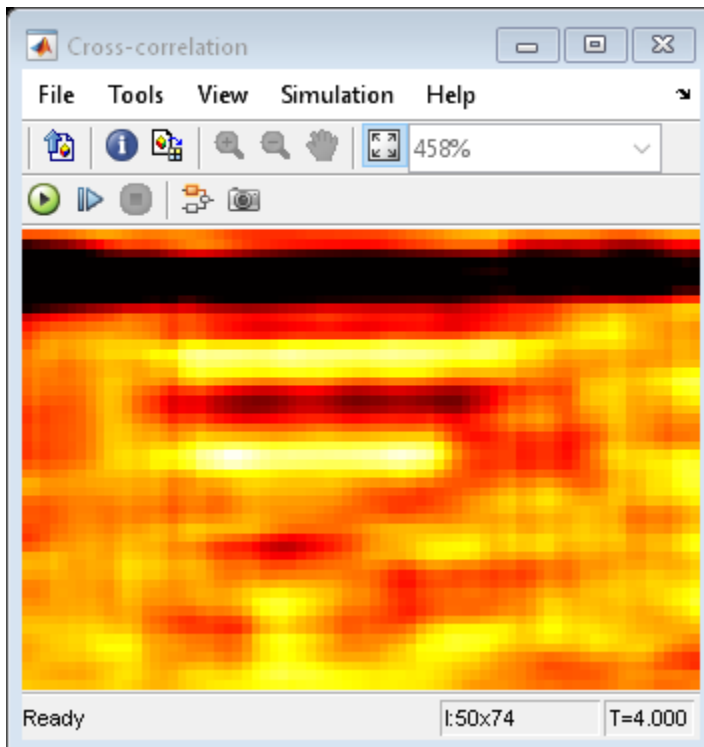
Copyright 2003-2008 The MathWorks, Inc.

Pattern Matching Results

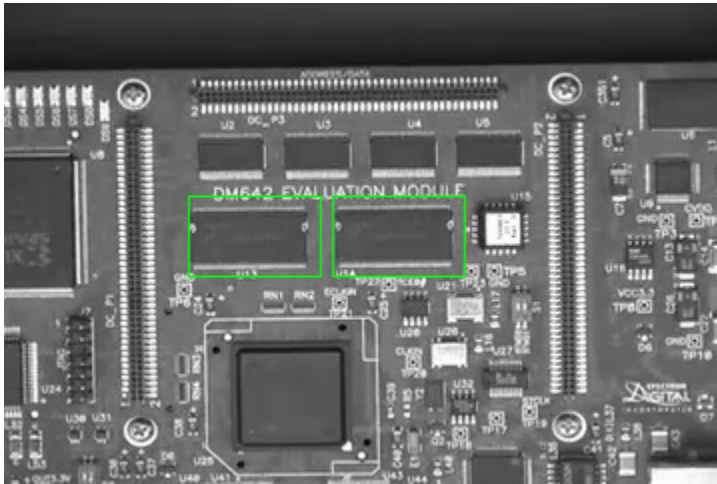
The Match metric window shows the variation of the target match metrics. The model determines that the target template is present in a video frame when the match metric exceeds a threshold (cyan line).



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.

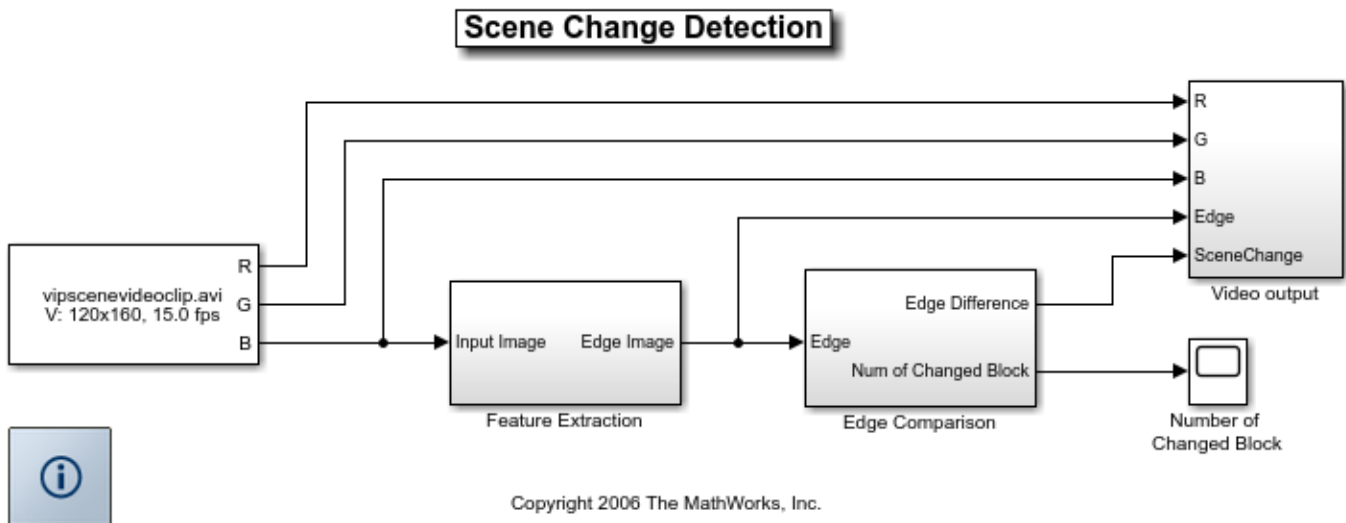


Scene Change Detection

This example shows how to segment video in time. The algorithm in this example can be used to detect major changes in video streams, such as when a commercial begins and ends. It can be useful when editing video or when you want to skip ahead through certain content.

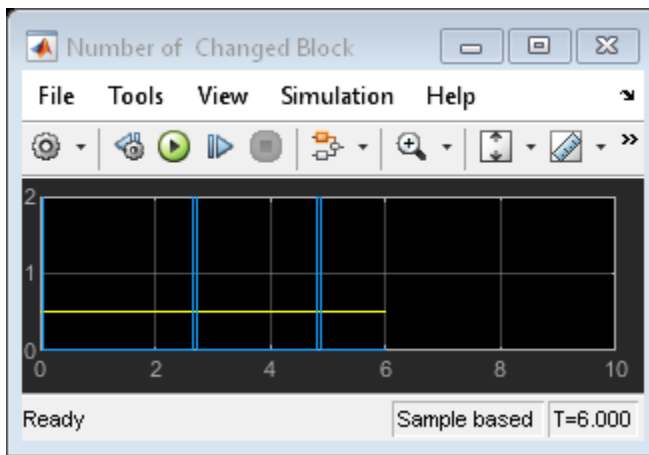
Example Model

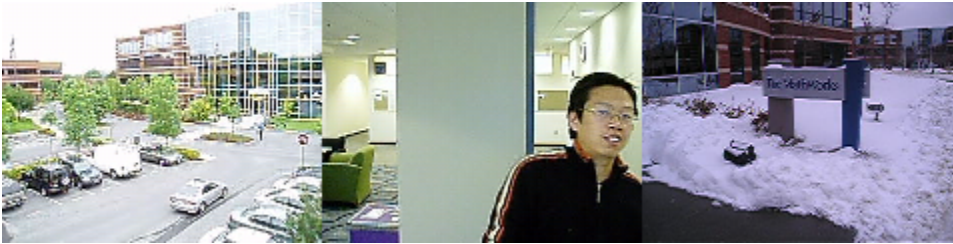
The following figure shows the Scene Change Detection example model:



Scene Change Detection Results

The model segments the video using the following steps. First, it finds the edges in two consecutive video frames, which makes the algorithm less sensitive to small changes. Based on these edges, the model uses the Block Processing block to compare sections of the video frames to one another. If the number of different sections exceeds a specified threshold, the example determines that the scene has changed.





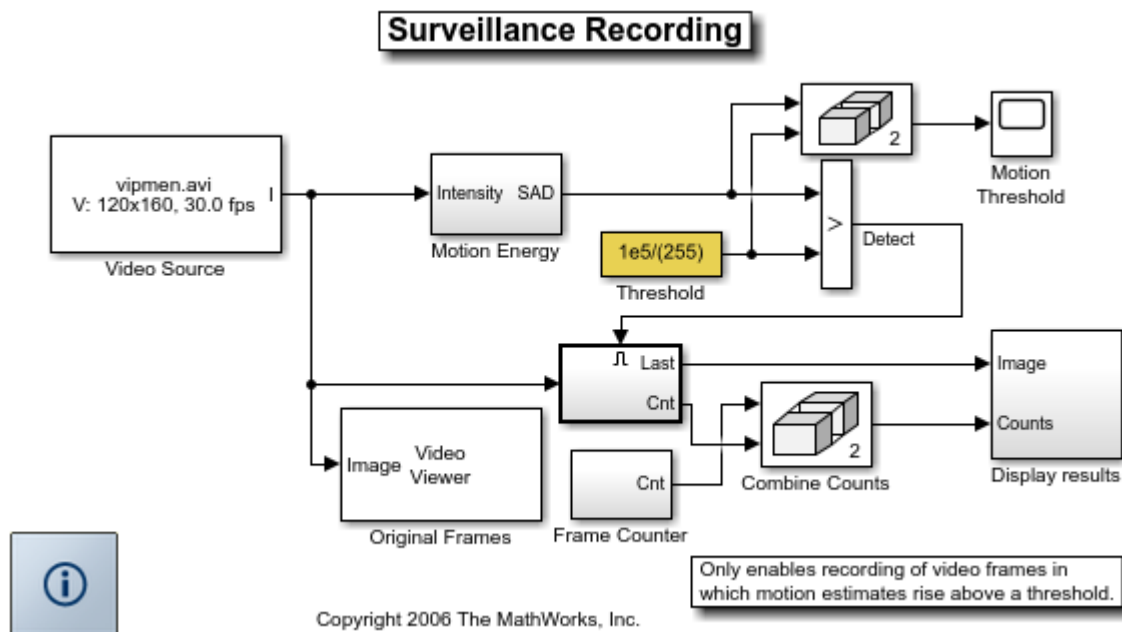
Surveillance Recording

This example shows how to process surveillance video to select frames that contain motion. Security concerns mandate continuous monitoring of important locations using video cameras. To efficiently record, review, and archive this massive amount of data, you can either reduce the video frame size or reduce the total number of video frames you record. This example illustrates the latter approach. In it, motion in the camera's field of view triggers the capture of "interesting" video frames.

Watch the Surveillance Recording example.

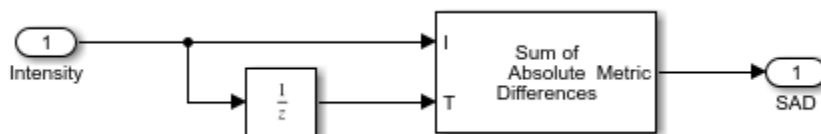
Example Model

The following figure shows the Surveillance Recording model:



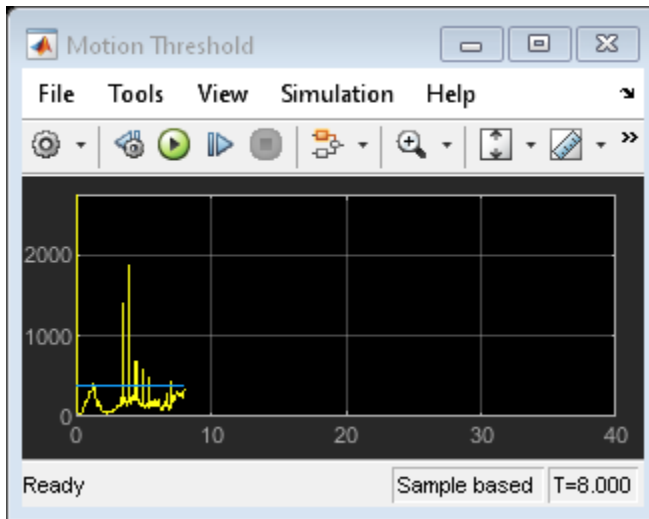
Motion Energy Subsystem

The example uses the Template Matching block to detect motion in the video sequence. When the Sum of Absolute Differences (SAD) value of a particular frame exceeds a threshold, the example records this video frame and displays it in the Motion Frames window.



Surveillance Recording Results

The Motion Threshold window displays the threshold value in blue, and plots the SAD values for each frame in yellow. Any time the SAD value exceeds the threshold, the model records the video frame.



The Original frames window shows a frame of the original video.



The Motion frames window shows the last recorded video frame. In this window, the Source frame value steadily increases as the video runs and the Captured frame value indicates the total number of frames recorded by the model.



Available Example Versions

Floating-point: vipsurveillance.slx

Fixed-point: vipsurveillance_fixpt.slx

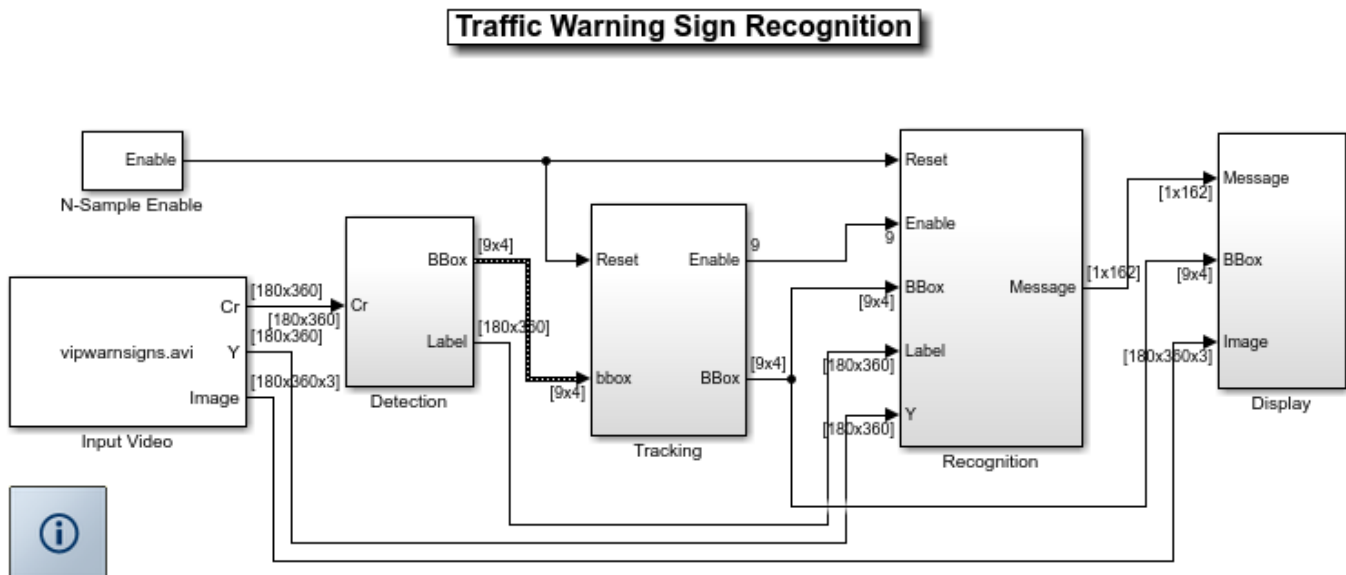
Traffic Warning Sign Recognition

This example shows how to recognize traffic warning signs, such as Stop, Do Not Enter, and Yield, in a color video sequence.

Watch the Traffic Warning Sign Recognition example.

Example Model

The following figure shows the Traffic Warning Sign Recognition model:



Copyright 2007-2010 The MathWorks, Inc.

Traffic Warning Sign Templates

The example uses two set of templates - one for detection and the other for recognition.

To save computation, the detection templates are low resolution, and the example uses one detection template per sign. Also, because the red pixels are the distinguishing feature of the traffic warning signs, the example uses these pixels in the detection step.

For the recognition step, accuracy is the highest priority. So, the example uses three high resolution templates for each sign. Each of these templates shows the sign in a slightly different orientation. Also, because the white pixels are the key to recognizing each traffic warning sign, the example uses these pixels in the recognition step.

The Detection Templates window shows the traffic warning sign detection templates.



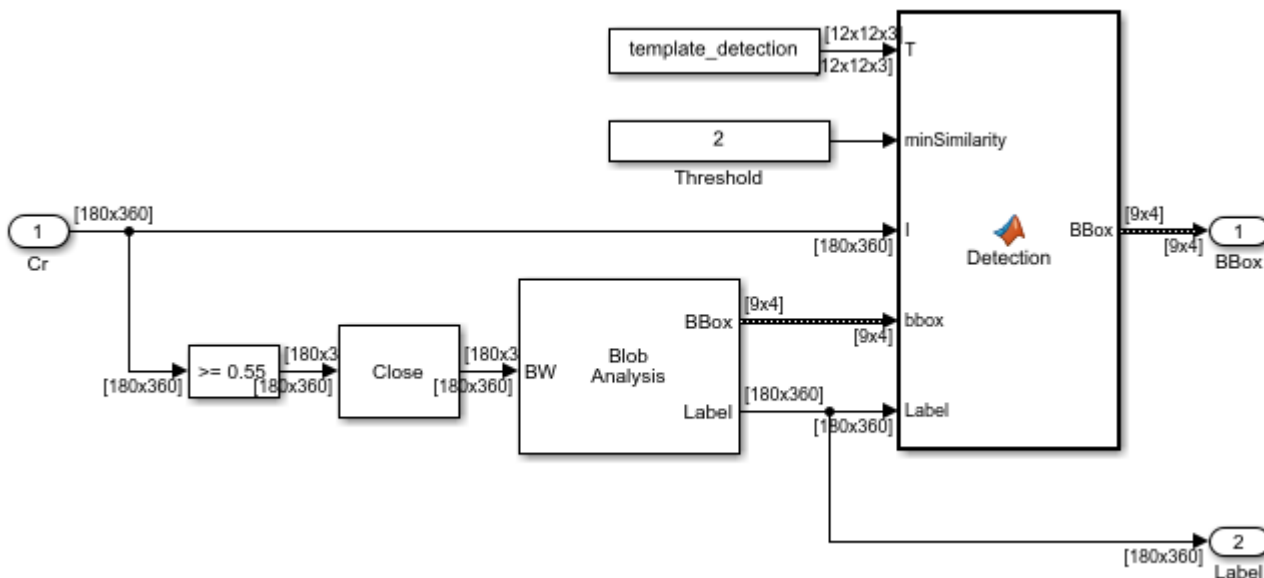
The Recognition Templates window shows the traffic warning sign recognition templates.



The templates were generated using `vipwarningsigns_templates.m` and were stored in `vipwarningsigns_templates.mat`.

Detection

The example analyzes each video frame in the YCbCr color space. By thresholding and performing morphological operations on the Cr channel, the example extracts the portions of the video frame that contain blobs of red pixels. Using the Blob Analysis block, the example finds the pixels and bounding box for each blob. The example then compares the blob with each warning sign detection template. If a blob is similar to any of the traffic warning sign detection templates, it is a potential traffic warning sign.



Tracking and Recognition

The example compares the bounding boxes of the potential traffic warning signs in the current video frame with those in the previous frame. Then the example counts the number of appearances of each potential traffic warning sign.

If a potential sign is detected in 4 contiguous video frames, the example compares it to the traffic warning sign recognition templates. If the potential traffic warning sign is similar enough to a traffic warning sign recognition template in 3 contiguous frames, the example considers the potential traffic warning sign to be an actual traffic warning sign.

When the example has recognized a sign, it continues to track it. However, to save computation, it no longer continues to recognize it.

Display

After a potential sign has been detected in 4 or more video frames, the example uses the Draw Shape block to draw a yellow rectangle around it. When a sign has been recognized, the example uses the

Insert Text block to write the name of the sign on the video stream. The example uses the term 'Tag' to indicate the order in which the sign is detected.

Traffic Warning Sign Recognition Results



Abandoned Object Detection

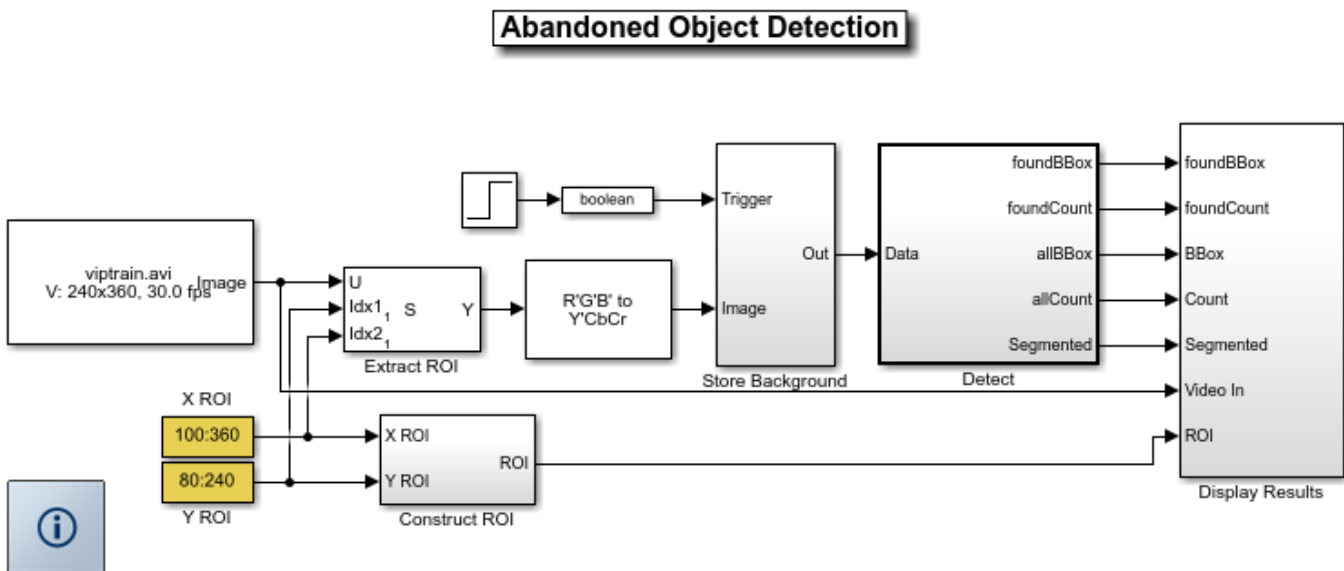
This example shows how to track objects at a train station and to determine which ones remain stationary. Abandoned objects in public areas concern authorities since they might pose a security risk. Algorithms, such as the one used in this example, can be used to assist security officers monitoring live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

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

Watch the Abandoned Object Detection example.

Example Model

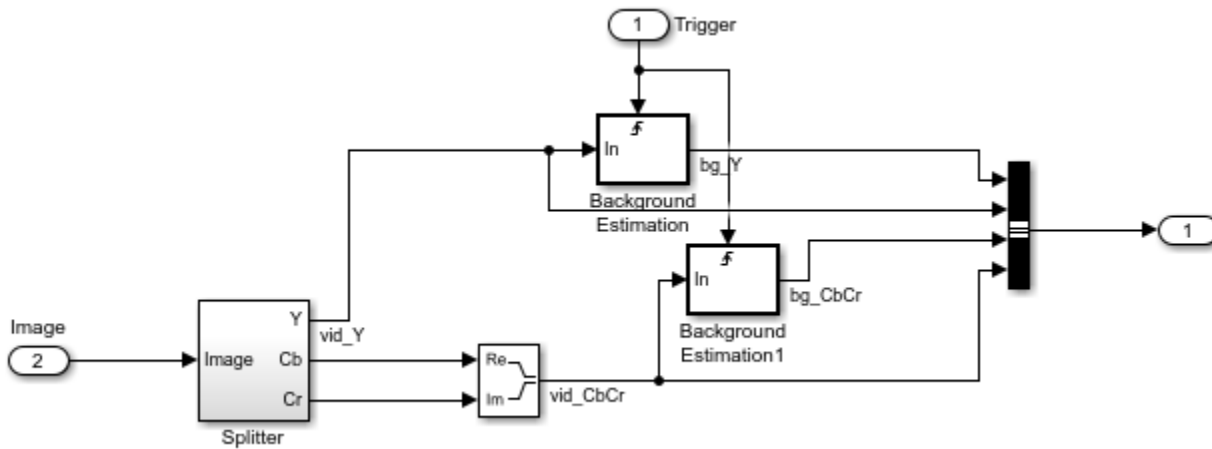
The following figure shows the Abandoned Object Detection example model.



Store Background Subsystem

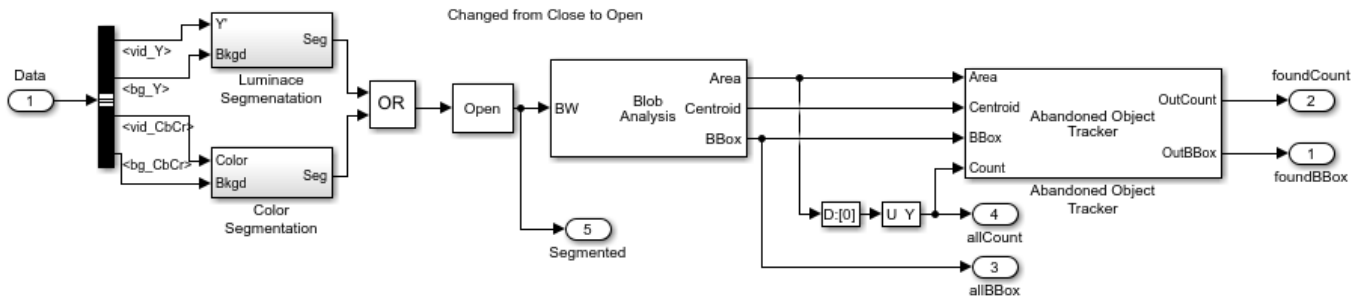
This example uses the first frame of the video as the background. To improve accuracy, the example uses both intensity and color information for the background subtraction operation. During this operation, Cb and Cr color channels are stored in a complex array.

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

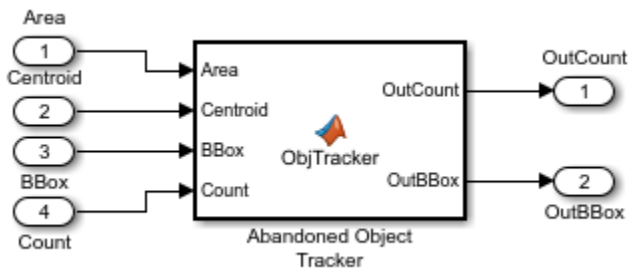


Detect Subsystem

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



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



Abandoned Object Detection Results

The All Objects window marks the region of interest (ROI) with a yellow box and all detected objects with green boxes.



The Threshold window shows the result of the background subtraction in the ROI.



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

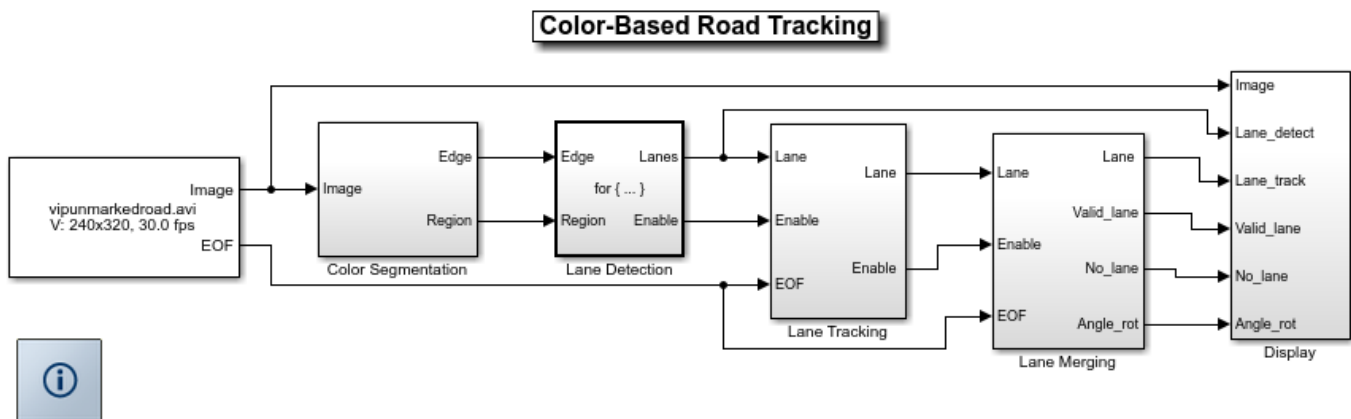


Color-based Road Tracking

This example shows how to use color information to detect and track road edges set in primarily residential settings where lane markings may not be present. The Color-based Tracking example illustrates how to use the Color Space Conversion block, the Hough Transform block, and the Kalman Filter block to detect and track information using hue and saturation.

Example Model

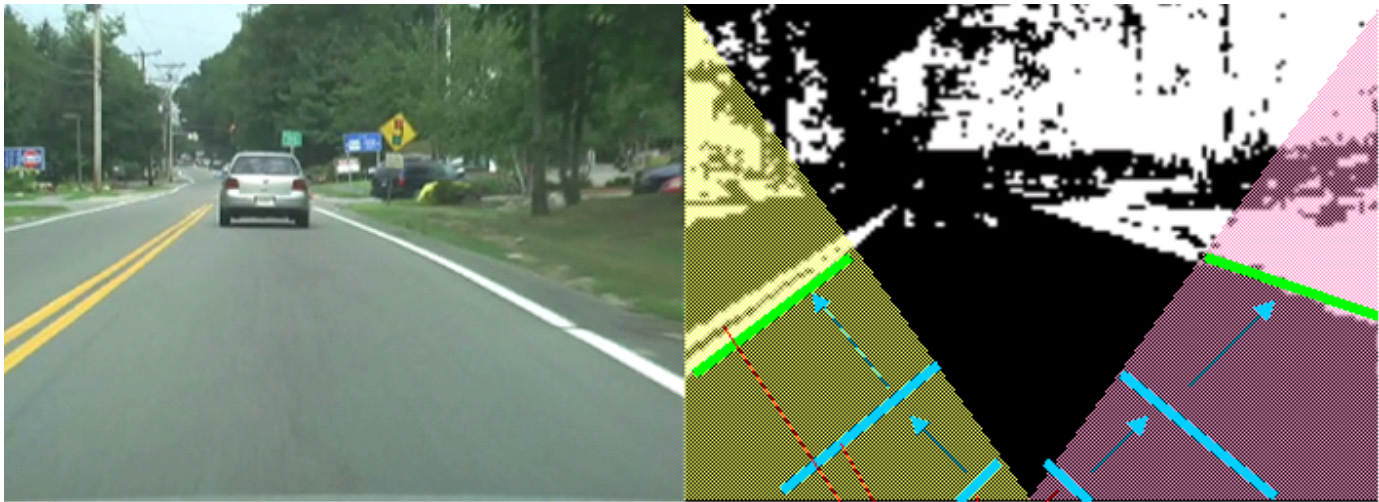
The following figure shows the Color-based Road Tracking model:



Copyright 2007-2010 The MathWorks, Inc.

Algorithm

The example algorithm performs a search to define the left and right edges of a road by analyzing video images for change in color behavior. First a search for edge pixels, or a line passing through enough number of color pixels, whichever comes first, is initiated from the bottom center of the image. The search moves to both the upper left and right corners of the image.

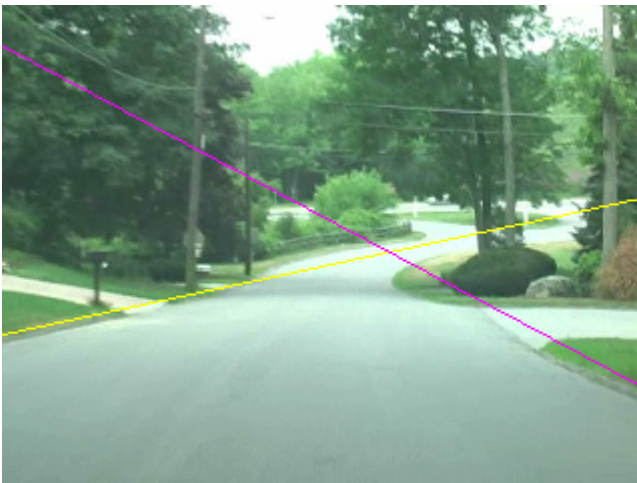


1. Initial line pair
2. Expanding ...
3. Final, convergent

To process low quality video sequences, where road sides might be difficult to see, or are obstructed, the algorithm will wait for multiple frames of valid edge information. The example uses the same process to decide when to begin to ignore a side.

Tracking Results

The Detection window shows the road sides detected in the current video frame.



When no road sides are visible, the Tracking window displays an error symbol.



When only one side of the road is visible, the example displays an arrow parallel to the road side. The direction of the arrow is toward the upper point of intersection between the road side and image boundary.



When both of the road sides are visible, the example shows an arrow in the center of the road in the direction calculated by averaging the directions of the left and right sides.



Detect and Track Face

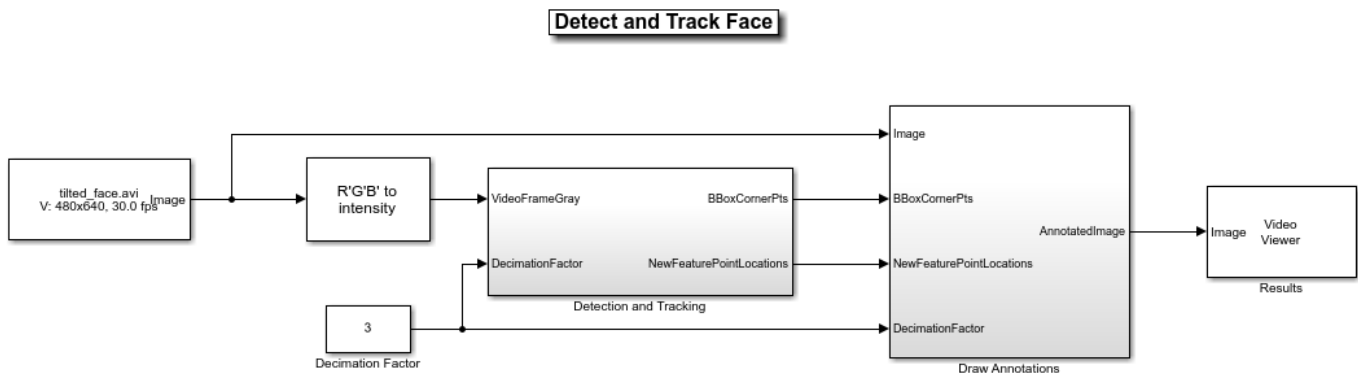
This example shows how to implement a face detection and tracking algorithm in Simulink® by using a MATLAB® Function block. It closely follows the “Face Detection and Tracking Using the KLT Algorithm” on page 7-20 MATLAB® example.

Introduction

Object detection and tracking are important in many computer vision applications, including activity recognition, automotive safety, and surveillance. In this example, you design a system in Simulink® to detect a face in a video frame, identify the facial features and track these features. The output video frame contains the detected face and the features tracked. If a face is not visible or goes out of focus, the system tries to re-acquire the face and then perform the tracking. This example is designed to detect and track a single face.

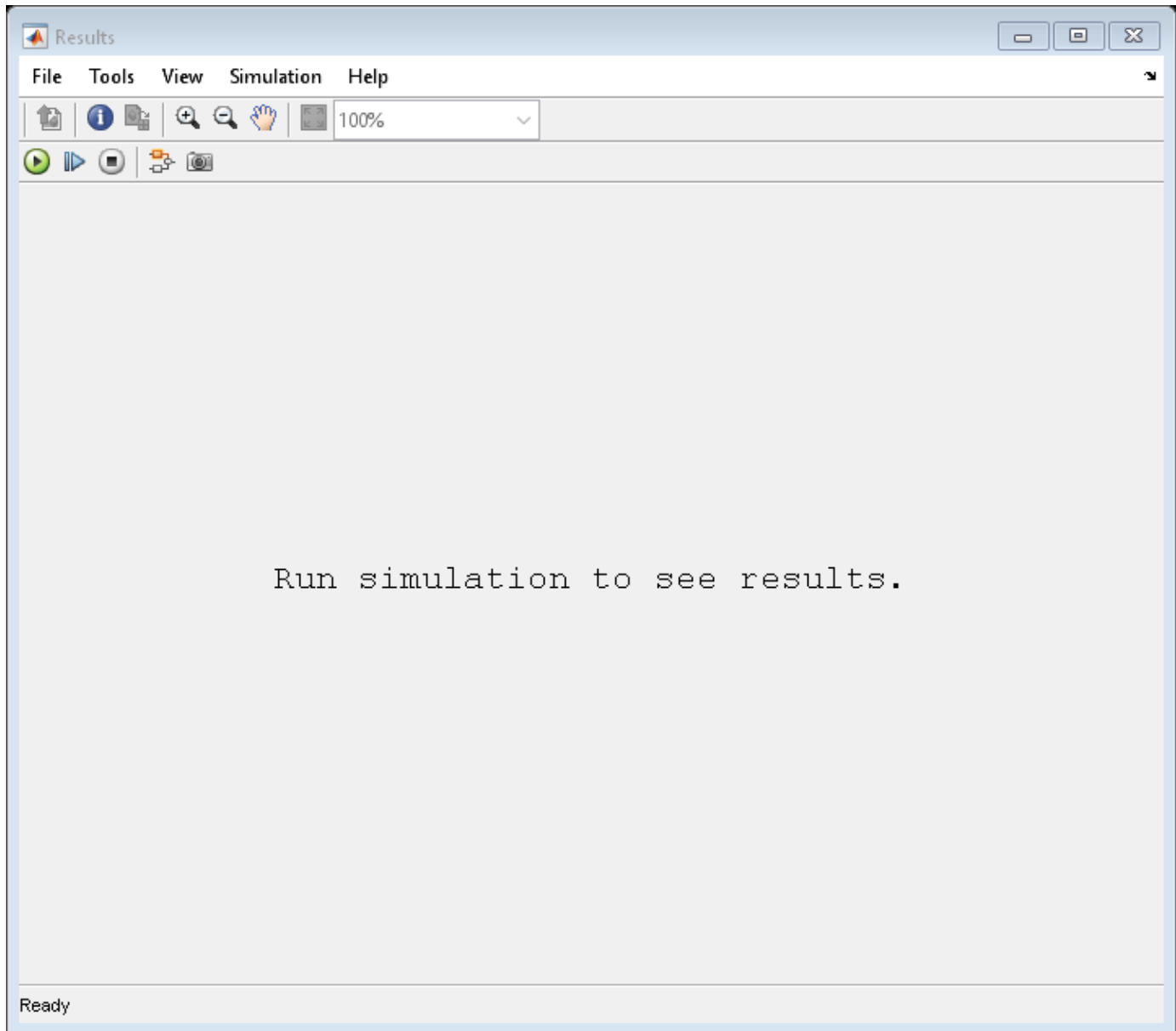
Example Model

```
close
open_system('DetectAndTrackFace');
```



Copyright 2016-2018 The MathWorks, Inc.





Setup

This example uses the From Multimedia File block to read the video frames from the video file. The Detection and Tracking subsystem takes in a video frame and provides a bounding box for the face and feature points within the bounding box as its output to the Draw Annotations subsystem. This subsystem inserts in the image a rectangle for the bounding box and markers for the feature points.

Detection and Tracking

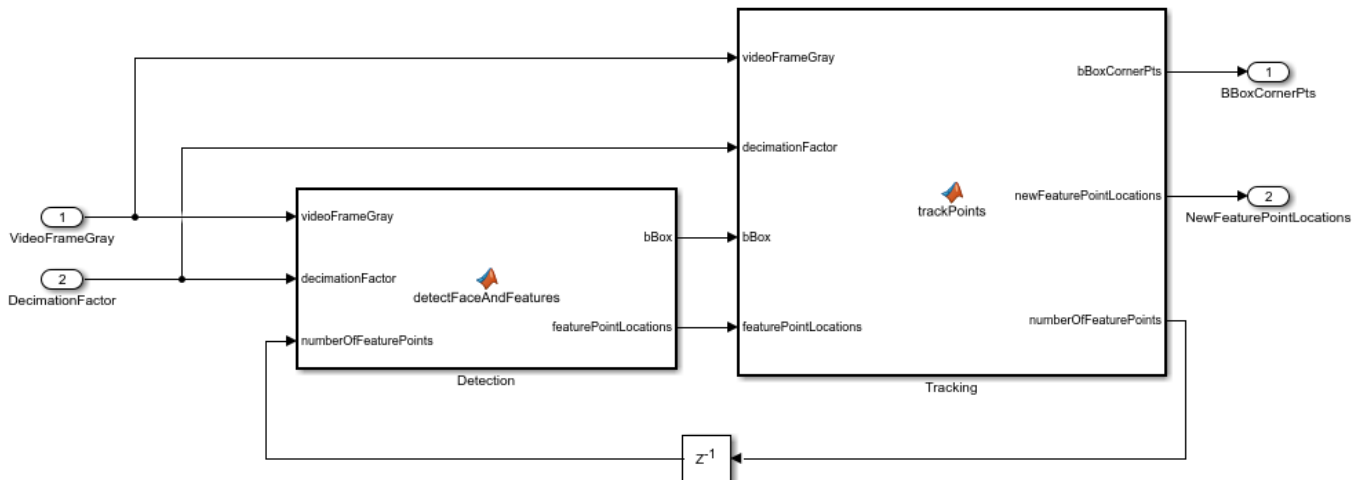
In this example, the `vision.CascadeObjectDetector` System object™ detects the location of the face in the current video frame. The cascade object detector uses the Viola-Jones detection algorithm and a trained classification model for detection. After the face is detected, facial feature points are identified using the "Good Features to Track" method proposed by Shi and Tomasi.

Then the `vision.PointTracker` System object™ tracks the identified feature points by using the Kanade-Lucas-Tomasi (KLT) feature-tracking algorithm. For each point in the previous frame, the point tracker attempts to find the corresponding point in the current frame. Then the `estimateGeometricTransform` function estimates the translation, rotation, and scale between the old points and the new points. This transformation is applied to the bounding box around the face.

Although it is possible to use the cascade object detector on every frame, it is computationally expensive. This technique can also fail to detect the face, such as when the subject turns or tilts his head. This limitation comes from the type of trained classification model used for detection. In this example, you detect the face once, and then the KLT algorithm tracks the face across the video frames. The detection is performed again only when the face is no longer visible or when the tracker cannot find enough feature points.

The ability to perform “Dynamic memory allocation in MATLAB functions” (Simulink) allows the usage of the previously mentioned System objects and methods inside the MATLAB® Function block.

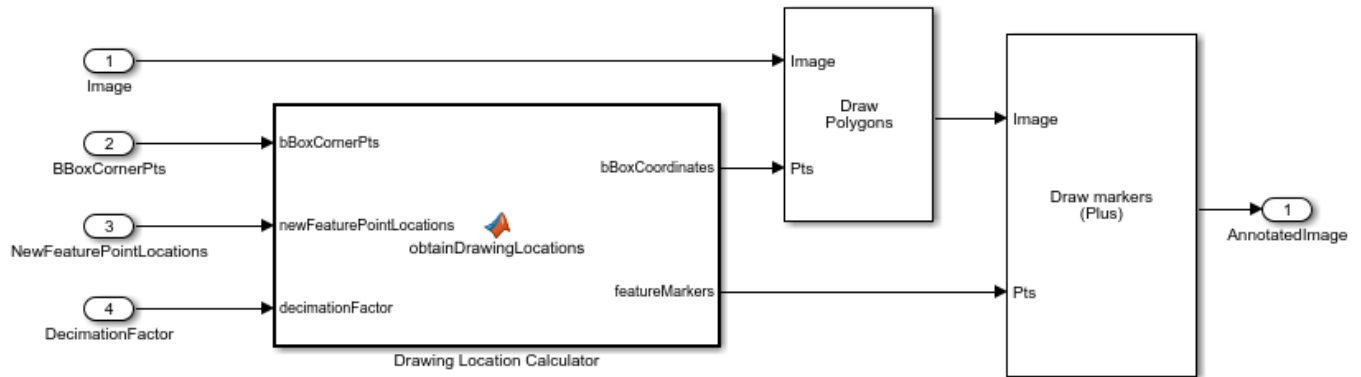
```
open_system('DetectAndTrackFace/Detection and Tracking')
```



Draw Annotations

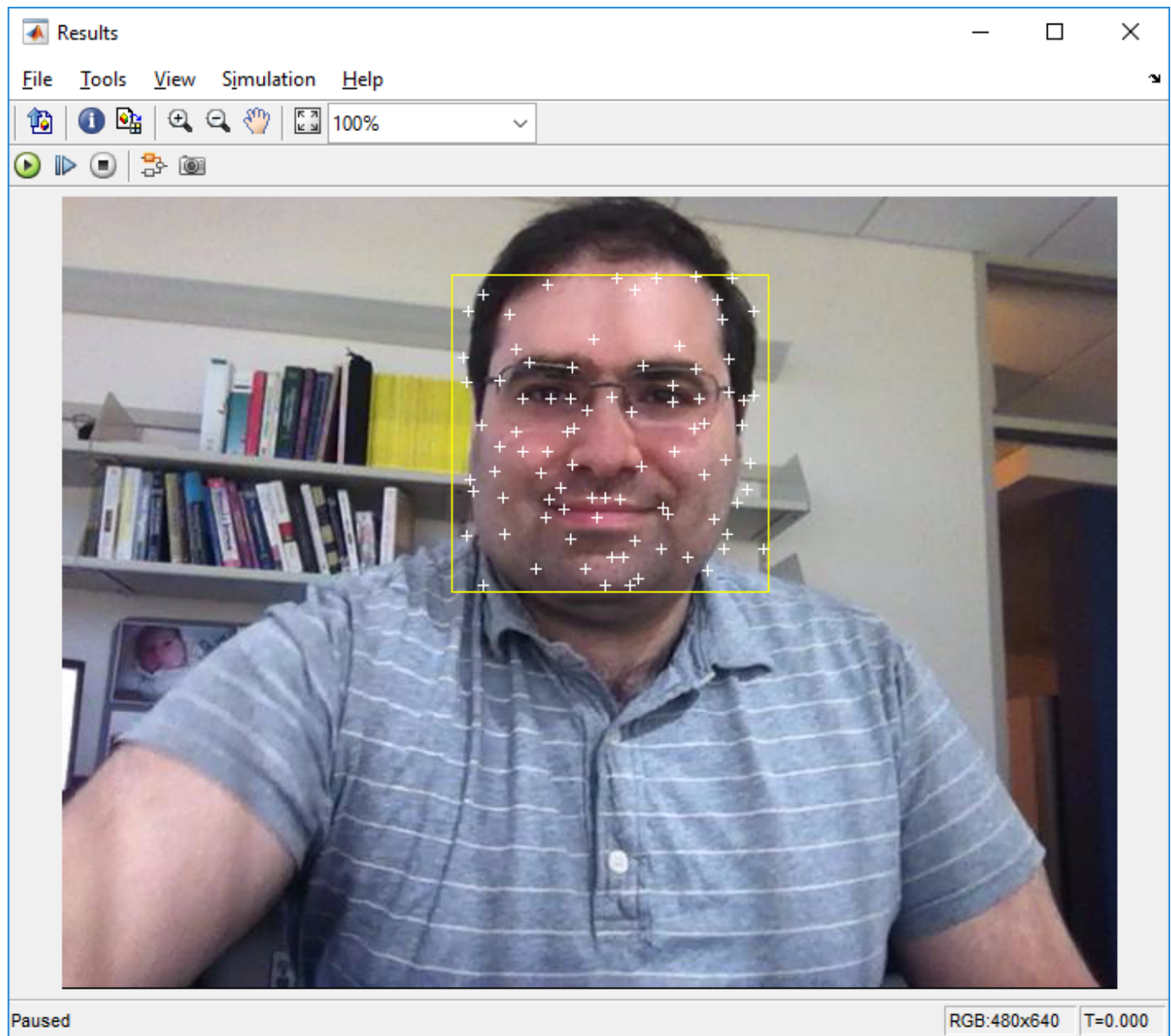
The bounding box corner points and feature point locations are used to draw on the output video frame. The Draw Shapes block draws the bounding box. The feature points are drawn using the Draw Markers block.

```
open_system('DetectAndTrackFace/Draw Annotations')
```

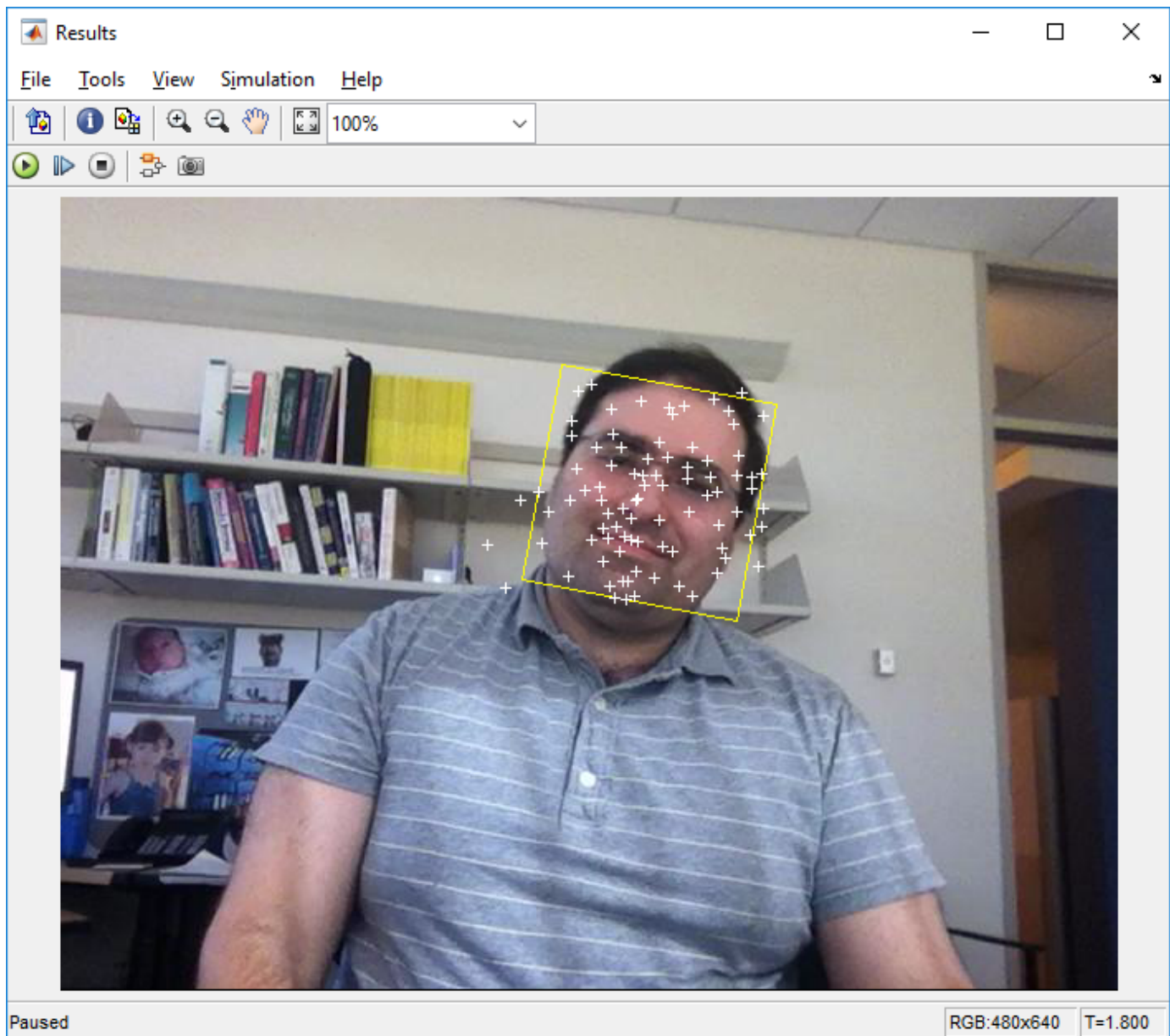


Results

The following display shows the detected face with the feature points.



The following display shows the tracked face and feature points.



References

Viola, Paul A., and Michael J. Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features", *IEEE CVPR*, 2001.

Lucas, Bruce D., and Takeo Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision." *International Joint Conference on Artificial Intelligence*, 1981.

Lucas, Bruce D., and Takeo Kanade. "Detection and Tracking of Point Features." *Carnegie Mellon University Technical Report CMU-CS-91-132*, 1991.

Shi, Jianbo, and Carlo Tomasi. "Good Features to Track." *IEEE Conference on Computer Vision and Pattern Recognition*, 1994.

ZKhalil, Zdenek, Krystian Mikolajczyk, and Jiri Matas. "Forward-Backward Error: Automatic Detection of Tracking Failures." *International Conference on Pattern Recognition*, 2010

Lane Departure Warning System

This example shows how to detect and track road lane markers in a video sequence and notifies the driver if they are moving across a lane. The example illustrates how to use the Hough Transform, Hough Lines and Kalman Filter blocks to create a line detection and tracking algorithm. The example implements this algorithm using the following steps: 1) Detect lane markers in the current video frame. 2) Match the current lane markers with those detected in the previous video frame. 3) Find the left and right lane markers. 4) Issue a warning message if the vehicle moves across either of the lane markers.

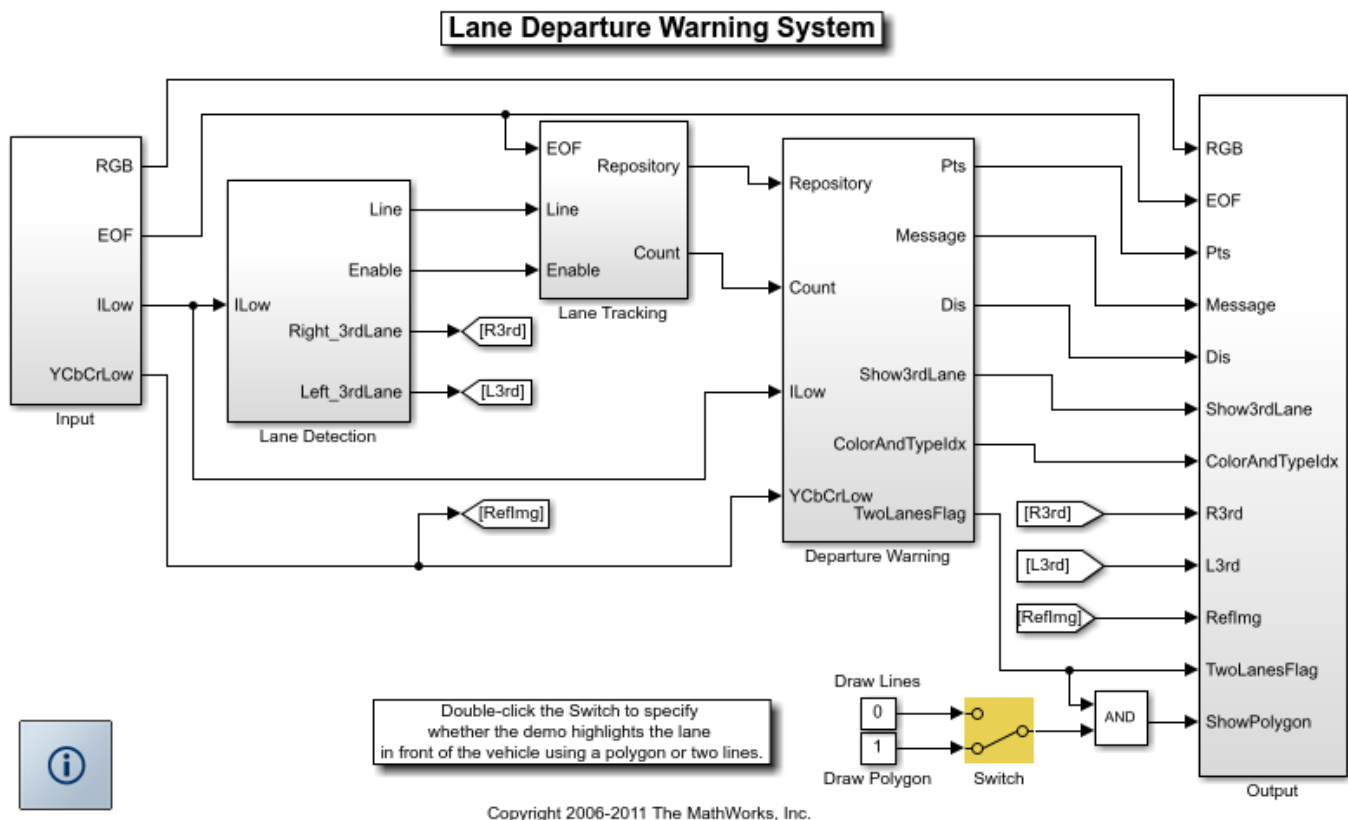
To process low quality video sequences, where lane markers might be difficult to see or are hidden behind objects, the example waits for a lane marker to appear in multiple frames before it considers the marker to be valid. The example uses the same process to decide when to begin to ignore a lane marker.

Note: The example parameters are defined in the model workspace. To access the parameters, click View > Model Explorer. Then navigate to Model Workspace under model's name.

Watch the Lane Departure Warning System example.

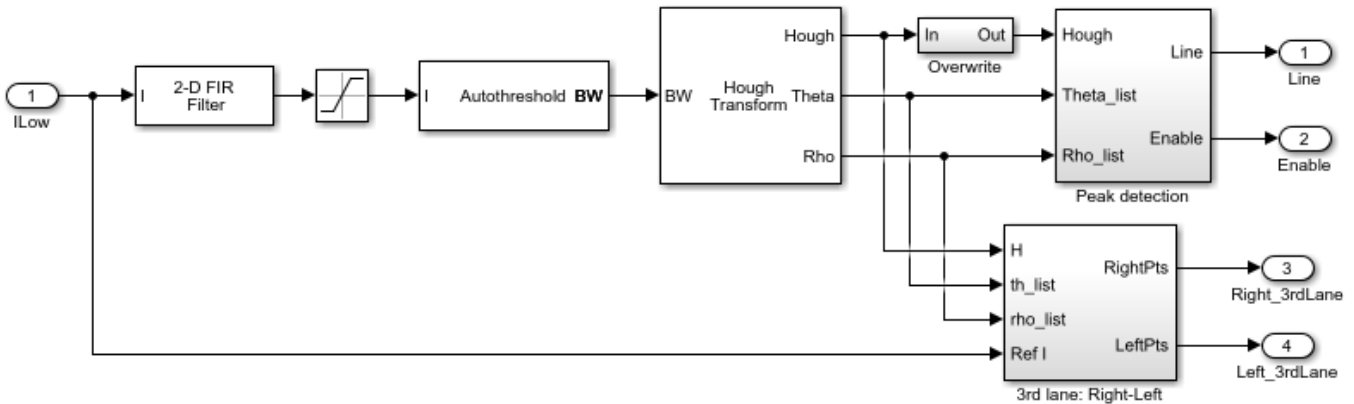
Example Model

The following figure shows the Lane Departure Warning System example model:



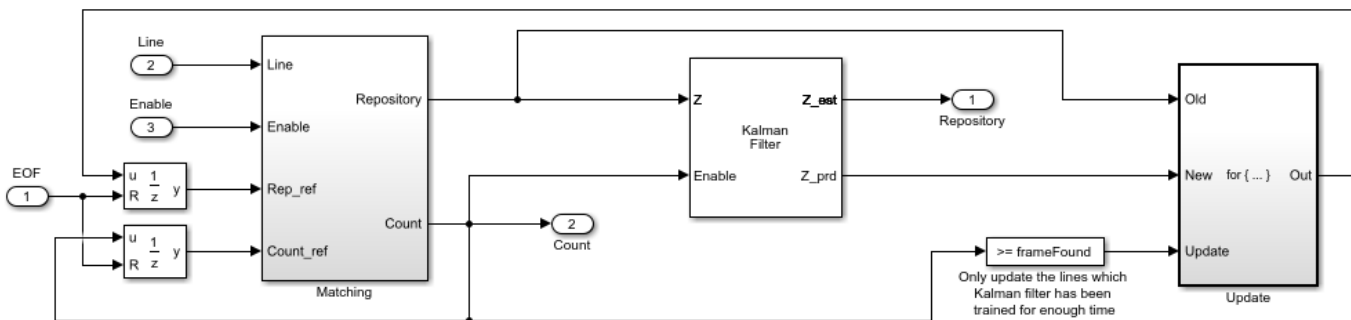
Lane Detection Subsystem

This subsystem uses the 2-D FIR Filter and Autothreshold blocks to detect the left boundaries of the lane markers in the current video frame. The boundaries of the lane markers resemble straight lines and correspond to peak values in the Hough transform matrix. This subsystem uses the Find Local Maxima block to determine the Polar coordinate location of the lane markers.



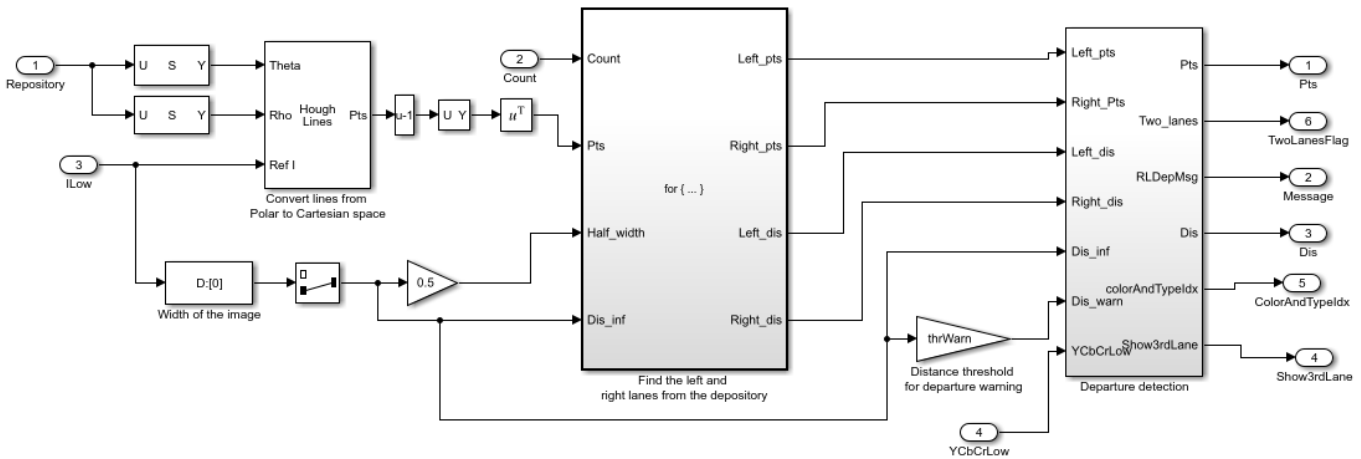
Lane Tracking Subsystem

The example saves the previously-detected lanes in a repository and counts the number of times each lane is detected. This subsystem matches the lanes found in the current video frame with those in the repository. If a current lane is similar enough to another lane in the repository, the example updates the repository with the lanes' current location. The Kalman Filter block predicts the location of each lane in the repository, which improves the accuracy of the lane tracking.



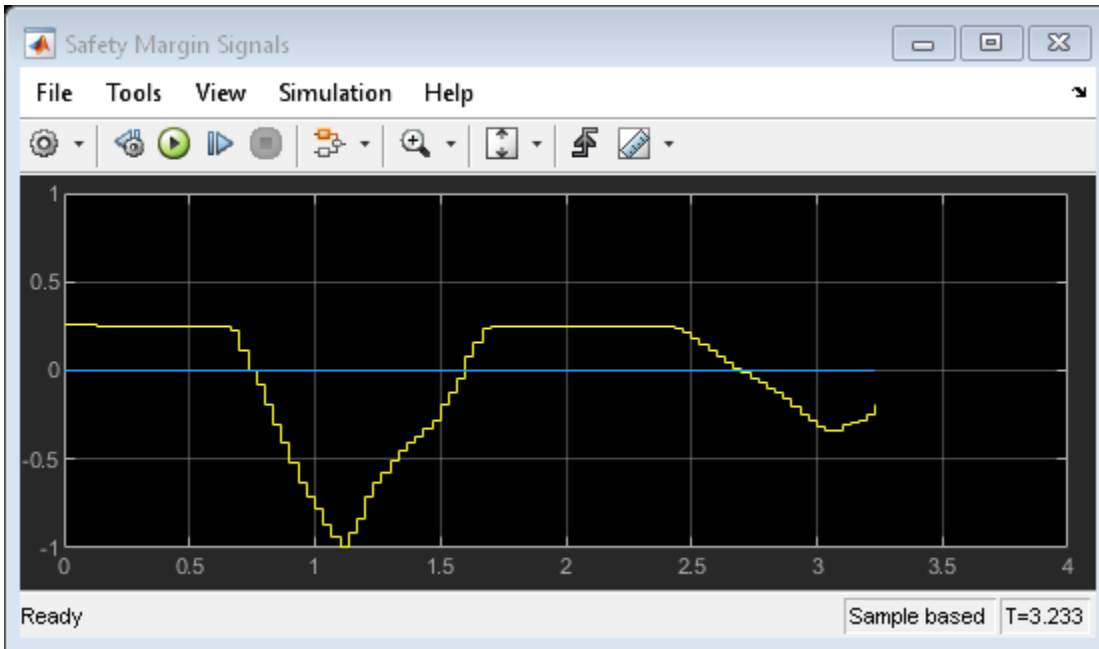
Departure Warning Subsystem

This subsystem uses the Hough Lines block to convert the Polar coordinates of a line to Cartesian coordinates. The subsystem uses these Cartesian coordinates to calculate the distance between the lane markers and the center of the video bottom boundary. If this distance is less than the threshold value, the example issues a warning. This subsystem also determines if the line is yellow or white and whether it is solid or broken.



Lane Departure Warning System Results

The Safety Margin Signals window shows a plot of a safety margin metric. The safety margin metric is determined by the distance between the car and the closest lane marker. When the safety margin metric, shown in yellow, drops below 0, shown in blue, the car is in lane departure mode otherwise the car is in normal driving mode.



The Results window shows the left and right lane markers and a warning message. The warning message indicates that the vehicle is moving across the right lane marker. The type and color of the lane markers are also shown in this window. In addition to the text message, the Windows® version of the example issues an audio warning.



Available Example Versions

Floating-point version of this example: `vipldws.slx`

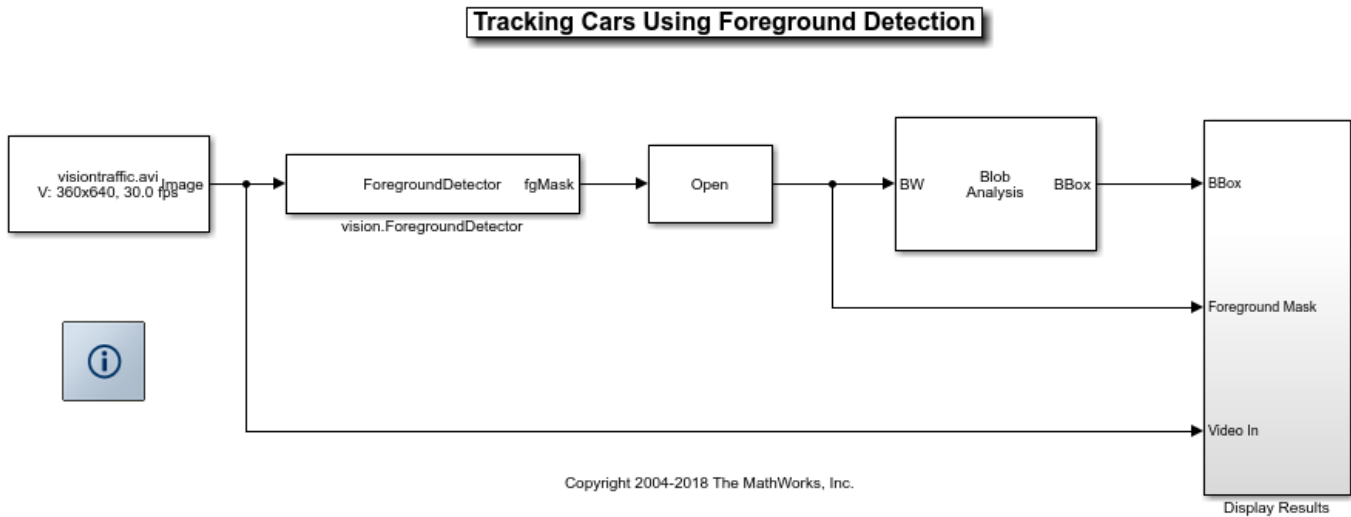
Fixed-point version of this example: `vipldws_fixpt.slx`

Tracking Cars Using Foreground Detection

This example shows how to detect and count cars in a video sequence using Gaussian mixture models (GMMs).

Example Model

The following figure shows the Tracking Cars Using Foreground Detection model:



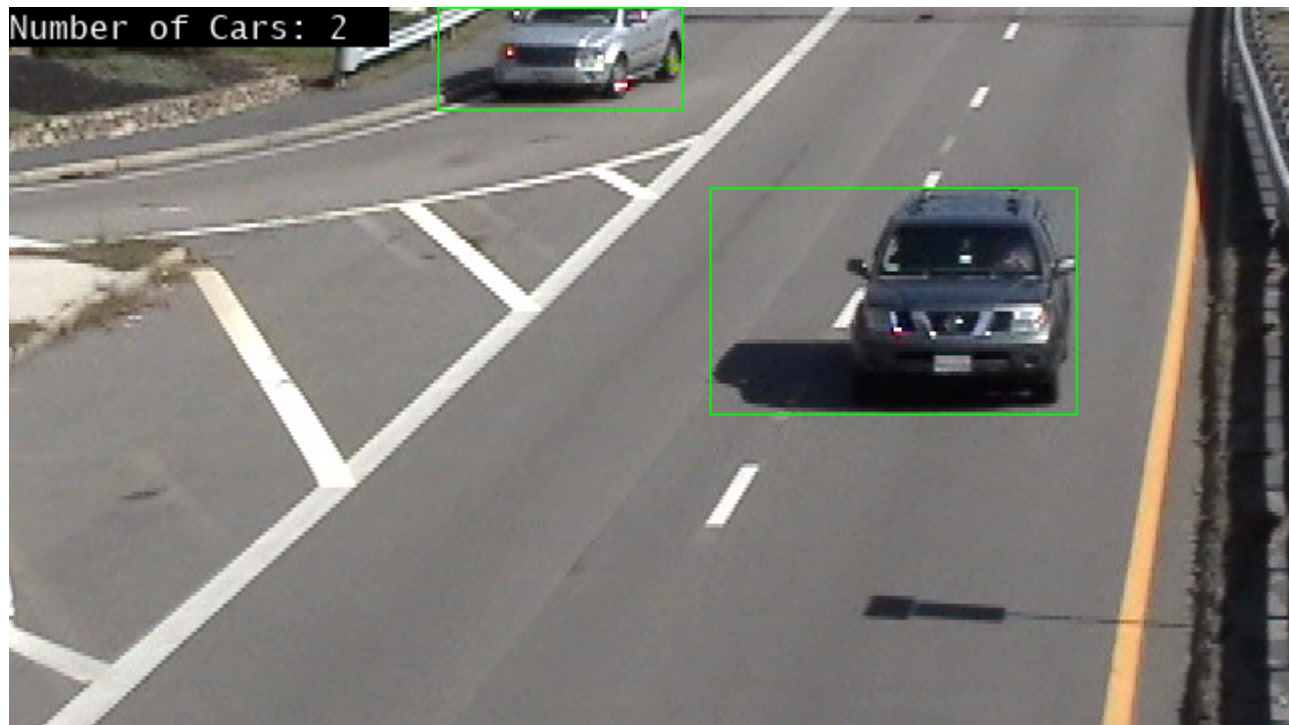
Detection and Tracking Results

Detecting and counting cars can be used to analyze traffic patterns. Detection is also a first step prior to performing more sophisticated tasks such as tracking or categorization of vehicles by their type.

This example uses the `vision.ForegroundDetector` to estimate the foreground pixels of the video sequence captured from a stationary camera. The `vision.ForegroundDetector` estimates the background using Gaussian Mixture Models and produces a foreground mask highlighting foreground objects; in this case, moving cars.

The foreground mask is then analyzed using the Blob Analysis block, which produces bounding boxes around the cars. Finally, the number of cars and the bounding boxes are drawn into the original video to display the final results.

Tracking Results



Prototype on a Xilinx Zynq Board

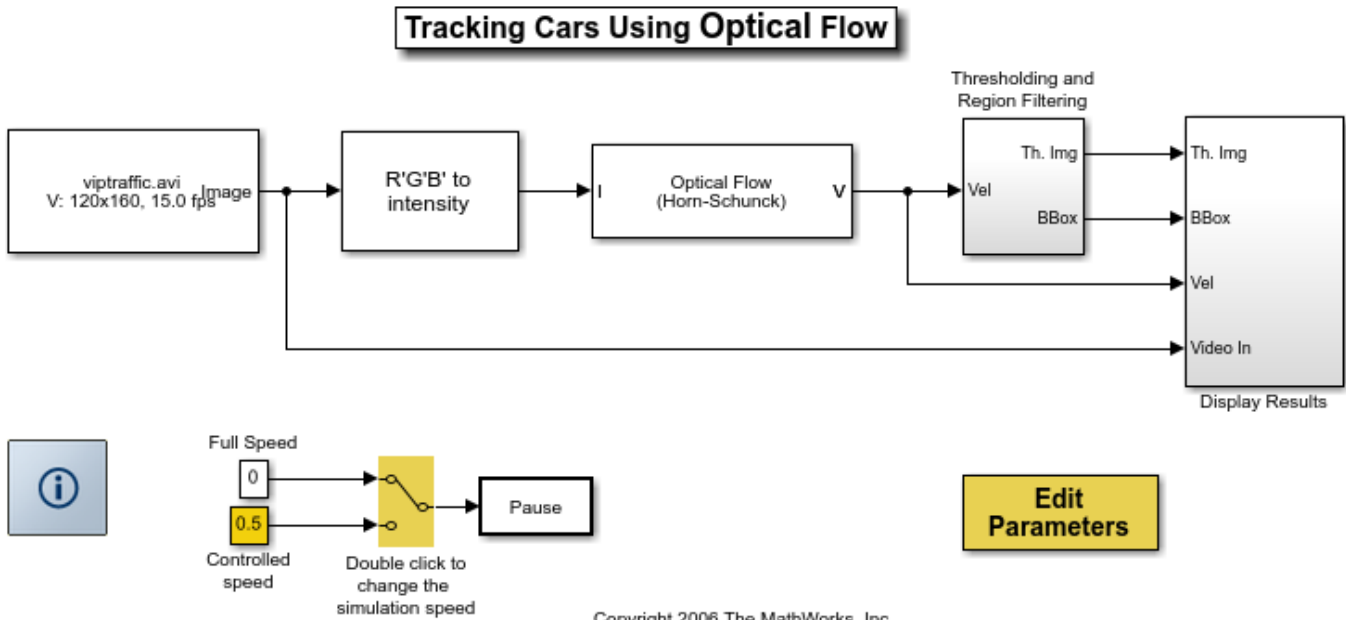
The algorithm in this example is suitable for an embedded software implementation. You can deploy it to an ARM™ processor using a Xilinx™ Zynq™ video processing reference design. See “Tracking Cars with Zynq-Based Hardware” (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Tracking Cars Using Optical Flow

This example shows how to detect and track cars in a video sequence using optical flow estimation.

Example Model

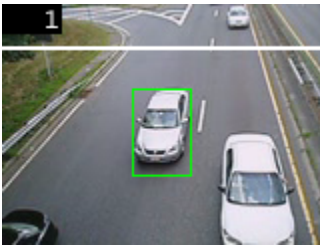
The following figure shows the Tracking Cars Using Optical Flow model:



Tracking Cars Using Optical Flow Results

The model uses an optical flow estimation technique to estimate the motion vectors in each frame of the video sequence. By thresholding the motion vectors, the model creates binary feature image containing blobs of moving objects. Median filtering is used to remove scattered noise; Close operation is performed moving to remove small holes in blobs. The model locates the cars in each binary feature image using the Blob Analysis block. Then it uses the Draw Shapes block to draw a green rectangle around the cars that pass beneath the white line. The counter in the upper left corner of the Results window tracks the number of cars in the region of interest.



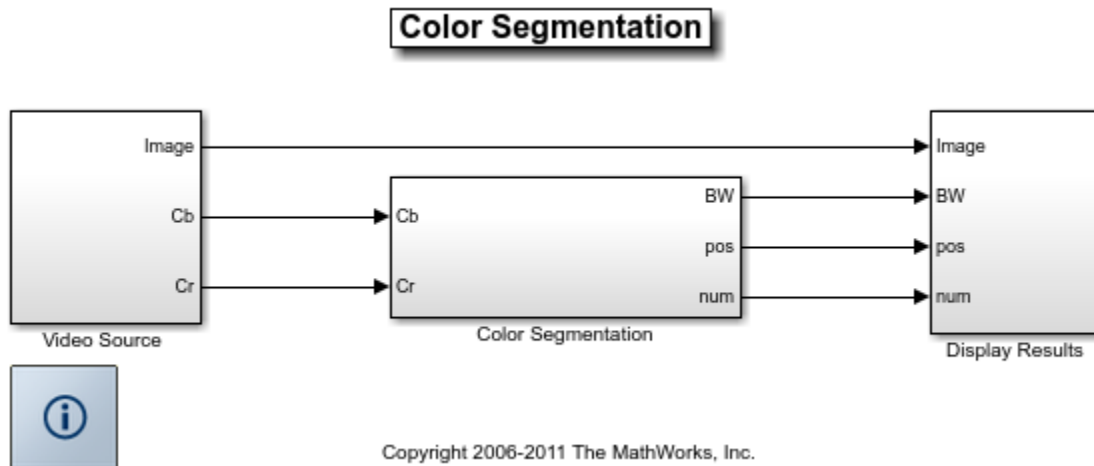


Tracking Based on Color

This example shows how to track a person's face and hand using a color-based segmentation method.

Example Model

The following figure shows the Color Segmentation example model:



Color Segmentation Results

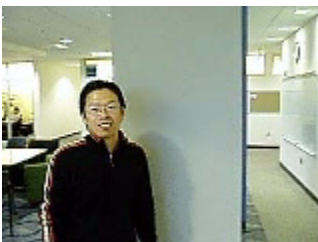
To create an accurate color model for the example, many images containing skin color samples were processed to compute the mean (m) and covariance (C) of the Cb and Cr color channels. Using this color model, the Color Segmentation/Color Classifier subsystem classifies each pixel as either skin or nonskin by computing the square of the Mahalanobis distance and comparing it to a threshold. The equation for the Mahalanobis distance is shown below:

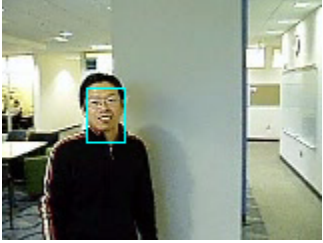
$$\text{SquaredDistance}(\text{Cb}, \text{Cr}) = (\mathbf{x} - \mathbf{m})' \text{inv}(\mathbf{C}) (\mathbf{x} - \mathbf{m}), \text{ where } \mathbf{x} = [\text{Cb}; \text{Cr}]$$

The result of this process is binary image, where pixel values equal to 1 indicate potential skin color locations.

The Color Segmentation/Filtering subsystem filters and performs morphological operations on each binary image, which creates the refined binary images shown in the Skin Region window.

The Color Segmentation/Region Filtering subsystem uses the Blob Analysis block and the Extract Face and Hand subsystem to determine the location of the person's face and hand in each binary image. The Display Results/Mark Image subsystem uses this location information to draw bounding boxes around these regions.





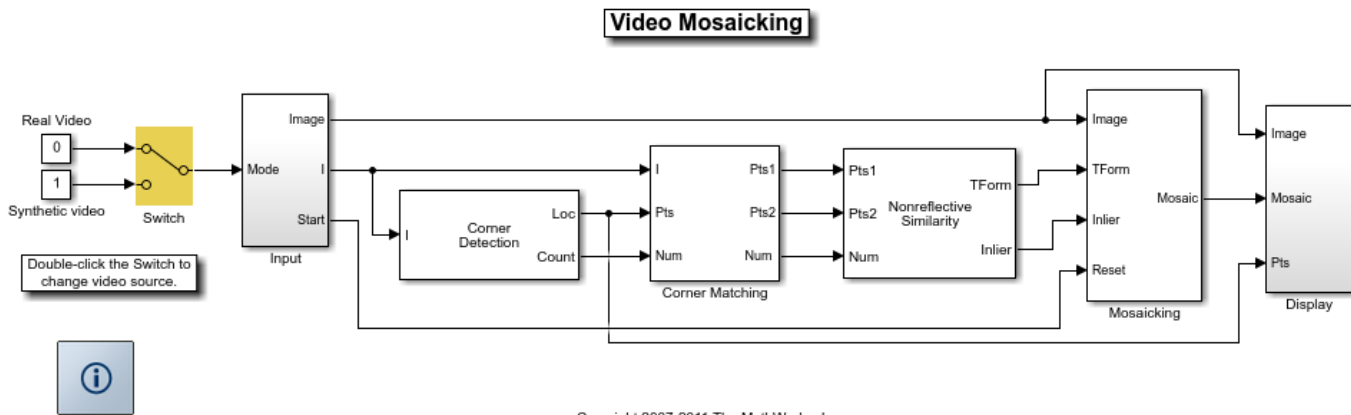
Video Mosaicking

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

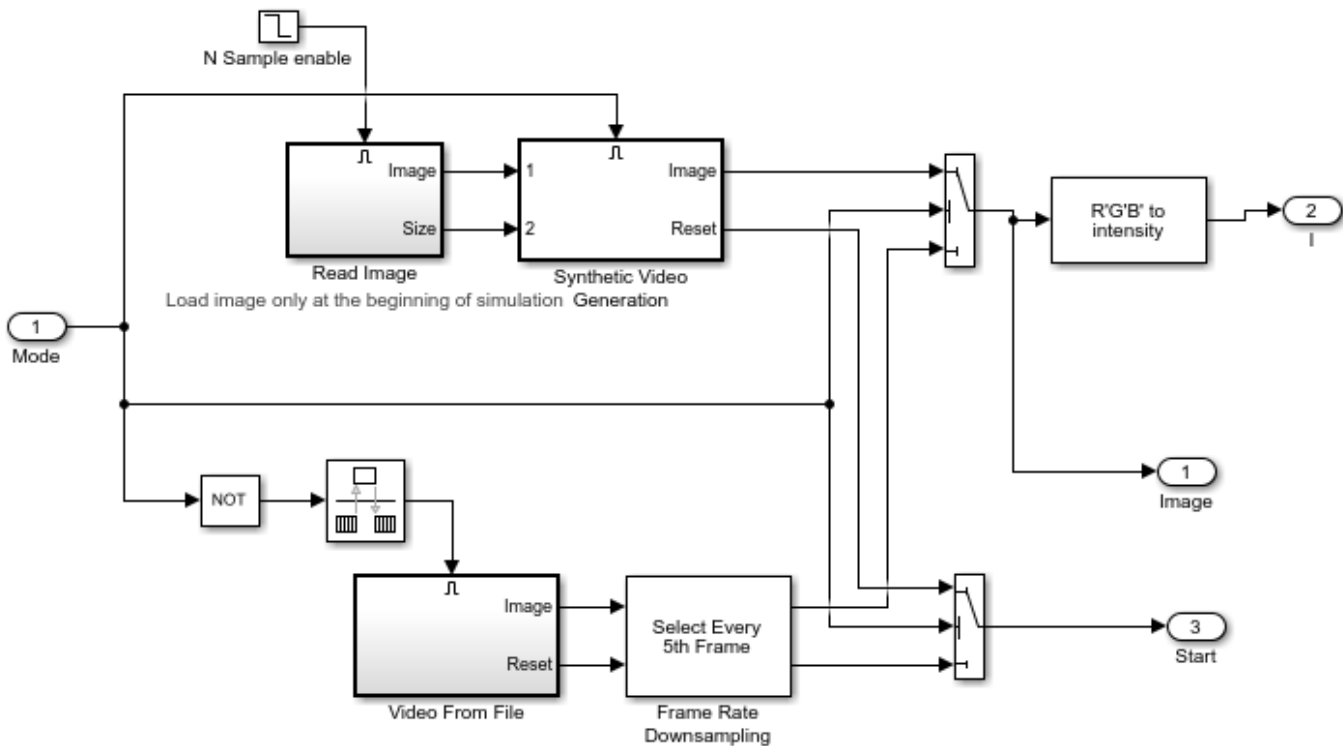
The following figure shows the Video Mosaicking model:



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.

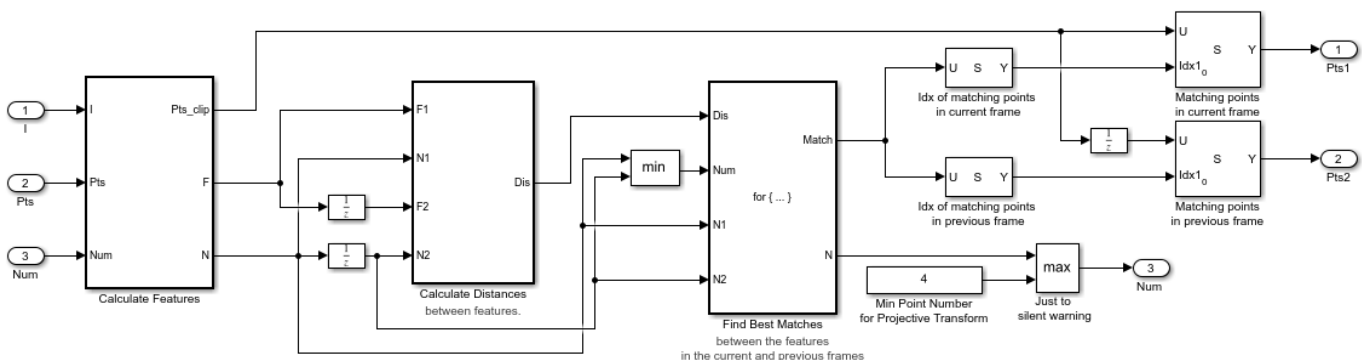


If you choose to use a video sequence from a file, you can reduce computation time by processing only some of the video frames. This is done by setting the downsampling rate in the Frame Rate Downsampling subsystem.

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

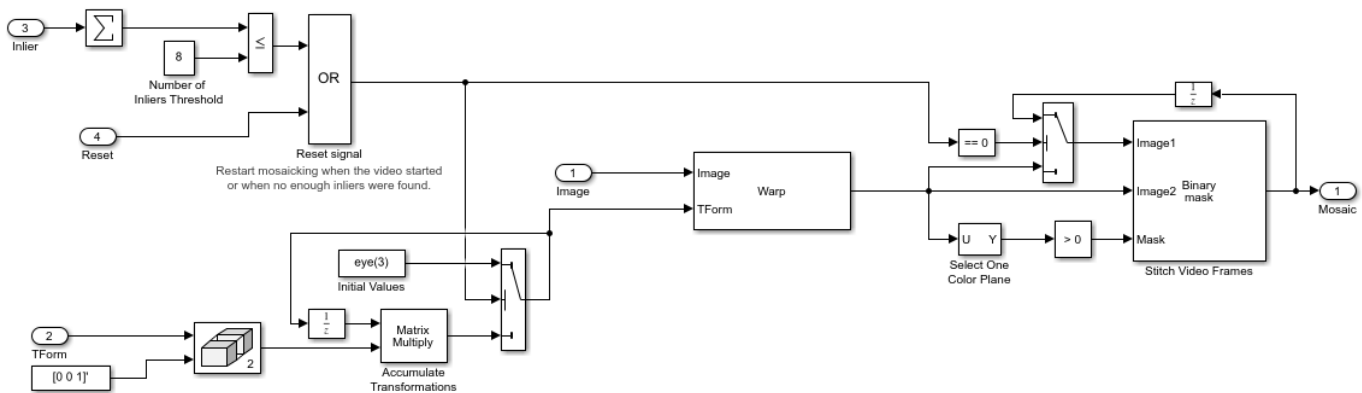
The subsystem finds corner features in the current video frame in one of three methods. The example uses Local intensity comparison (Rosen & Drummond), which is the fastest method. The other methods available are the Harris corner detection (Harris & Stephens) and the Minimum Eigenvalue (Shi & Tomasi).



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.

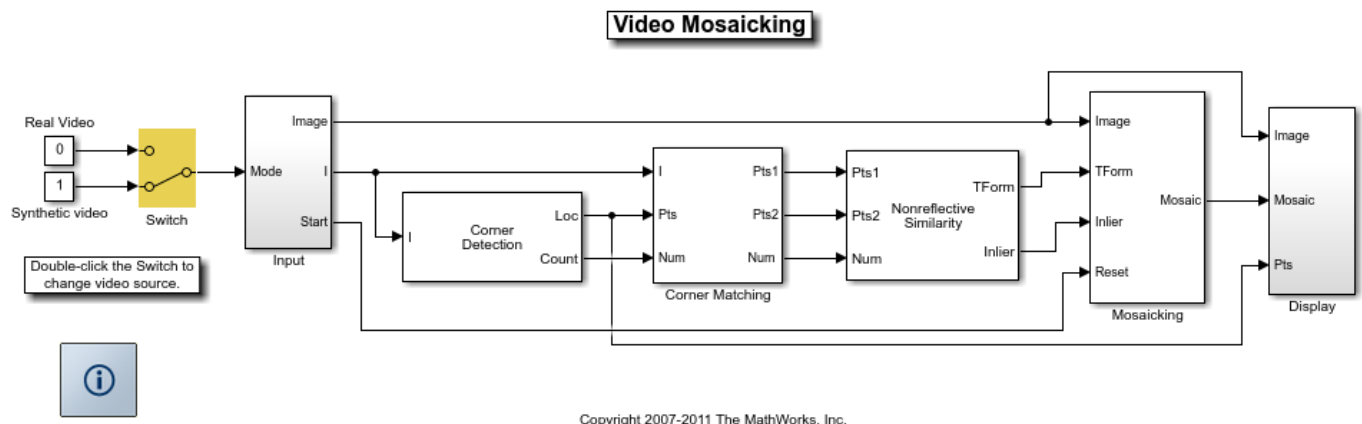
Mosaicking Subsystem

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



Copyright 2007-2011 The MathWorks, Inc.

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.

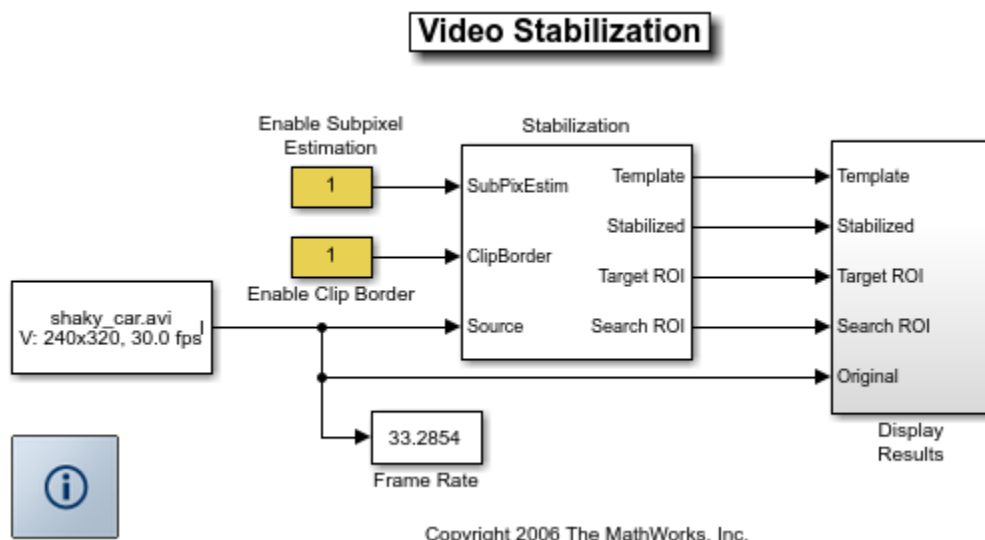


Video Stabilization

This example shows how to remove the effect of camera motion from a video stream. In the first video frame, the model defines the target to track. In this case, it is the back of a car and the license plate. It also establishes a dynamic search region, whose position is determined by the last known target location. The model only searches for the target within this search region, which reduces the number of computations required to find the target. In each subsequent video frame, the model determines how much the target has moved relative to the previous frame. It uses this information to remove unwanted translational camera motions and generate a stabilized video.

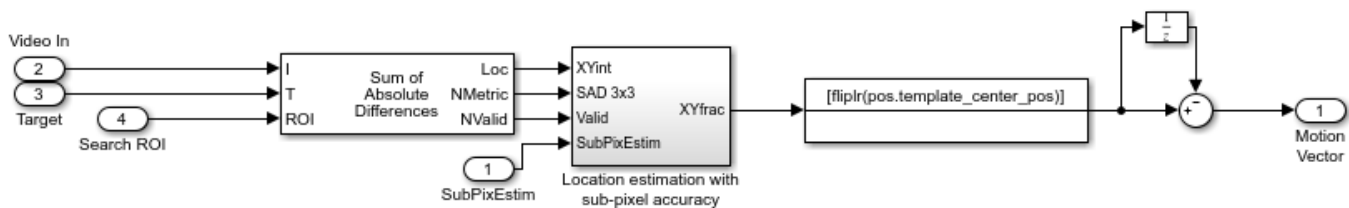
Example Model

The following figure shows the Video Stabilization model:



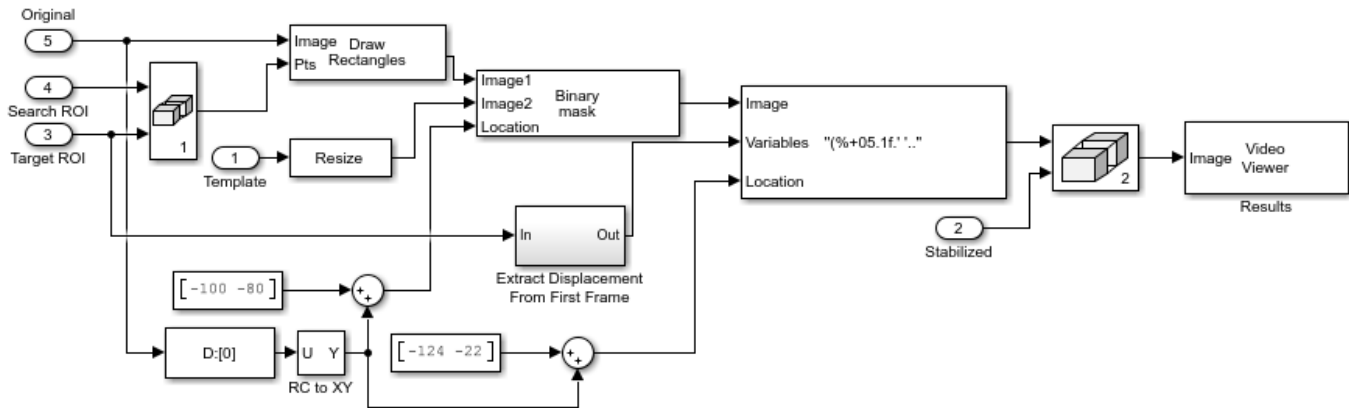
Estimate Motion Subsystem

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



Display Results Subsystem

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



Video Stabilization Results

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



Available Example Versions

Floating-point version of this example: vipstabilize.slx

Fixed-point version of this example: vipstabilize_fixpt.slx

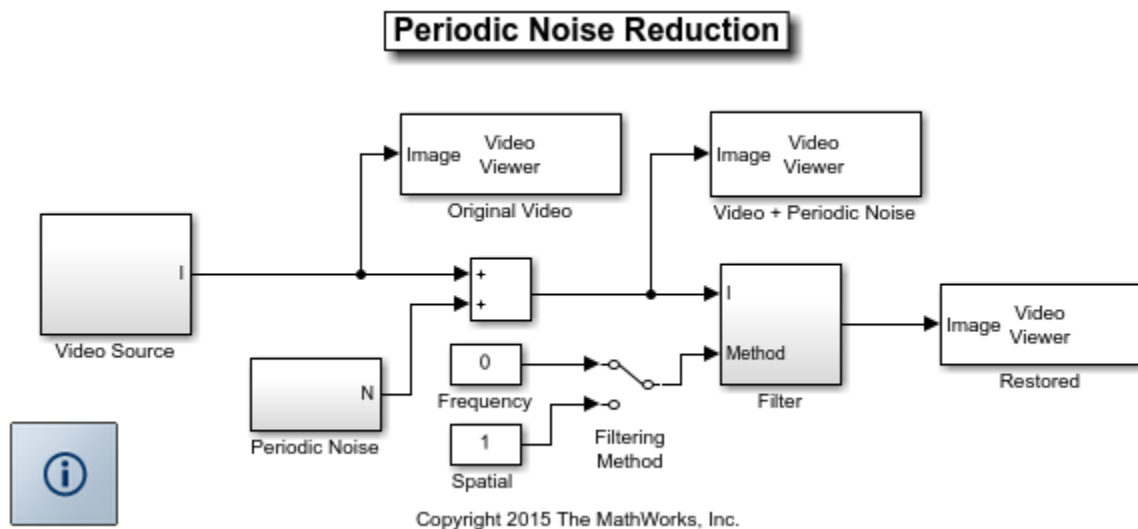
Fixed-point version of this example that simulates row major data organization:
vipstabilize_fixpt_rowmajor.slx

Periodic Noise Reduction

This example shows how to remove periodic noise from a video. In a video stream, periodic noise is typically caused by the presence of electrical or electromechanical interference during video acquisition or transmission. This type of noise is most effectively reduced with frequency domain filtering, which isolates the frequencies occupied by the noise and suppresses them using a band-reject filter.

Example Model

The following figure shows the Periodic Noise Reduction example model:

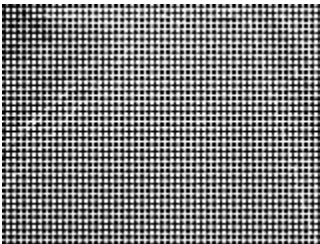


Periodic Noise Reduction Results

This example creates periodic noise by adding two 2-D sinusoids with varying frequency and phase to the video frames. Then it removes this noise using a frequency-domain or spatial-domain filter. You can specify which filter the example uses by double-clicking the Filtering Method switch.

For the frequency-domain filter, the model uses a binary mask, which it creates using Draw Shapes blocks, to eliminate a band of frequencies from the frequency domain representation of the image. For the spatial-domain filter, the model uses the 2-D FIR Filter block and precomputed band-reject filter coefficients that were derived using the Filter Designer (filterDesigner) and the ftrans2 function.



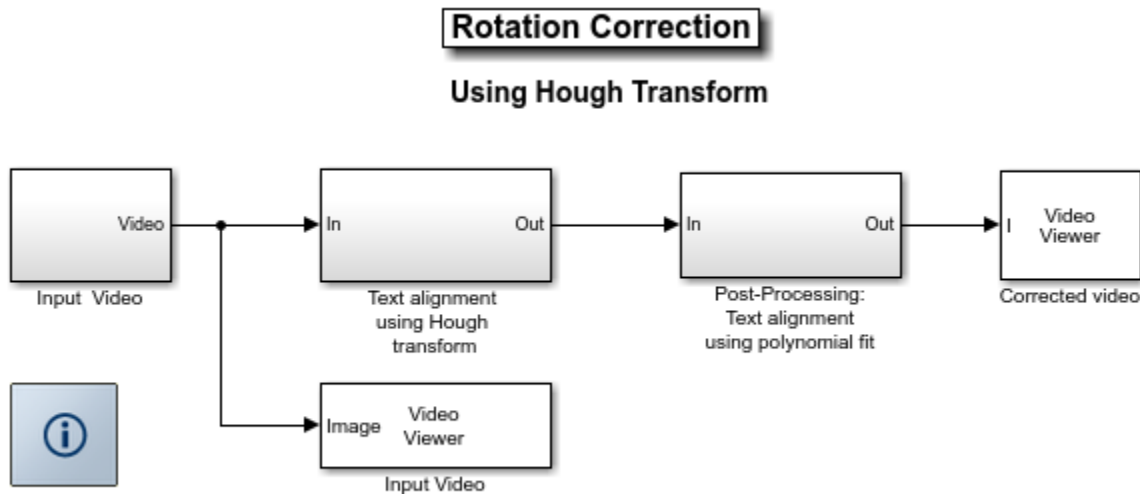


Rotation Correction

This example shows how to use the Hough Transform and Polyfit blocks to horizontally align text rotating in a video sequence. The techniques illustrated by this example can be used in video stabilization and optical character recognition (OCR).

Example Model

The following figure shows the Rotation Correction example model:

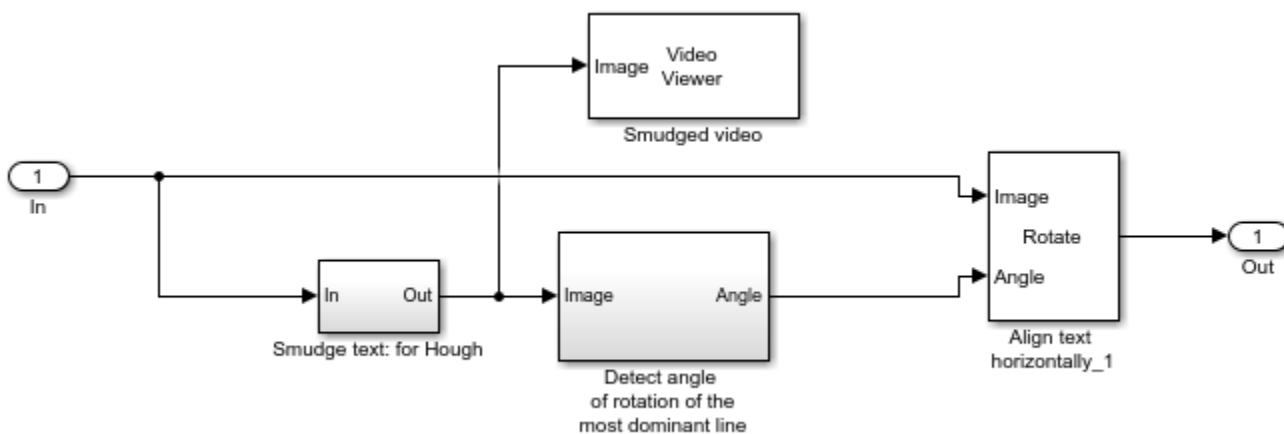


Copyright 2004-2011 The MathWorks, Inc.

Text Alignment Using Hough Transform Subsystem

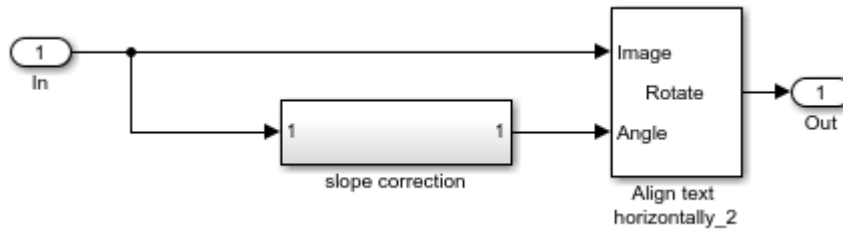
The morphological operators in the Smudge text subsystem blur the letters to create a binary image with two distinct lines. You can see the result of this process in the Smudged Video window.

By transforming the binary image into the Hough parameter space, the example determines the theta and rho values of the lines created by the Smudge text subsystem. Once the theta values of the text lines are known, the example uses the Rotate block to eliminate the large angular variations.



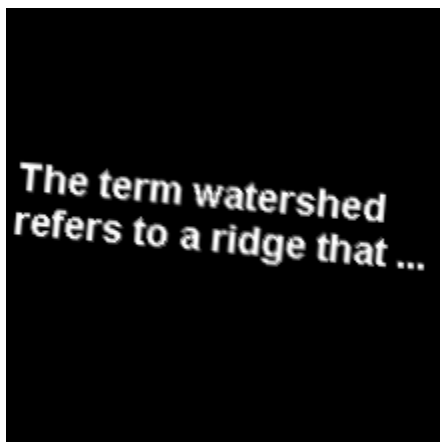
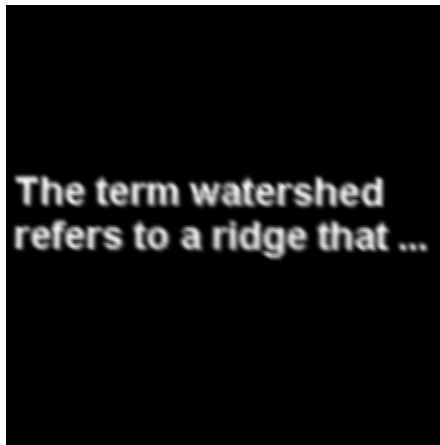
Post-Processing: Text Alignment Using Polynomial Fit Subsystem

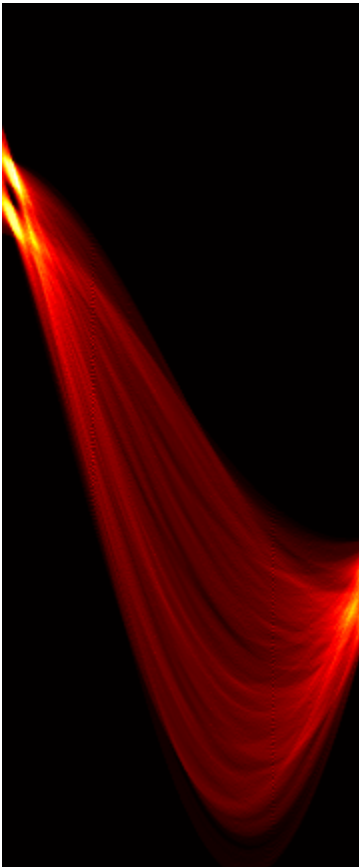
The example uses the Polyfit block, in the slope correction subsystem, and the Rotate block to eliminate small angular variations in the text. The Polyfit block fits a straight line to the smudged text. Then the slope correction subsystem calculates the slope of the line and its angle of inclination. The Rotate block uses this angle to correct for the small rotations.



Rotation Correction Results

The Input Video window shows the original video. The Smudged video window shows the result of blurring the letters to create a binary image with two distinct lines. In the Hough Matrix window, the x- and y-coordinates of the two dominant yellow dots correspond to the theta and rho values of the text lines, respectively. The Corrected video window shows the result of the rotation correction process.





Barcode Recognition Using Live Video Acquisition

This example shows how to use the From Video Device block provided by Image Acquisition Toolbox™ to acquire live image data from a Point Grey Flea® 2 camera into Simulink®. The example uses the Computer Vision Toolbox™ to create an image processing system which can recognize and interpret a GTIN-13 barcode. The GTIN-13 barcode, formally known as EAN-13, is an international barcode standard. It is a superset of the widely used UPC standard.

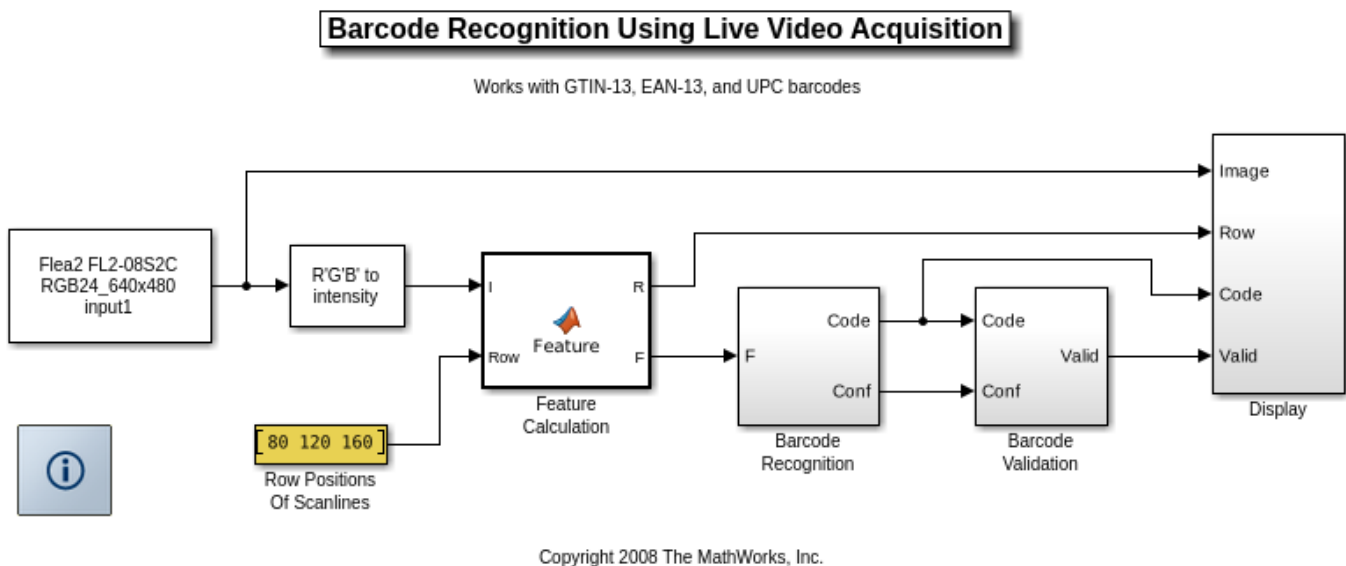
Image Acquisition Toolbox™ provides functions for acquiring images and video directly into MATLAB® and Simulink from PC-compatible imaging hardware. You can detect hardware automatically, configure hardware properties, preview an acquisition, and acquire images and video.

This example requires Image Acquisition Toolbox and a Point Grey Flea® 2 camera to run the model.

Watch barcode recognition on live video stream. (11 seconds)

Example Model

The following figure shows the example model using the From Video Device block.

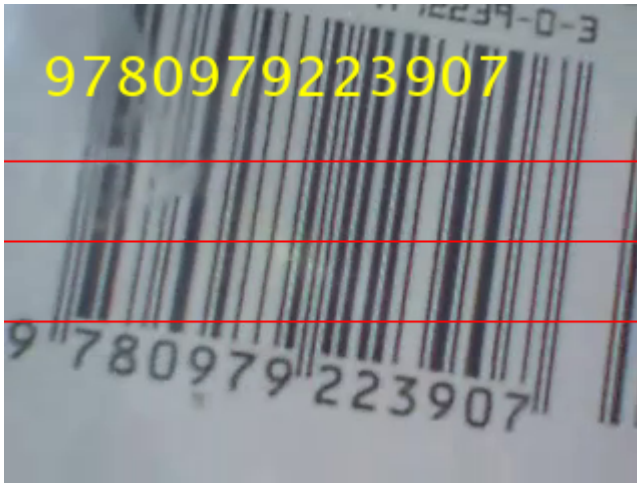


Example Description

This example uses the same algorithm as the Barcode Recognition example. Refer to the Barcode Recognition example for detailed information.

Results

The scan lines that have been used to detect barcodes are displayed in red. When a GTIN-13 is correctly recognized and verified, the code is displayed at the top of the image.



Even though a Point Grey Flea® 2 camera was used for this example, you can update this model to use other supported image acquisition devices, for example, webcams. This enables you to use the same Simulink model with different image acquisition hardware. Before using this example, please adjust the focus of your imaging device such that the barcodes are legible.

Available Example Versions

Example using live video acquisition: `viplivebarcoderecognition_win.slx` (Windows® only)

Example using stored video data: `vipbarcoderecognition.slx` (platform independent)

Edge Detection Using Live Video Acquisition

This example shows how to use the From Video Device block provided by Image Acquisition Toolbox™ to acquire live image data from a Hamamatsu C8484 camera into Simulink®. The Prewitt method is applied to find the edges of objects in the input video stream.

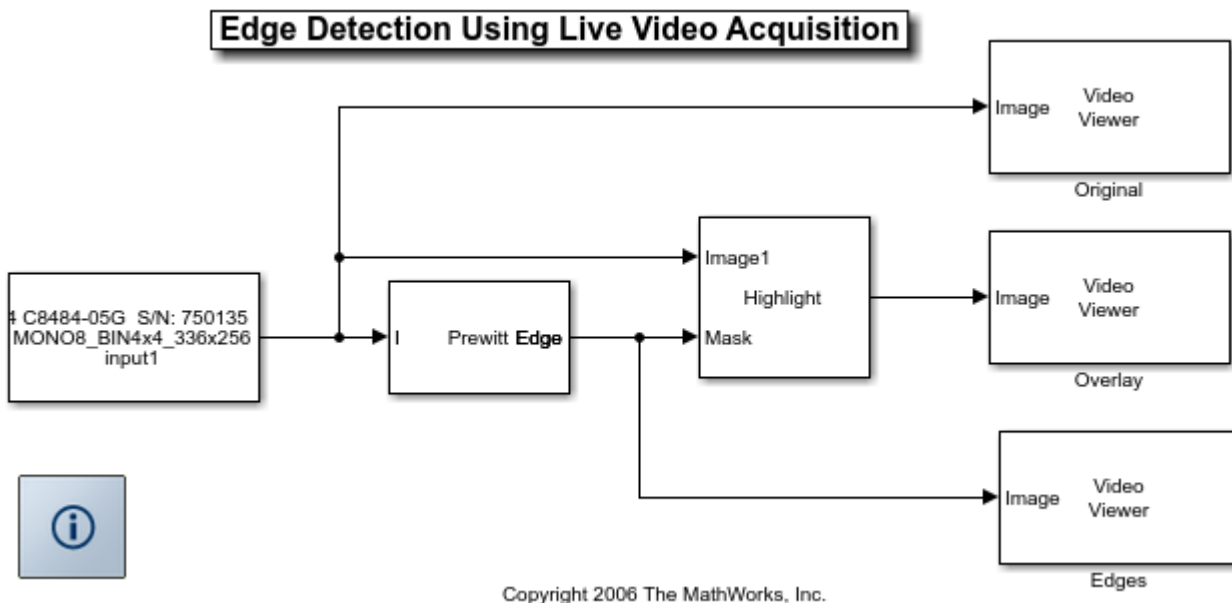
Image Acquisition Toolbox provides functions for acquiring images and video directly into MATLAB® and Simulink from PC-compatible imaging hardware. You can detect hardware automatically, configure hardware properties, preview an acquisition, and acquire images and video.

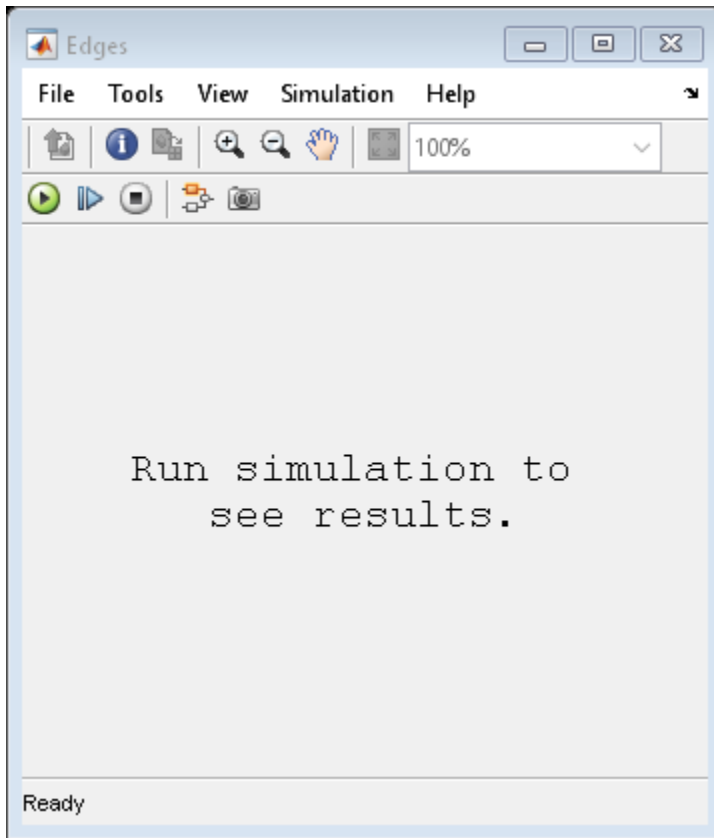
This example requires Image Acquisition Toolbox and Hamamatsu image acquisition device (C8484) to run the model.

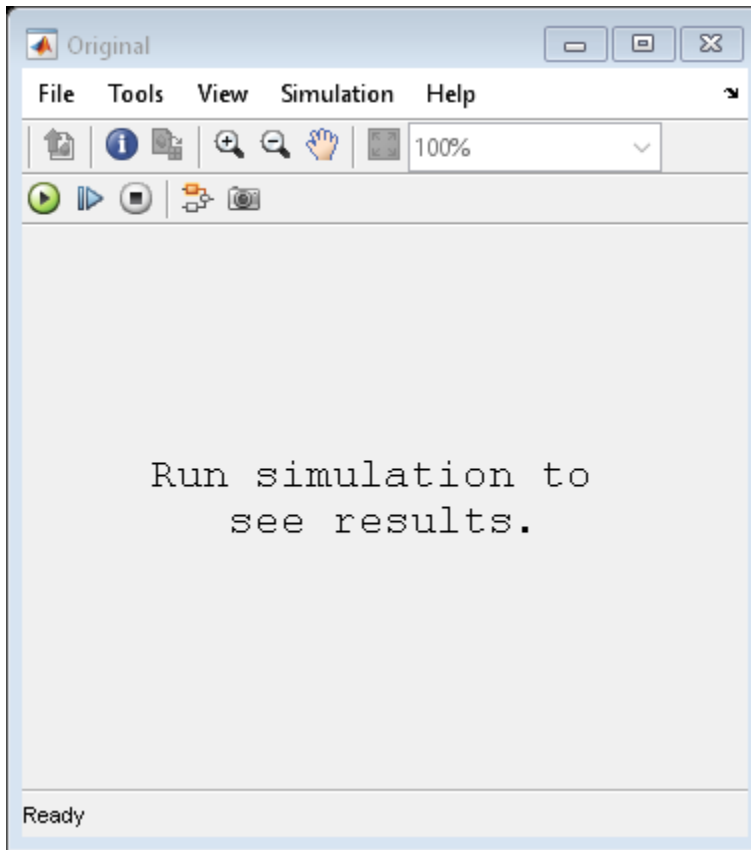
Watch edge detection using live video acquisition. (4 seconds)

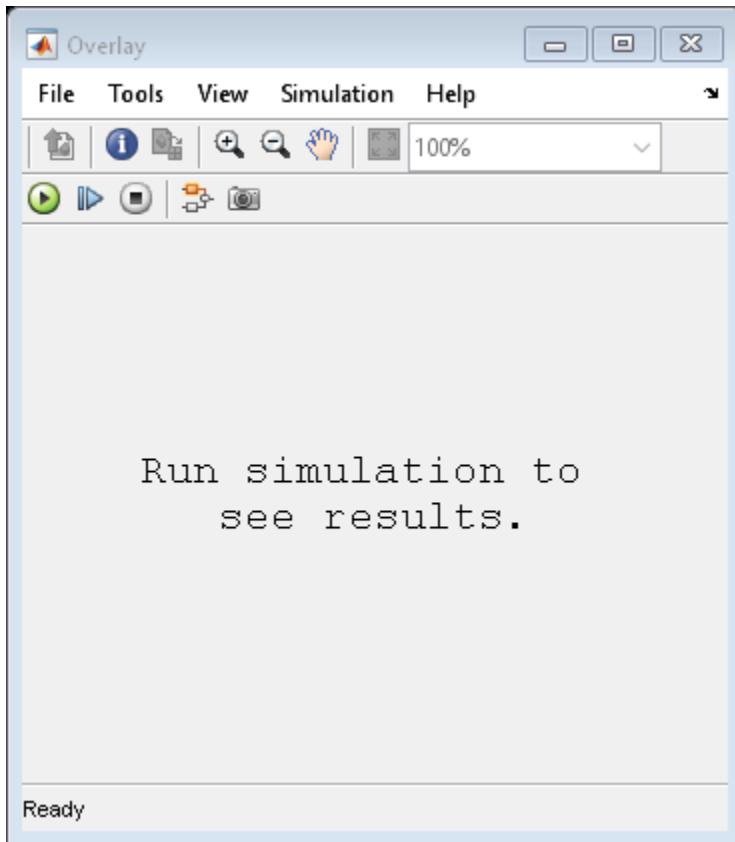
Example Model

The following figure shows the example model using the From Video Device block.







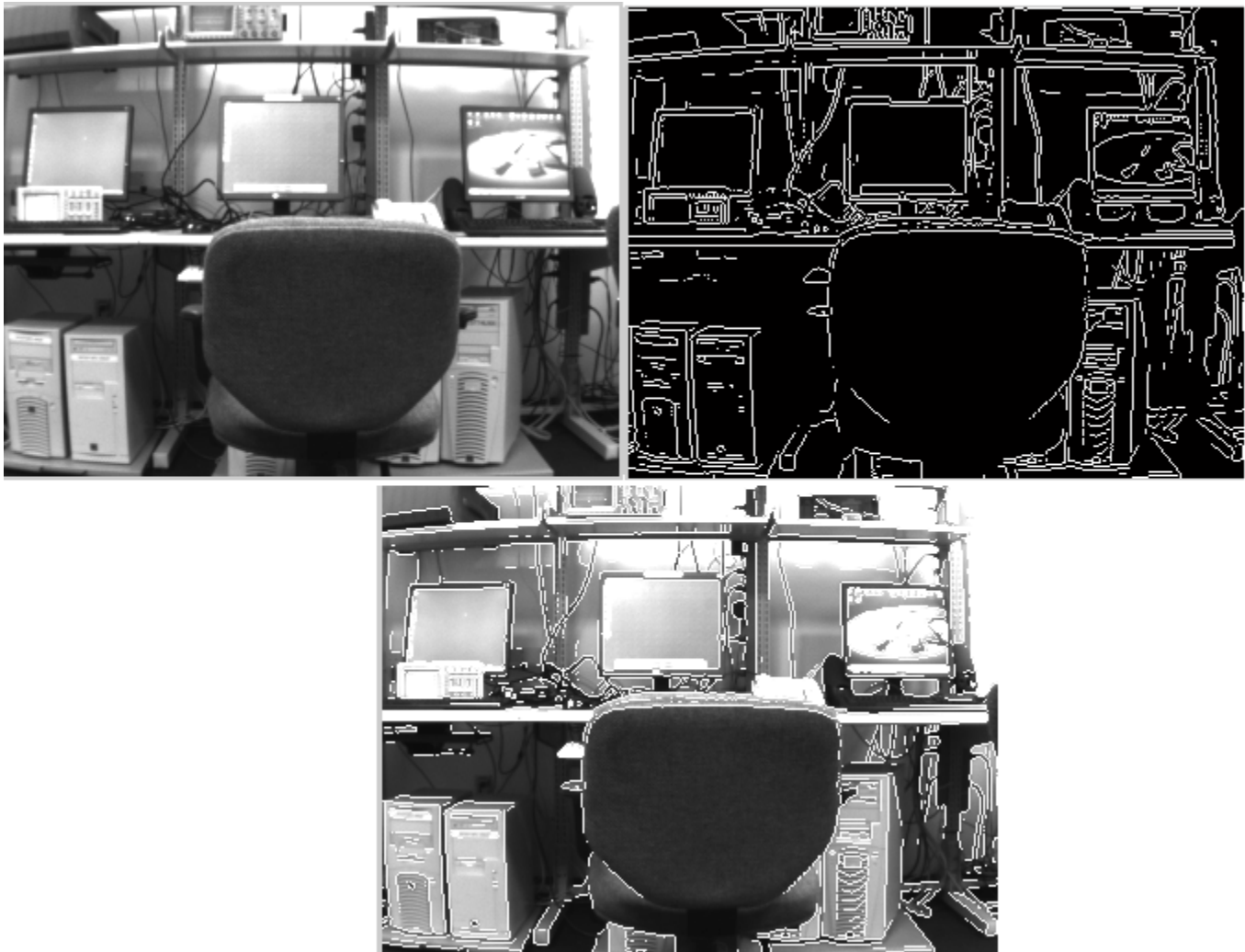


Live Video Input

The example acquires the input video live from a Hamamatsu image acquisition device (C8484). In this example, the block acquires intensity data from the camera and outputs it into the Simulink model at every simulation time step.

Edge Detection Analysis

This example uses Computer Vision Toolbox™ to find the edges of objects in the video input. When you run the model, you can double-click the Edge Detection block and adjust the threshold parameter while the simulation is running. The higher you make the threshold, the smaller the amount of edges the example finds in the video stream.



Even though a Hamamatsu camera was used for this example, you can update this model to use other supported image acquisition devices. This enables you to use the same Simulink model with different image acquisition hardware.

Noise Removal and Image Sharpening

This example shows how to use Vision HDL Toolbox™ to implement an FPGA-based module for image enhancement.

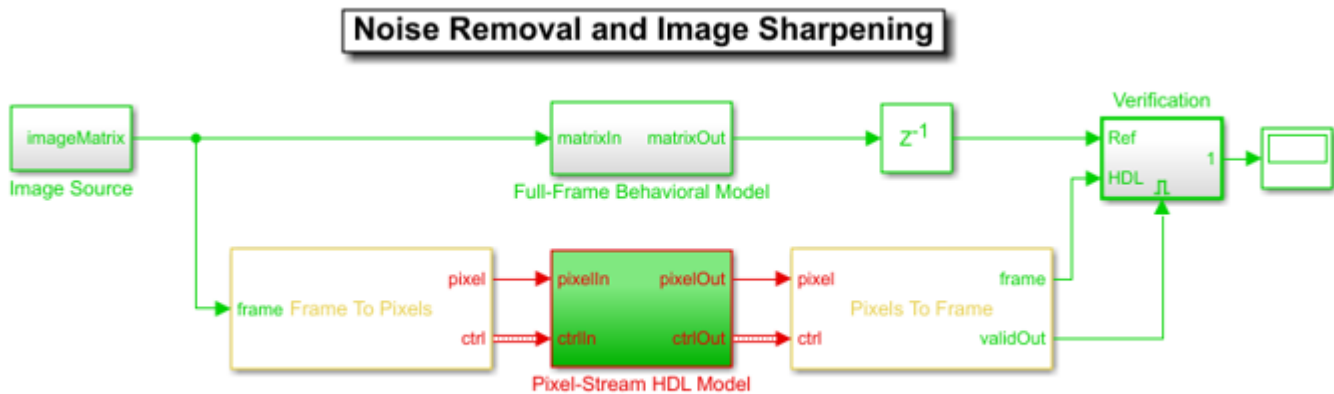
Vision HDL Toolbox provides video processing algorithms designed to generate readable, synthesizable code in VHDL and Verilog (with HDL Coder™). The generated HDL code can process 1080p video at a rate of 60 frames per second.

The Computer Vision Toolbox™ product models at a high level of abstraction. The blocks and objects perform full-frame processing, operating on one image frame at a time. However, FPGA or ASIC systems perform pixel-stream processing, operating on one image pixel at a time.

Input images from physical systems frequently contain impairments such as blur and noise. An object out of focus results in a blurred image. Dead or stuck pixels on the camera or video sensor, or thermal noise from hardware components, contribute to the noise in the image. This example removes noise and sharpens the input image, and it can be used at an early stage of the processing chain to provide a better initial condition for subsequent processing. This example uses two pixel-stream filter blocks from the Vision HDL Toolbox. The median filter removes the noise and the image filter sharpens the image. To verify the pixel-stream design, the results are compared with those generated by the full-frame blocks from the Computer Vision Toolbox.

Model Overview

The NoiseRemovalAndSharpeningHDLExample.slx system is shown.

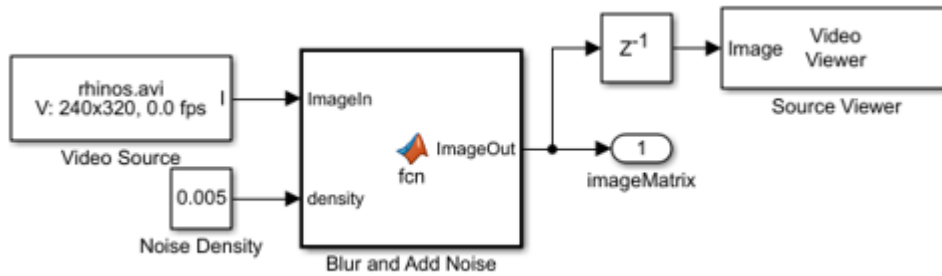


Copyright 2015 The MathWorks, Inc.

Computer Vision Toolbox blocks operate on an entire frame at a time. Vision HDL Toolbox blocks operate on a stream of pixel data, one pixel at a time. The conversion blocks in Vision HDL Toolbox, Frame To Pixels and Pixels To Frame, enable you to simulate streaming-pixel designs and to compare with full-frame designs.

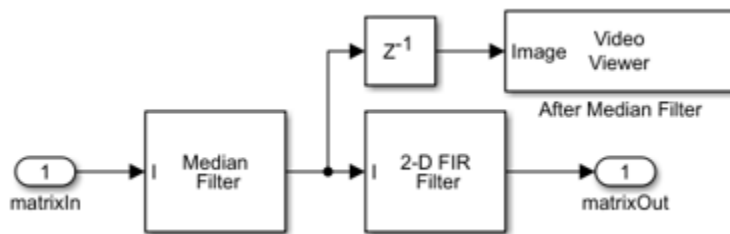
The difference in the color of the lines feeding the Full-Frame Behavioral Model and Pixel-Stream HDL Model subsystems indicates the change in the image rate on the streaming branch of the model. This rate transition occurs because the pixel stream is sent out in the same amount of time as the full video frames and therefore it is transmitted at a higher rate. To turn on colors and view sample time information, in the left palette, click the **Sample Time** icon and select **Colors**.

The following figure shows the Image Source subsystem.

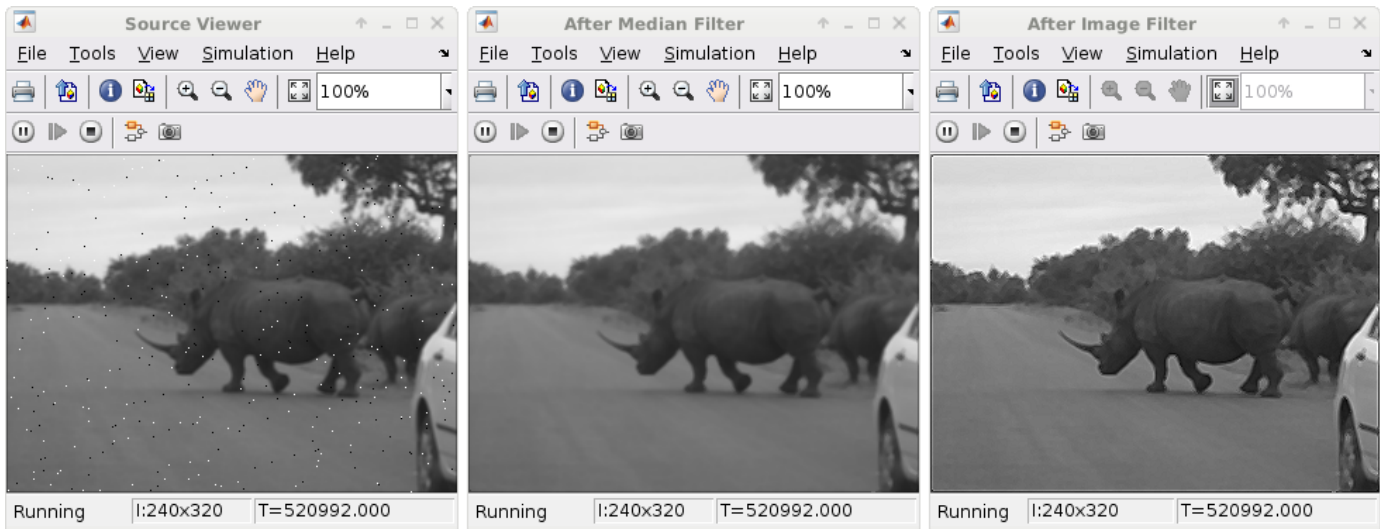


The Image Source block imports a greyscale image, then uses a MATLAB function block named Blur and Add Noise to blur the image and inject salt-and-pepper noise. The `imfilter` function uses a 3-by-3 averaging kernel to blur the image. The salt-and-pepper noise is injected by calling the `imnoise` command. The noise density is defined as the ratio of the combined number of salt and pepper pixels to the total pixels in the image. This density value is specified by the Noise Density constant block, and it must be between 0 and 1. The Image Source subsystem outputs a 2-D matrix of a full-frame image.

The diagram below shows the structure of the Full-Frame Behavioral Model subsystem, which consists of the frame-based Median Filter and 2-D FIR Filter from Computer Vision Toolbox. Median filter removes the noise and 2-D FIR Filter is configured to sharpen the image.



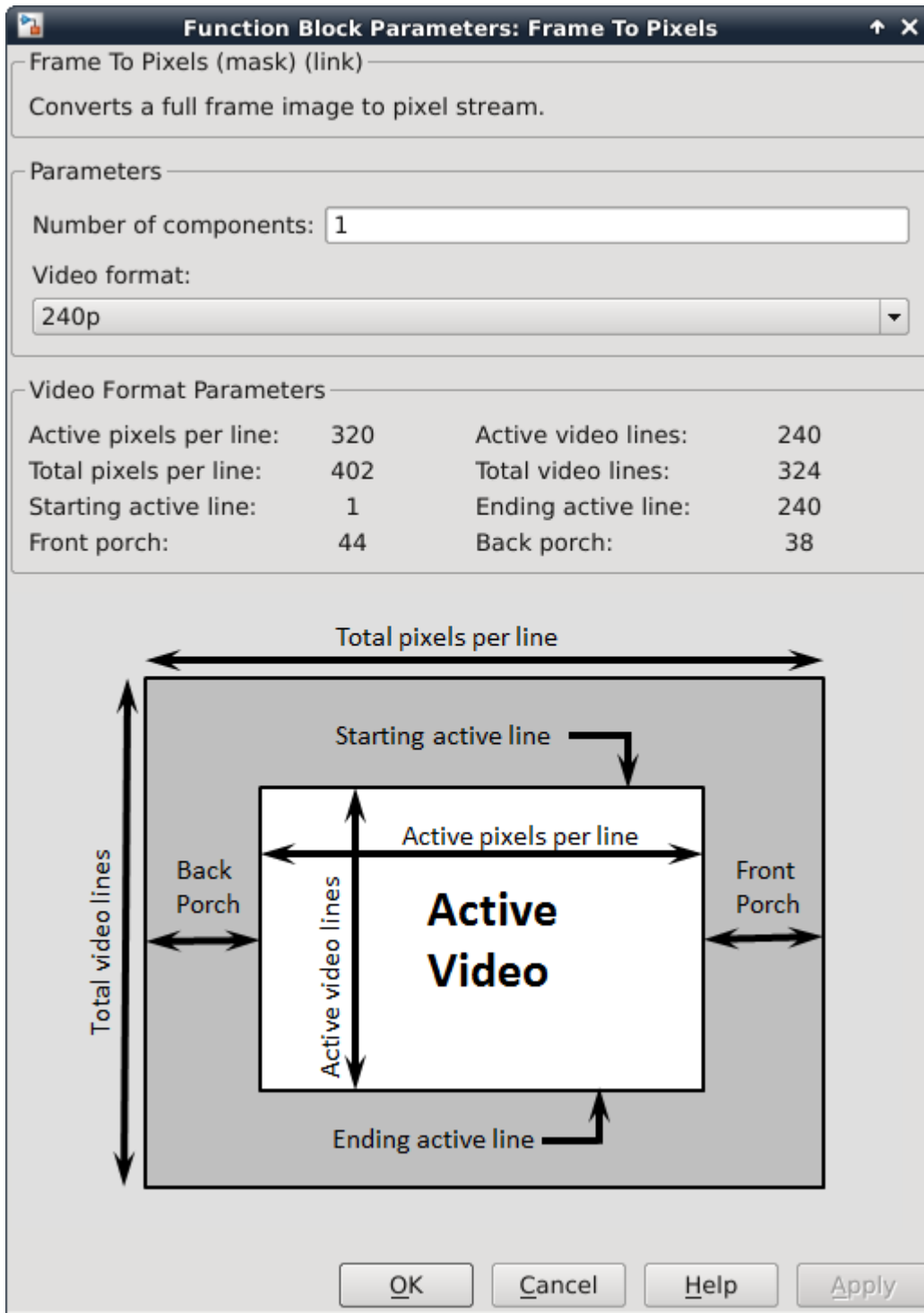
The displays below show one frame of the blurred and noisy source video, its de-noised version after median filtering, and the sharpened output after 2-D FIR filtering.



The Pixel-Stream HDL Model subsystem uses Vision HDL Toolbox to implement streaming based median filter and 2-D FIR filter. The Verification subsystem compares the results from full-frame processing with those from pixel-stream processing. These two subsystems are described in the next two sections.

Pixel-Streaming HDL Design

The Frame To Pixels block converts a full-frame image to a pixel stream since blocks in Vision HDL Toolbox operate on stream input signals required by FPGA hardware. To simulate the effect of horizontal and vertical blanking periods found in video systems based on FPGAs or ASICs, the active image is augmented with non-image data. For more information on the streaming pixel protocol, see the “Streaming Pixel Interface” (Vision HDL Toolbox). The Frame To Pixels block is configured as shown:



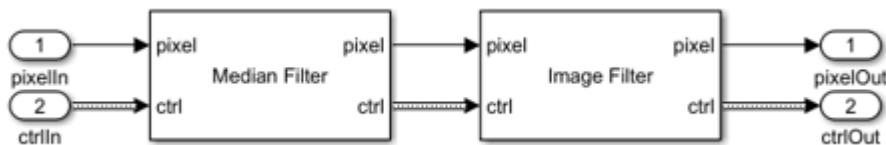
The Number of components field is set to 1 for grayscale image input, and the Video format field is 240p to match that of the video source.

In this example, the Active Video region corresponds to the 240x320 matrix of the blurred and noisy image from the upstream Image Source subsystem. Six other parameters, namely, Total pixels per

line, Total video lines, Starting active line, Ending active line, Front porch, and Back porch specify how many non-image pixels will be added on the four sides of the Active Video. For more information, see the Frame To Pixels (Vision HDL Toolbox) block reference page.

Note that the Desired sample time of the Video Source inside Image Source is determined by the product of Total pixels per line and Total video lines.

The Pixel-Stream HDL Model subsystem contains the streaming implementation of the median filter and 2-D FIR filter from Vision HDL Toolbox, as shown in the diagram below. You can generate HDL code from the Pixel-Stream HDL Model subsystem using HDL Coder™.

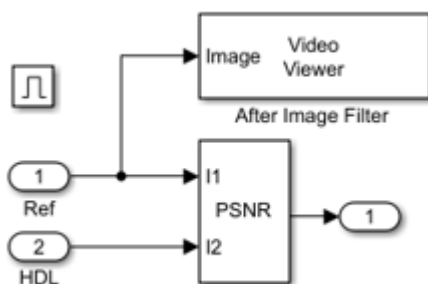


The Median Filter (Vision HDL Toolbox) block is used to remove the salt and pepper noise. Based on the filter coefficients, the Image Filter (Vision HDL Toolbox) block can be used to blur, sharpen, or detect the edges of the recovered image after median filtering. In this example, Image Filter is configured to sharpen an image.

Verifying the Pixel-Stream Processing Design

In order to compare with the output of the full-frame filters from the Computer Vision Toolbox, the model converts the pixel stream data back to full frame using the Pixels To Frame block. The Number of components field and the Video format fields of both Frame To Pixels and Pixels To Frame are set at 1 and 240p, respectively, to match the format of the video source.

The output of the Pixels To Frame block is a 2-D matrix of a full image. This allows us to compare the HDL model against the behavioral model in the full-frame domain, as shown in the Verification subsystem shown below.



The peak signal to noise ratio (PSNR) is calculated between the reference image and the stream processed image. Ideally, the ratio should be inf, indicating that the output image from the Full-Frame Behavioral Model matches that generated from the Pixel-Stream HDL Model.

Generate HDL Code and Verify Its Behavior

To check and generate the HDL code referenced in this example, you must have an HDL Coder license.

To generate the HDL code, use the following command:

```
makehdl('NoiseRemovalAndSharpeningHDLExample/Pixel-Stream HDL Model');
```

To generate test bench, use the following command:

```
makehdltb('NoiseRemovalAndSharpeningHDLExample/Pixel-Stream HDL Model');
```

Tracking and Motion Estimation Examples

- “Video Stabilization” on page 7-2
- “Video Stabilization Using Point Feature Matching” on page 7-5
- “Face Detection and Tracking Using CAMShift” on page 7-15
- “Face Detection and Tracking Using the KLT Algorithm” on page 7-20
- “Face Detection and Tracking Using Live Video Acquisition” on page 7-26
- “Motion-Based Multiple Object Tracking” on page 7-31
- “Tracking Pedestrians from a Moving Car” on page 7-40
- “Use Kalman Filter for Object Tracking” on page 7-50
- “Detect Cars Using Gaussian Mixture Models” on page 7-61

Video Stabilization

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

Introduction

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

Initialization

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

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

```
hVideoSource = VideoReader(filename);
```

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

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

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

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

Here we initialize some variables used in the processing loop.

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

Stream Processing Loop

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

```

while hasFrame(hVideoSource)
    input = rgb2gray(im2double(readFrame(hVideoSource)));

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

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

        MotionVector = double(Idx-IdxPrev);
    end

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

    % Translate video frame to offset the camera motion
    Stabilized = imtranslate(input, Offset, 'linear');

    Target = Stabilized(TargetRowIndices, TargetColIndices);

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

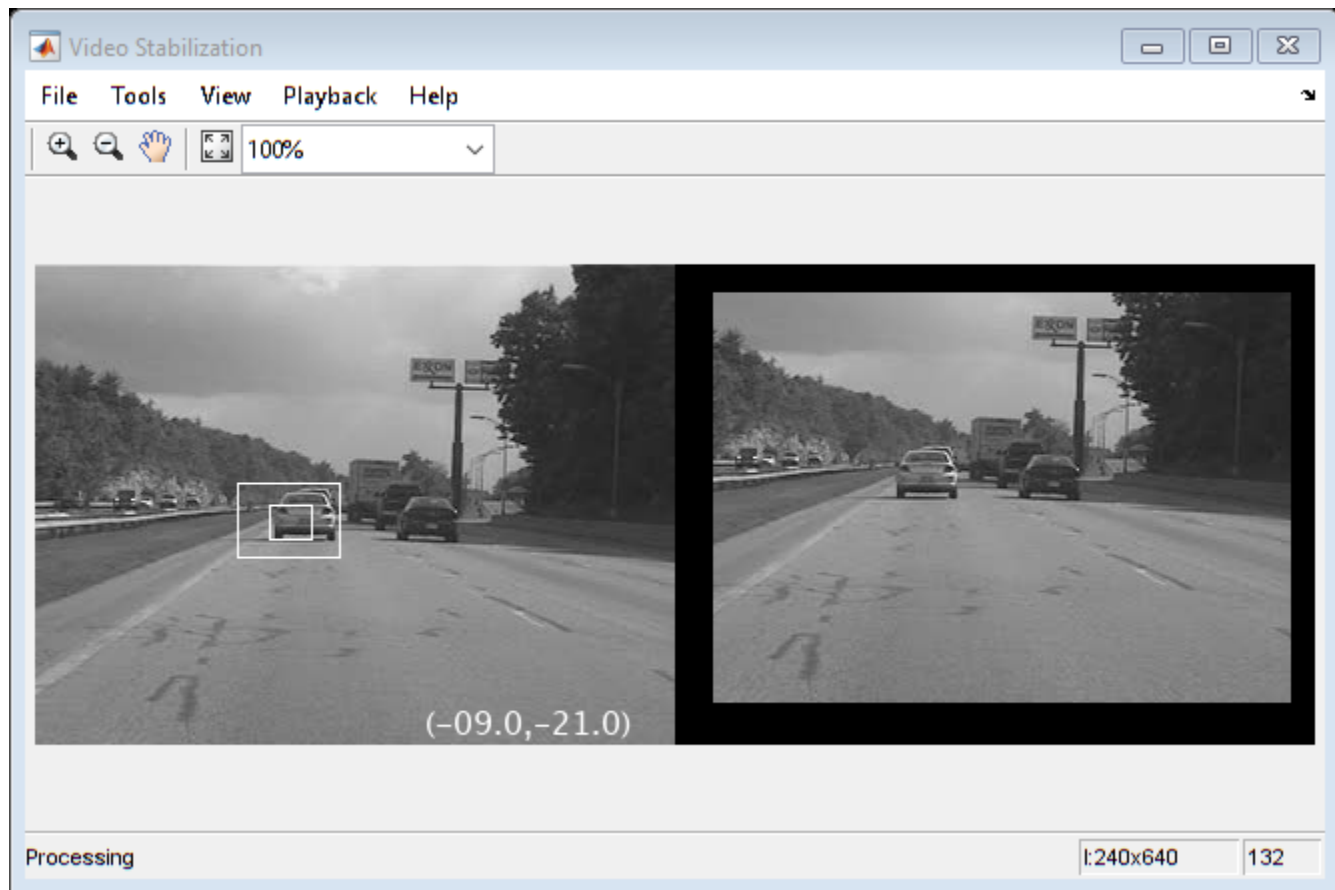
    TargetRect = [pos.template_orig-Offset, pos.template_size];
    SearchRegionRect = [SearchRegion, pos.template_size + 2*pos.search_border];

    % Draw rectangles on input to show target and search region
    input = insertShape(input, 'Rectangle', [TargetRect; SearchRegionRect],...
        'Color', 'white');

    % Display the offset (displacement) values on the input image
    txt = sprintf(' %+05.1f,%+05.1f', Offset);
    input = insertText(input(:,:,1),[191 215],txt,'FontSize',16, ...
        'TextColor', 'white', 'BoxOpacity', 0);

    % Display video
    hVideoOut([input(:,:,1) Stabilized]);
end

```



Conclusion

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

Appendix

The following helper function is used in this example.

- `updatesearch.m`

Video Stabilization Using Point Feature Matching

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

This stabilization algorithm involves two steps. First, we determine the affine image transformations between all neighboring frames of a video sequence using the `estimateGeometricTransform2D` function applied to point correspondences between two images. Second, we warp the video frames to achieve a stabilized video. We will use the Computer Vision Toolbox™, both for the algorithm and for display.

Step 1. Read Frames from a Movie File

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

```
filename = 'shaky_car.avi';
hVideoSrc = VideoReader(filename);

imgA = rgb2gray(im2single(readFrame(hVideoSrc))); % Read first frame into imgA
imgB = rgb2gray(im2single(readFrame(hVideoSrc))); % Read second frame into imgB

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



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

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



Step 2. Collect Salient Points from Each Frame

Our goal is to determine a transformation that will correct for the distortion between the two frames. We can use the `estimateGeometricTransform2D` function for this, which will return an affine transform. As input we must provide this function with a set of point correspondences between the two frames. To generate these correspondences, we first collect points of interest from both frames, then select likely correspondences between them.

In this step we produce these candidate points for each frame. To have the best chance that these points will have corresponding points in the other frame, we want points around salient image features such as corners. For this we use the `detectFASTFeatures` function, which implements one of the fastest corner detection algorithms.

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

```
ptThresh = 0.1;
pointsA = detectFASTFeatures(imgA, 'MinContrast', ptThresh);
pointsB = detectFASTFeatures(imgB, 'MinContrast', ptThresh);

% Display corners found in images A and B.
figure; imshow(imgA); hold on;
plot(pointsA);
title('Corners in A');
```


Corners in A



```
figure; imshow(imgB); hold on;  
plot(pointsB);  
title('Corners in B');
```

Corners in B



Step 3. Select Correspondences Between Points

Next we pick correspondences between the points derived above. For each point, we extract a Fast Retina Keypoint (FREAK) descriptor centered around it. The matching cost we use between points is

the Hamming distance since FREAK descriptors are binary. Points in frame A and frame B are matched putatively. Note that there is no uniqueness constraint, so points from frame B can correspond to multiple points in frame A.

```
% Extract FREAK descriptors for the corners
[featuresA, pointsA] = extractFeatures(imgA, pointsA);
[featuresB, pointsB] = extractFeatures(imgB, pointsB);
```

Match features which were found in the current and the previous frames. Since the FREAK descriptors are binary, the `matchFeatures` function uses the Hamming distance to find the corresponding points.

```
indexPairs = matchFeatures(featuresA, featuresB);
pointsA = pointsA(indexPairs(:, 1), :);
pointsB = pointsB(indexPairs(:, 2), :);
```

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

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



Step 4. Estimating Transform from Noisy Correspondences

Many of the point correspondences obtained in the previous step are incorrect. But we can still derive a robust estimate of the geometric transform between the two images using the M-estimator Sample Consensus (MSAC) algorithm, which is a variant of the RANSAC algorithm. The MSAC algorithm is implemented in the `estimateGeometricTransform2D` function. This function, when given a set of point correspondences, will search for the valid inlier correspondences. From these it will then derive the affine transform that makes the inliers from the first set of points match most closely with the inliers from the second set. This affine transform will be a 3-by-3 matrix of the form:

```
[a_1 a_3 0;
 a_2 a_4 0;
 t_x t_y 1]
```

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

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

```
[tform, inlierIdx] = estimateGeometricTransform2D(...
    pointsB, pointsA, 'affine');
pointsBm = pointsB(inlierIdx, :);
pointsAm = pointsA(inlierIdx, :);
imgBp = imwarp(imgB, tform, 'OutputView', imref2d(size(imgB)));
pointsBmp = transformPointsForward(tform, pointsBm.Location);
```

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

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

```
figure;
showMatchedFeatures(imgA, imgBp, pointsAm, pointsBmp);
legend('A', 'B');
```



Step 5. Transform Approximation and Smoothing

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

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

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

```
[s*cos(ang)  s*-sin(ang)  0;
 s*sin(ang)  s*cos(ang)  0;
 t_x         t_y         1]
```

We show this conversion procedure below by fitting the above-obtained transform H with a scale-rotation-translation equivalent, H_{sRt} . To show that the error of converting the transform is minimal, we reproject frame B with both transforms and show the two images below as a red-cyan color composite. As the image appears black and white, obviously the pixel-wise difference between the different reprojections is negligible.

```
% Extract scale and rotation part sub-matrix.
H = tform.T;
R = H(1:2,1:2);
% Compute theta from mean of two possible arctangents
theta = mean([atan2(R(2),R(1)) atan2(-R(3),R(4))]);
% Compute scale from mean of two stable mean calculations
scale = mean(R([1 4])/cos(theta));
```

```

% Translation remains the same:
translation = H(3, 1:2);
% Reconstitute new s-R-t transform:
HsRt = [[scale*[cos(theta) -sin(theta); sin(theta) cos(theta)]; ...
        translation], [0 0 1]'];
tformsRT = affine2d(HsRt);

imgBold = imwarp(imgB, tform, 'OutputView', imref2d(size(imgB)));
imgBsRt = imwarp(imgB, tformsRT, 'OutputView', imref2d(size(imgB)));

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

```

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



Step 6. Run on the Full Video

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

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

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

```

% Reset the video source to the beginning of the file.
read(hVideoSrc, 1);

hVPlayer = vision.VideoPlayer; % Create video viewer

```

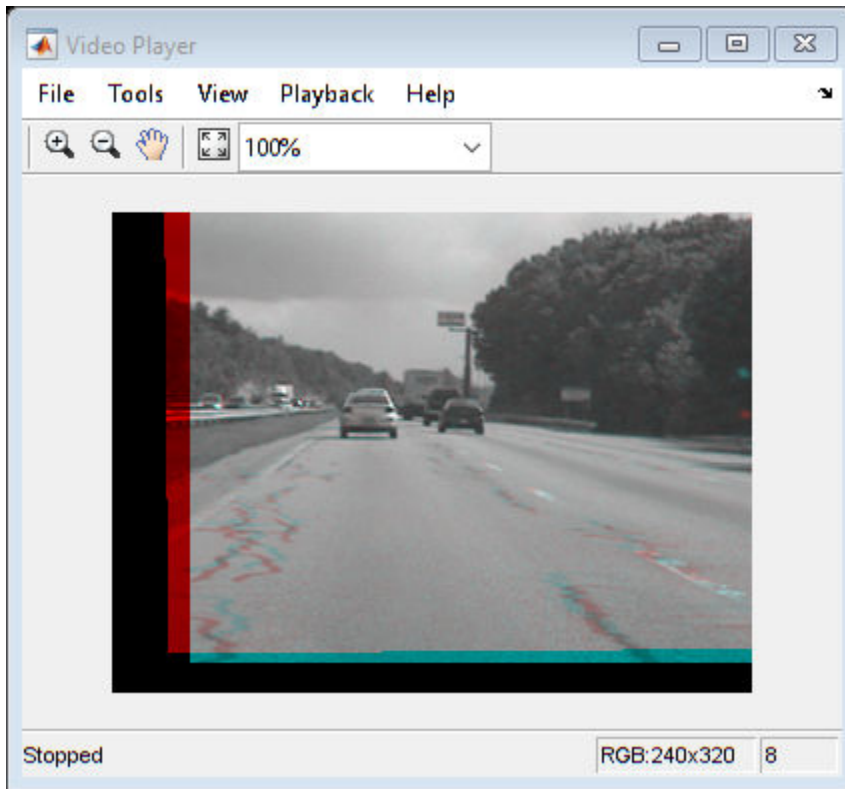
```
% Process all frames in the video
movMean = rgb2gray(im2single(readFrame(hVideoSrc)));
imgB = movMean;
imgBp = imgB;
correctedMean = imgBp;
ii = 2;
Hcumulative = eye(3);
while hasFrame(hVideoSrc) && ii < 10
    % Read in new frame
    imgA = imgB; % z^-1
    imgAp = imgBp; % z^-1
    imgB = rgb2gray(im2single(readFrame(hVideoSrc)));
    movMean = movMean + imgB;

    % Estimate transform from frame A to frame B, and fit as an s-R-t
    H = cvexEstStabilizationTform(imgA,imgB);
    HsRt = cvexTformToSRT(H);
    Hcumulative = HsRt * Hcumulative;
    imgBp = imwarp(imgB,affine2d(Hcumulative),'OutputView',imref2d(size(imgB)));

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

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

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



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

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



References

- [1] Tordoff, B; Murray, DW. "Guided sampling and consensus for motion estimation." European Conference n Computer Vision, 2002.
- [2] Lee, KY; Chuang, YY; Chen, BY; Ouhyoung, M. "Video Stabilization using Robust Feature Trajectories." National Taiwan University, 2009.
- [3] Litvin, A; Konrad, J; Karl, WC. "Probabilistic video stabilization using Kalman filtering and mosaicking." IS&T/SPIE Symposium on Electronic Imaging, Image and Video Communications and Proc., 2003.
- [4] Matsushita, Y; Ofek, E; Tang, X; Shum, HY. "Full-frame Video Stabilization." Microsoft® Research Asia. CVPR 2005.

Face Detection and Tracking Using CAMShift

This example shows how to automatically detect and track a face.

Introduction

Object detection and tracking are important in many computer vision applications including activity recognition, automotive safety, and surveillance. In this example, you will develop a simple face tracking system by dividing the tracking problem into three separate problems:

- 1 Detect a face to track
- 2 Identify facial features to track
- 3 Track the face

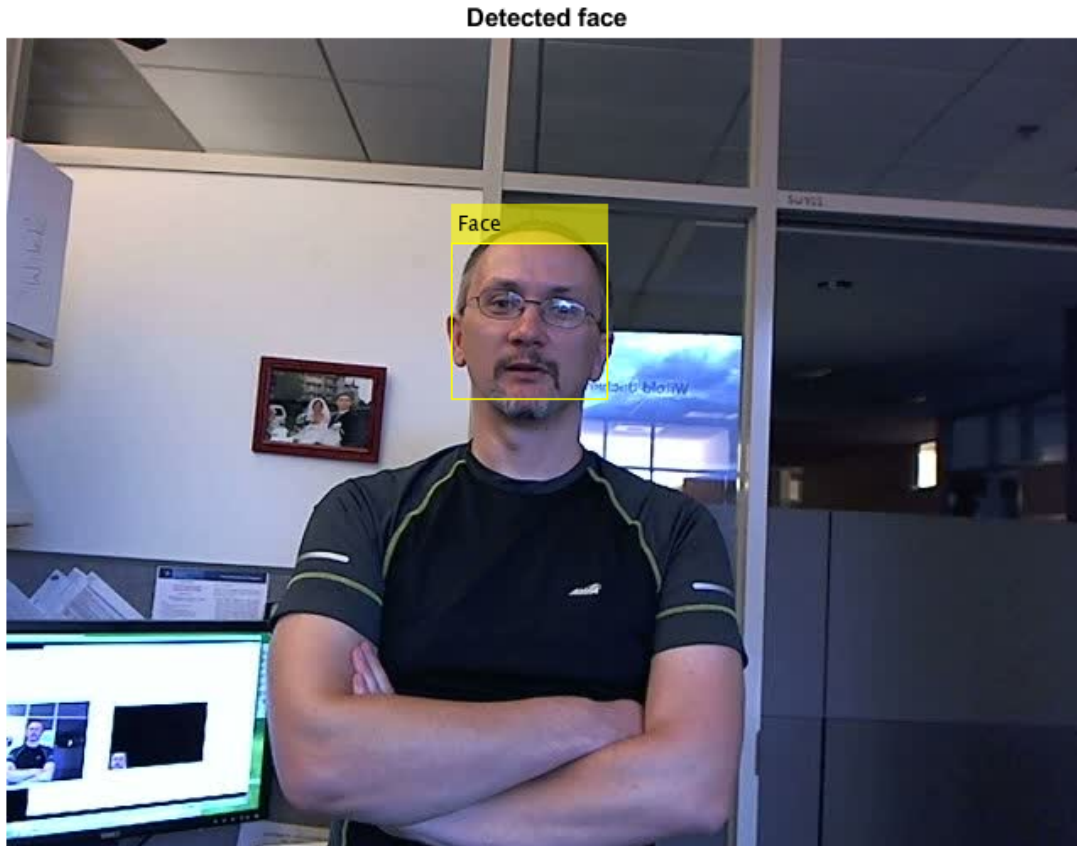
Step 1: Detect a Face To Track

Before you begin tracking a face, you need to first detect it. Use the `vision.CascadeObjectDetector` to detect the location of a face in a video frame. The cascade object detector uses the Viola-Jones detection algorithm and a trained classification model for detection. By default, the detector is configured to detect faces, but it can be configured for other object types.

```
% Create a cascade detector object.
faceDetector = vision.CascadeObjectDetector();

% Read a video frame and run the detector.
videoFileReader = VideoReader('visionface.avi');
videoFrame      = readFrame(videoFileReader);
bbox            = step(faceDetector, videoFrame);

% Draw the returned bounding box around the detected face.
videoOut = insertObjectAnnotation(videoFrame,'rectangle',bbox,'Face');
figure, imshow(videoOut), title('Detected face');
```



You can use the cascade object detector to track a face across successive video frames. However, when the face tilts or the person turns their head, you may lose tracking. This limitation is due to the type of trained classification model used for detection. To avoid this issue, and because performing face detection for every video frame is computationally intensive, this example uses a simple facial feature for tracking.

Step 2: Identify Facial Features To Track

Once the face is located in the video, the next step is to identify a feature that will help you track the face. For example, you can use the shape, texture, or color. Choose a feature that is unique to the object and remains invariant even when the object moves.

In this example, you use skin tone as the feature to track. The skin tone provides a good deal of contrast between the face and the background and does not change as the face rotates or moves.

Get the skin tone information by extracting the Hue from the video frame converted to the HSV color space.

```
[hueChannel,~,~] = rgb2hsv(videoFrame);
```

```
% Display the Hue Channel data and draw the bounding box around the face.
figure, imshow(hueChannel), title('Hue channel data');
rectangle('Position',bbox(1,:), 'LineWidth',2, 'EdgeColor',[1 1 0])
```



Step 3: Track the Face

With the skin tone selected as the feature to track, you can now use the `vision.HistogramBasedTracker` for tracking. The histogram based tracker uses the CAMShift algorithm, which provides the capability to track an object using a histogram of pixel values. In this example, the Hue channel pixels are extracted from the nose region of the detected face. These pixels are used to initialize the histogram for the tracker. The example tracks the object over successive video frames using this histogram.

Detect the nose within the face region. The nose provides a more accurate measure of the skin tone because it does not contain any background pixels.

```
noseDetector = vision.CascadeObjectDetector('Nose', 'UseROI', true);
noseBBBox    = step(noseDetector, videoFrame, bbox(1,:));

% Create a tracker object.
tracker = vision.HistogramBasedTracker;

% Initialize the tracker histogram using the Hue channel pixels from the
% nose.
initializeObject(tracker, hueChannel, noseBBBox(1,:));
```

```

% Create a video player object for displaying video frames.
videoPlayer = vision.VideoPlayer;

% Track the face over successive video frames until the video is finished.
while hasFrame(videoFileReader)

    % Extract the next video frame
    videoFrame = readFrame(videoFileReader);

    % RGB -> HSV
    [hueChannel,~,~] = rgb2hsv(videoFrame);

    % Track using the Hue channel data
    bbox = step(tracker, hueChannel);

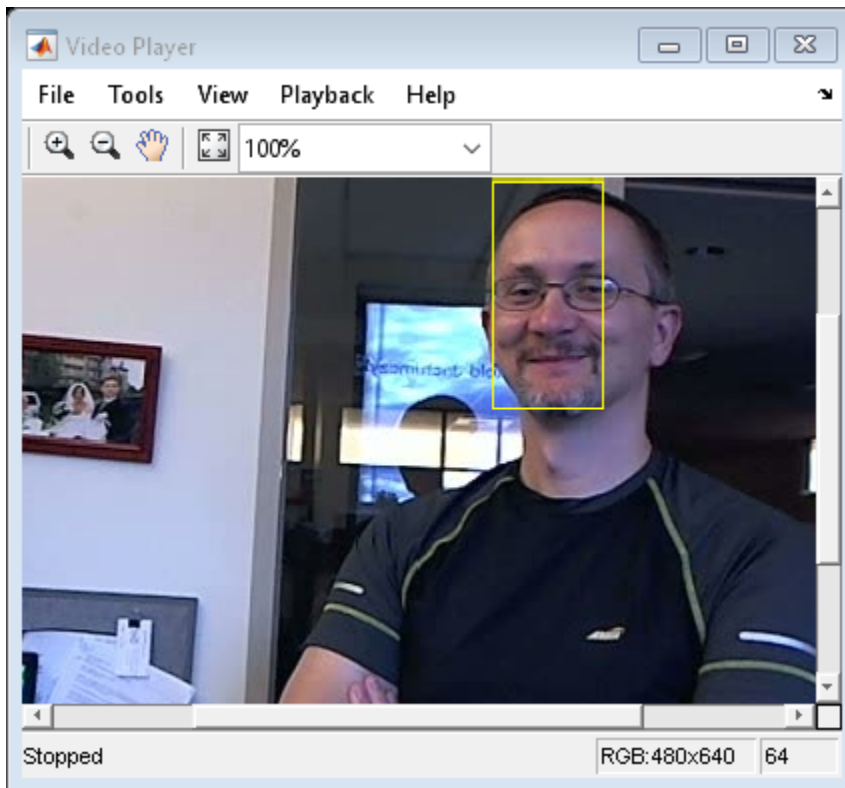
    % Insert a bounding box around the object being tracked
    videoOut = insertObjectAnnotation(videoFrame,'rectangle',bbox,'Face');

    % Display the annotated video frame using the video player object
    step(videoPlayer, videoOut);

end

% Release resources
release(videoPlayer);

```



Summary

In this example, you created a simple face tracking system that automatically detects and tracks a single face. Try changing the input video and see if you are able to track a face. If you notice poor tracking results, check the Hue channel data to see if there is enough contrast between the face region and the background.

Reference

- [1] G.R. Bradski "Real Time Face and Object Tracking as a Component of a Perceptual User Interface", Proceedings of the 4th IEEE Workshop on Applications of Computer Vision, 1998.
- [2] Viola, Paul A. and Jones, Michael J. "Rapid Object Detection using a Boosted Cascade of Simple Features", IEEE CVPR, 2001.

Face Detection and Tracking Using the KLT Algorithm

This example shows how to automatically detect and track a face using feature points. The approach in this example keeps track of the face even when the person tilts his or her head, or moves toward or away from the camera.

Introduction

Object detection and tracking are important in many computer vision applications including activity recognition, automotive safety, and surveillance. In this example, you will develop a simple face tracking system by dividing the tracking problem into three parts:

- 1 Detect a face
- 2 Identify facial features to track
- 3 Track the face

Detect a Face

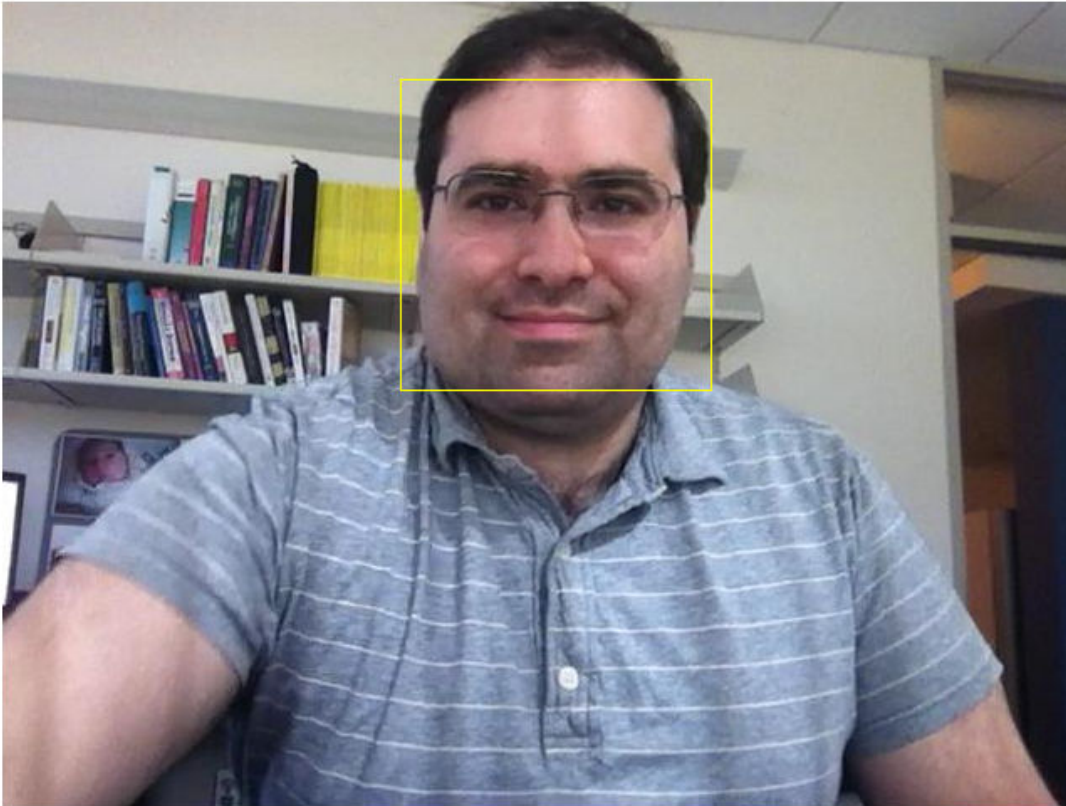
First, you must detect the face. Use the `vision.CascadeObjectDetector` object to detect the location of a face in a video frame. The cascade object detector uses the Viola-Jones detection algorithm and a trained classification model for detection. By default, the detector is configured to detect faces, but it can be used to detect other types of objects.

```
% Create a cascade detector object.
faceDetector = vision.CascadeObjectDetector();

% Read a video frame and run the face detector.
videoReader = VideoReader('tilted_face.avi');
videoFrame   = readFrame(videoReader);
bbox         = step(faceDetector, videoFrame);

% Draw the returned bounding box around the detected face.
videoFrame = insertShape(videoFrame, 'Rectangle', bbox);
figure; imshow(videoFrame); title('Detected face');
```

Detected face



```
% Convert the first box into a list of 4 points
% This is needed to be able to visualize the rotation of the object.
bboxPoints = bbox2points(bbox(1, :));
```

To track the face over time, this example uses the Kanade-Lucas-Tomasi (KLT) algorithm. While it is possible to use the cascade object detector on every frame, it is computationally expensive. It may also fail to detect the face, when the subject turns or tilts his head. This limitation comes from the type of trained classification model used for detection. The example detects the face only once, and then the KLT algorithm tracks the face across the video frames.

Identify Facial Features To Track

The KLT algorithm tracks a set of feature points across the video frames. Once the detection locates the face, the next step in the example identifies feature points that can be reliably tracked. This example uses the standard, "good features to track" proposed by Shi and Tomasi.

Detect feature points in the face region.

```
points = detectMinEigenFeatures(rgb2gray(videoFrame), 'ROI', bbox);
% Display the detected points.
```

```
figure, imshow(videoFrame), hold on, title('Detected features');  
plot(points);
```



Initialize a Tracker to Track the Points

With the feature points identified, you can now use the `vision.PointTracker` System object to track them. For each point in the previous frame, the point tracker attempts to find the corresponding point in the current frame. Then the `estimateGeometricTransform2D` function is used to estimate the translation, rotation, and scale between the old points and the new points. This transformation is applied to the bounding box around the face.

Create a point tracker and enable the bidirectional error constraint to make it more robust in the presence of noise and clutter.

```
pointTracker = vision.PointTracker('MaxBidirectionalError', 2);  
  
% Initialize the tracker with the initial point locations and the initial  
% video frame.  
points = points.Location;  
initialize(pointTracker, points, videoFrame);
```


Initialize a Video Player to Display the Results

Create a video player object for displaying video frames.

```
videoPlayer = vision.VideoPlayer('Position',...
    [100 100 [size(videoFrame, 2), size(videoFrame, 1)]+30]);
```

Track the Face

Track the points from frame to frame, and use `estimateGeometricTransform2D` function to estimate the motion of the face.

Make a copy of the points to be used for computing the geometric transformation between the points in the previous and the current frames

```
oldPoints = points;

while hasFrame(videoReader)
    % get the next frame
    videoFrame = readFrame(videoReader);

    % Track the points. Note that some points may be lost.
    [points, isFound] = step(pointTracker, videoFrame);
    visiblePoints = points(isFound, :);
    oldInliers = oldPoints(isFound, :);

    if size(visiblePoints, 1) >= 2 % need at least 2 points

        % Estimate the geometric transformation between the old points
        % and the new points and eliminate outliers
        [xform, inlierIdx] = estimateGeometricTransform2D(...
            oldInliers, visiblePoints, 'similarity', 'MaxDistance', 4);
        oldInliers = oldInliers(inlierIdx, :);
        visiblePoints = visiblePoints(inlierIdx, :);

        % Apply the transformation to the bounding box points
        bboxPoints = transformPointsForward(xform, bboxPoints);

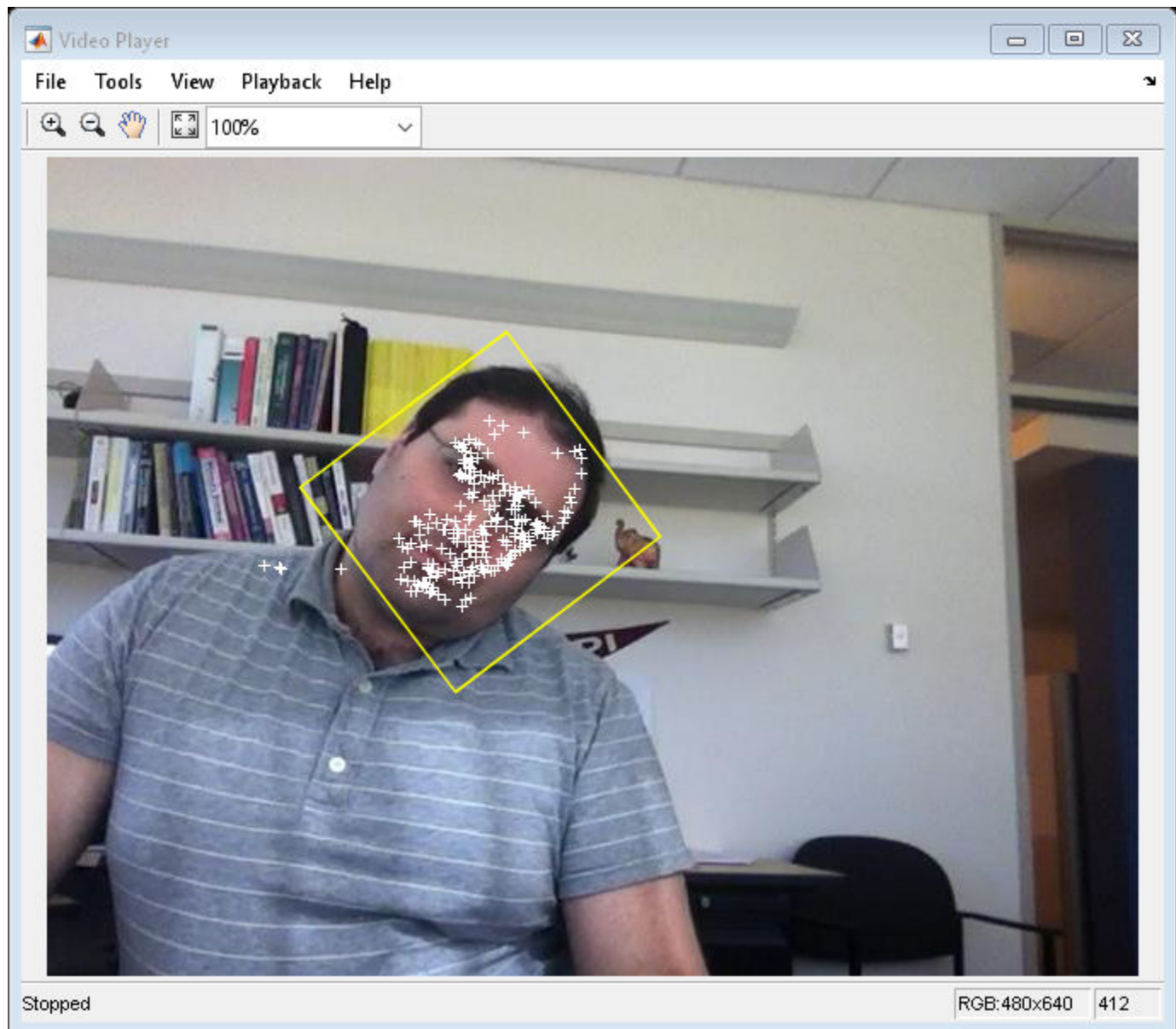
        % Insert a bounding box around the object being tracked
        bboxPolygon = reshape(bboxPoints', 1, []);
        videoFrame = insertShape(videoFrame, 'Polygon', bboxPolygon, ...
            'LineWidth', 2);

        % Display tracked points
        videoFrame = insertMarker(videoFrame, visiblePoints, '+', ...
            'Color', 'white');

        % Reset the points
        oldPoints = visiblePoints;
        setPoints(pointTracker, oldPoints);
    end

    % Display the annotated video frame using the video player object
    step(videoPlayer, videoFrame);
end

% Clean up
release(videoPlayer);
```



```
release(pointTracker);
```

Summary

In this example, you created a simple face tracking system that automatically detects and tracks a single face. Try changing the input video, and see if you are still able to detect and track a face. Make sure the person is facing the camera in the initial frame for the detection step.

References

Viola, Paul A. and Jones, Michael J. "Rapid Object Detection using a Boosted Cascade of Simple Features", IEEE CVPR, 2001.

Bruce D. Lucas and Takeo Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. International Joint Conference on Artificial Intelligence, 1981.

Carlo Tomasi and Takeo Kanade. Detection and Tracking of Point Features. Carnegie Mellon University Technical Report CMU-CS-91-132, 1991.

Jianbo Shi and Carlo Tomasi. Good Features to Track. IEEE Conference on Computer Vision and Pattern Recognition, 1994.

Zdenek Kalal, Krystian Mikolajczyk and Jiri Matas. Forward-Backward Error: Automatic Detection of Tracking Failures. International Conference on Pattern Recognition, 2010

Face Detection and Tracking Using Live Video Acquisition

This example shows how to automatically detect and track a face in a live video stream, using the KLT algorithm.

Overview

Object detection and tracking are important in many computer vision applications including activity recognition, automotive safety, and surveillance. In this example you will develop a simple system for tracking a single face in a live video stream captured by a webcam. MATLAB provides webcam support through a Hardware Support Package, which you will need to download and install in order to run this example. The support package is available via the Support Package Installer.

The face tracking system in this example can be in one of two modes: detection or tracking. In the detection mode you can use a `vision.CascadeObjectDetector` object to detect a face in the current frame. If a face is detected, then you must detect corner points on the face, initialize a `vision.PointTracker` object, and then switch to the tracking mode.

In the tracking mode, you must track the points using the point tracker. As you track the points, some of them will be lost because of occlusion. If the number of points being tracked falls below a threshold, that means that the face is no longer being tracked. You must then switch back to the detection mode to try to re-acquire the face.

Setup

Create objects for detecting faces, tracking points, acquiring and displaying video frames.

```
% Create the face detector object.
faceDetector = vision.CascadeObjectDetector();

% Create the point tracker object.
pointTracker = vision.PointTracker('MaxBidirectionalError', 2);

% Create the webcam object.
cam = webcam();

% Capture one frame to get its size.
videoFrame = snapshot(cam);
frameSize = size(videoFrame);

% Create the video player object.
videoPlayer = vision.VideoPlayer('Position', [100 100 [frameSize(2), frameSize(1)]+30]);
```

Detection and Tracking

Capture and process video frames from the webcam in a loop to detect and track a face. The loop will run for 400 frames or until the video player window is closed.

```
runLoop = true;
numPts = 0;
frameCount = 0;

while runLoop && frameCount < 400
    % Get the next frame.
    videoFrame = snapshot(cam);
```

```

videoFrameGray = rgb2gray(videoFrame);
frameCount = frameCount + 1;

if numPts < 10
    % Detection mode.
    bbox = faceDetector.step(videoFrameGray);

    if ~isempty(bbox)
        % Find corner points inside the detected region.
        points = detectMinEigenFeatures(videoFrameGray, 'ROI', bbox(1, :));

        % Re-initialize the point tracker.
        xyPoints = points.Location;
        numPts = size(xyPoints,1);
        release(pointTracker);
        initialize(pointTracker, xyPoints, videoFrameGray);

        % Save a copy of the points.
        oldPoints = xyPoints;

        % Convert the rectangle represented as [x, y, w, h] into an
        % M-by-2 matrix of [x,y] coordinates of the four corners. This
        % is needed to be able to transform the bounding box to display
        % the orientation of the face.
        bboxPoints = bbox2points(bbox(1, :));

        % Convert the box corners into the [x1 y1 x2 y2 x3 y3 x4 y4]
        % format required by insertShape.
        bboxPolygon = reshape(bboxPoints', 1, []);

        % Display a bounding box around the detected face.
        videoFrame = insertShape(videoFrame, 'Polygon', bboxPolygon, 'LineWidth', 3);

        % Display detected corners.
        videoFrame = insertMarker(videoFrame, xyPoints, '+', 'Color', 'white');
    end
else
    % Tracking mode.
    [xyPoints, isFound] = step(pointTracker, videoFrameGray);
    visiblePoints = xyPoints(isFound, :);
    oldInliers = oldPoints(isFound, :);

    numPts = size(visiblePoints, 1);

    if numPts >= 10
        % Estimate the geometric transformation between the old points
        % and the new points.
        [xform, inlierIdx] = estimateGeometricTransform2D(...
            oldInliers, visiblePoints, 'similarity', 'MaxDistance', 4);
        oldInliers = oldInliers(inlierIdx, :);
        visiblePoints = visiblePoints(inlierIdx, :);

        % Apply the transformation to the bounding box.
        bboxPoints = transformPointsForward(xform, bboxPoints);

        % Convert the box corners into the [x1 y1 x2 y2 x3 y3 x4 y4]
        % format required by insertShape.

```

```
        bboxPolygon = reshape(bboxPoints', 1, []);

        % Display a bounding box around the face being tracked.
        videoFrame = insertShape(videoFrame, 'Polygon', bboxPolygon, 'LineWidth', 3);

        % Display tracked points.
        videoFrame = insertMarker(videoFrame, visiblePoints, '+', 'Color', 'white');

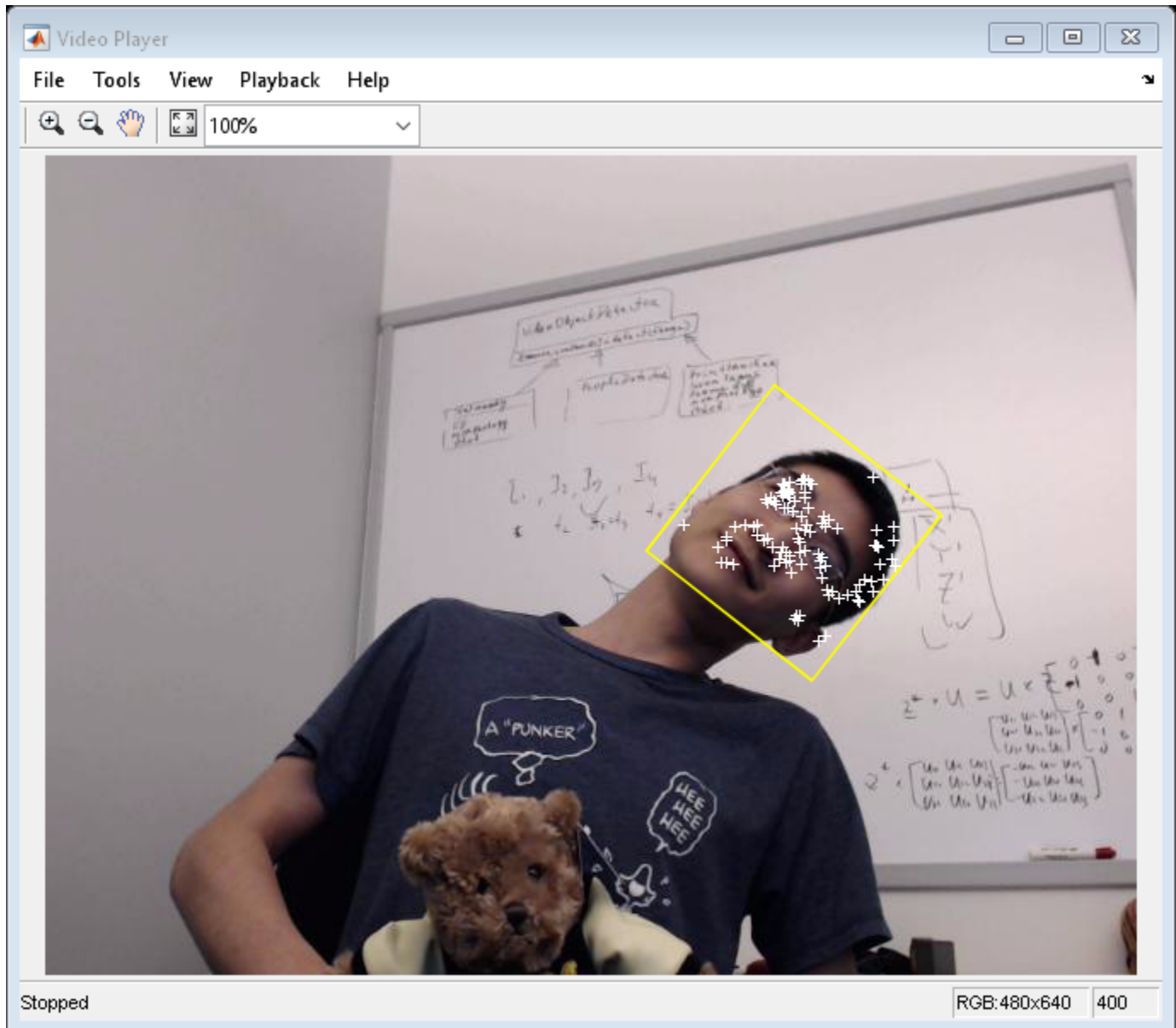
        % Reset the points.
        oldPoints = visiblePoints;
        setPoints(pointTracker, oldPoints);
    end

end

% Display the annotated video frame using the video player object.
step(videoPlayer, videoFrame);

% Check whether the video player window has been closed.
runLoop = isOpen(videoPlayer);
end

% Clean up.
clear cam;
release(videoPlayer);
release(pointTracker);
release(faceDetector);
```



References

Viola, Paul A. and Jones, Michael J. "Rapid Object Detection using a Boosted Cascade of Simple Features", IEEE CVPR, 2001.

Bruce D. Lucas and Takeo Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. International Joint Conference on Artificial Intelligence, 1981.

Carlo Tomasi and Takeo Kanade. Detection and Tracking of Point Features. Carnegie Mellon University Technical Report CMU-CS-91-132, 1991.

Jianbo Shi and Carlo Tomasi. Good Features to Track. IEEE Conference on Computer Vision and Pattern Recognition, 1994.

Zdenek Kalal, Krystian Mikolajczyk and Jiri Matas. Forward-Backward Error: Automatic Detection of Tracking Failures. International Conference on Pattern Recognition, 2010

Motion-Based Multiple Object Tracking

This example shows how to perform automatic detection and motion-based tracking of moving objects in a video from a stationary camera.

Detection of moving objects and motion-based tracking are important components of many computer vision applications, including activity recognition, traffic monitoring, and automotive safety. The problem of motion-based object tracking can be divided into two parts:

- 1 Detecting moving objects in each frame
- 2 Associating the detections corresponding to the same object over time

The detection of moving objects uses a background subtraction algorithm based on Gaussian mixture models. Morphological operations are applied to the resulting foreground mask to eliminate noise. Finally, blob analysis detects groups of connected pixels, which are likely to correspond to moving objects.

The association of detections to the same object is based solely on motion. The motion of each track is estimated by a Kalman filter. The filter is used to predict the track's location in each frame, and determine the likelihood of each detection being assigned to each track.

Track maintenance becomes an important aspect of this example. In any given frame, some detections may be assigned to tracks, while other detections and tracks may remain unassigned. The assigned tracks are updated using the corresponding detections. The unassigned tracks are marked invisible. An unassigned detection begins a new track.

Each track keeps count of the number of consecutive frames, where it remained unassigned. If the count exceeds a specified threshold, the example assumes that the object left the field of view and it deletes the track.

For more information please see “Multiple Object Tracking” on page 15-2.

This example is a function with the main body at the top and helper routines in the form of nested functions.

```
function MotionBasedMultiObjectTrackingExample()

% Create System objects used for reading video, detecting moving objects,
% and displaying the results.
obj = setupSystemObjects();

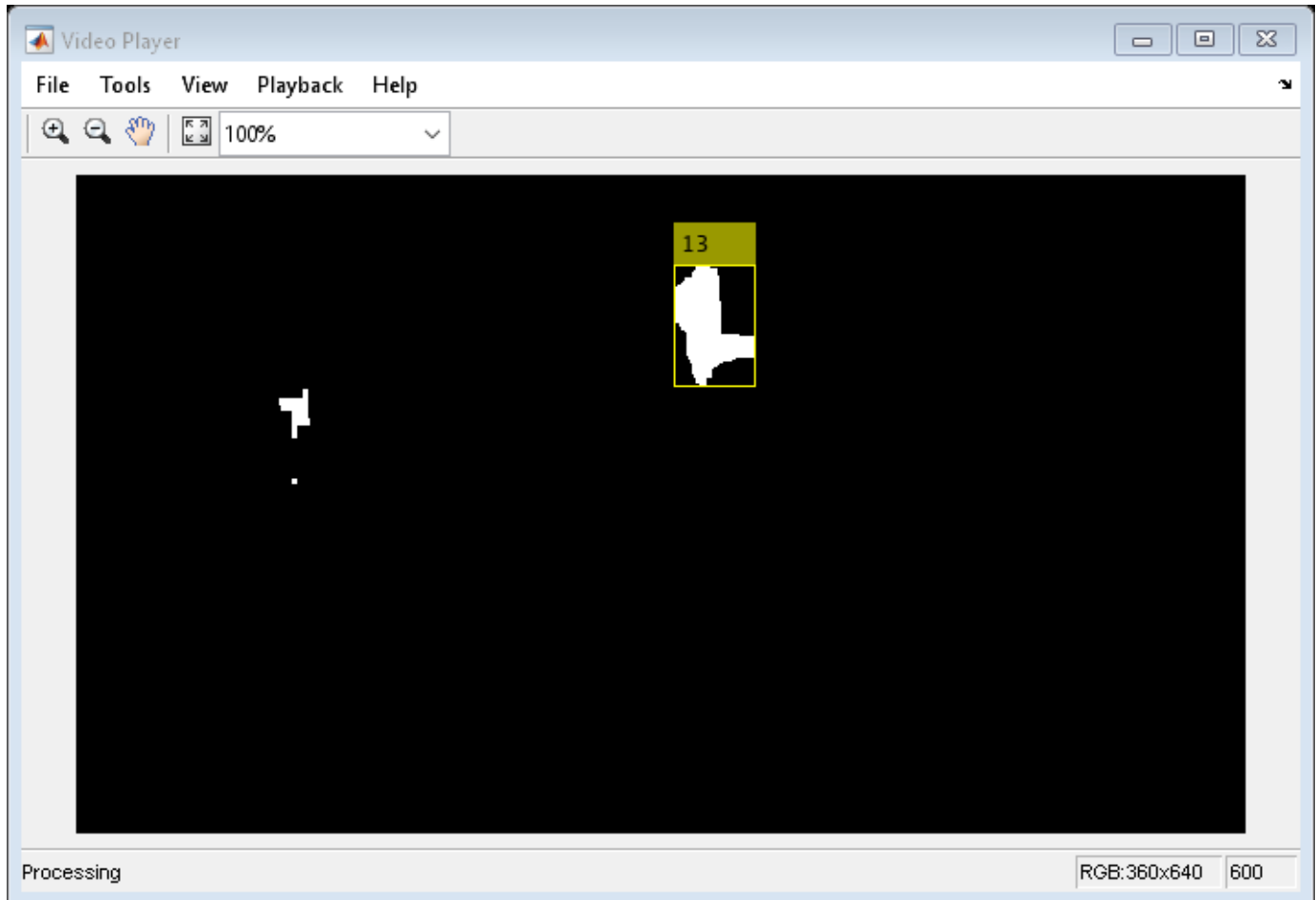
tracks = initializeTracks(); % Create an empty array of tracks.

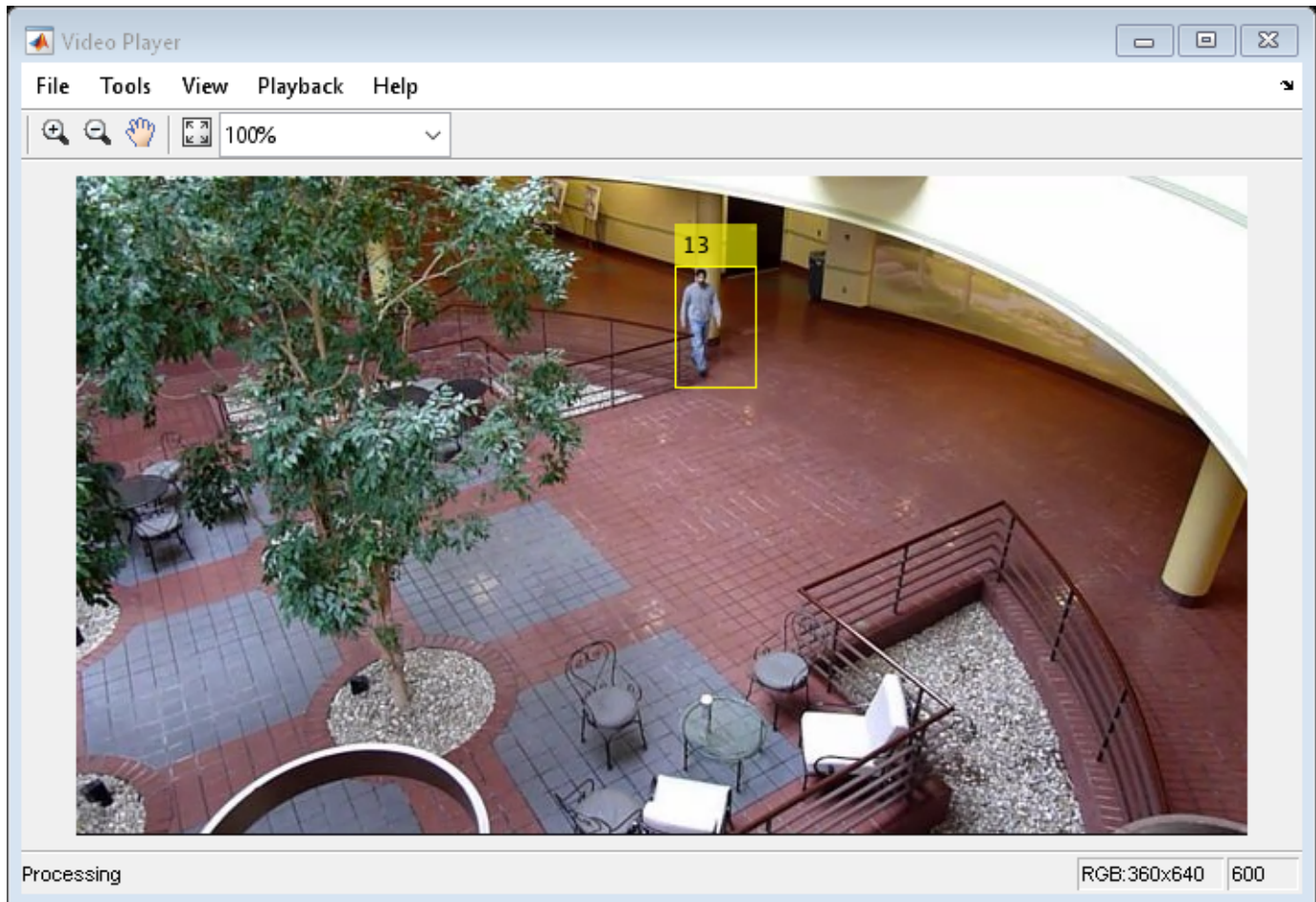
nextId = 1; % ID of the next track

% Detect moving objects, and track them across video frames.
while hasFrame(obj.reader)
    frame = readFrame(obj.reader);
    [centroids, bboxes, mask] = detectObjects(frame);
    predictNewLocationsOfTracks();
    [assignments, unassignedTracks, unassignedDetections] = ...
        detectionToTrackAssignment();

    updateAssignedTracks();
    updateUnassignedTracks();
```

```
deleteLostTracks();  
createNewTracks();  
  
displayTrackingResults();  
end
```





Create System Objects

Create System objects used for reading the video frames, detecting foreground objects, and displaying results.

```
function obj = setupSystemObjects()
    % Initialize Video I/O
    % Create objects for reading a video from a file, drawing the tracked
    % objects in each frame, and playing the video.

    % Create a video reader.
    obj.reader = VideoReader('atrium.mp4');

    % Create two video players, one to display the video,
    % and one to display the foreground mask.
    obj.maskPlayer = vision.VideoPlayer('Position', [740, 400, 700, 400]);
    obj.videoPlayer = vision.VideoPlayer('Position', [20, 400, 700, 400]);

    % Create System objects for foreground detection and blob analysis

    % The foreground detector is used to segment moving objects from
    % the background. It outputs a binary mask, where the pixel value
    % of 1 corresponds to the foreground and the value of 0 corresponds
    % to the background.
```

```

obj.detector = vision.ForegroundDetector('NumGaussians', 3, ...
    'NumTrainingFrames', 40, 'MinimumBackgroundRatio', 0.7);

% Connected groups of foreground pixels are likely to correspond to moving
% objects. The blob analysis System object is used to find such groups
% (called 'blobs' or 'connected components'), and compute their
% characteristics, such as area, centroid, and the bounding box.

obj.blobAnalyser = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
    'AreaOutputPort', true, 'CentroidOutputPort', true, ...
    'MinimumBlobArea', 400);
end

```

Initialize Tracks

The `initializeTracks` function creates an array of tracks, where each track is a structure representing a moving object in the video. The purpose of the structure is to maintain the state of a tracked object. The state consists of information used for detection to track assignment, track termination, and display.

The structure contains the following fields:

- `id` : the integer ID of the track
- `bbox` : the current bounding box of the object; used for display
- `kalmanFilter` : a Kalman filter object used for motion-based tracking
- `age` : the number of frames since the track was first detected
- `totalVisibleCount` : the total number of frames in which the track was detected (visible)
- `consecutiveInvisibleCount` : the number of consecutive frames for which the track was not detected (invisible).

Noisy detections tend to result in short-lived tracks. For this reason, the example only displays an object after it was tracked for some number of frames. This happens when `totalVisibleCount` exceeds a specified threshold.

When no detections are associated with a track for several consecutive frames, the example assumes that the object has left the field of view and deletes the track. This happens when `consecutiveInvisibleCount` exceeds a specified threshold. A track may also get deleted as noise if it was tracked for a short time, and marked invisible for most of the frames.

```

function tracks = initializeTracks()
% create an empty array of tracks
tracks = struct(...
    'id', {}, ...
    'bbox', {}, ...
    'kalmanFilter', {}, ...
    'age', {}, ...
    'totalVisibleCount', {}, ...
    'consecutiveInvisibleCount', {});
end

```

Detect Objects

The `detectObjects` function returns the centroids and the bounding boxes of the detected objects. It also returns the binary mask, which has the same size as the input frame. Pixels with a value of 1 correspond to the foreground, and pixels with a value of 0 correspond to the background.

The function performs motion segmentation using the foreground detector. It then performs morphological operations on the resulting binary mask to remove noisy pixels and to fill the holes in the remaining blobs.

```
function [centroids, bboxes, mask] = detectObjects(frame)

    % Detect foreground.
    mask = obj.detector.step(frame);

    % Apply morphological operations to remove noise and fill in holes.
    mask = imopen(mask, strel('rectangle', [3,3]));
    mask = imclose(mask, strel('rectangle', [15, 15]));
    mask = imfill(mask, 'holes');

    % Perform blob analysis to find connected components.
    [~, centroids, bboxes] = obj.blobAnalyser.step(mask);
end
```

Predict New Locations of Existing Tracks

Use the Kalman filter to predict the centroid of each track in the current frame, and update its bounding box accordingly.

```
function predictNewLocationsOfTracks()
    for i = 1:length(tracks)
        bbox = tracks(i).bbox;

        % Predict the current location of the track.
        predictedCentroid = predict(tracks(i).kalmanFilter);

        % Shift the bounding box so that its center is at
        % the predicted location.
        predictedCentroid = int32(predictedCentroid) - bbox(3:4) / 2;
        tracks(i).bbox = [predictedCentroid, bbox(3:4)];
    end
end
```

Assign Detections to Tracks

Assigning object detections in the current frame to existing tracks is done by minimizing cost. The cost is defined as the negative log-likelihood of a detection corresponding to a track.

The algorithm involves two steps:

Step 1: Compute the cost of assigning every detection to each track using the `distance` method of the `vision.KalmanFilter` System object™. The cost takes into account the Euclidean distance between the predicted centroid of the track and the centroid of the detection. It also includes the confidence of the prediction, which is maintained by the Kalman filter. The results are stored in an $M \times N$ matrix, where M is the number of tracks, and N is the number of detections.

Step 2: Solve the assignment problem represented by the cost matrix using the `assignDetectionsToTracks` function. The function takes the cost matrix and the cost of not assigning any detections to a track.

The value for the cost of not assigning a detection to a track depends on the range of values returned by the `distance` method of the `vision.KalmanFilter`. This value must be tuned experimentally. Setting it too low increases the likelihood of creating a new track, and may result in track fragmentation. Setting it too high may result in a single track corresponding to a series of separate moving objects.

The `assignDetectionsToTracks` function uses the Munkres' version of the Hungarian algorithm to compute an assignment which minimizes the total cost. It returns an $M \times 2$ matrix containing the corresponding indices of assigned tracks and detections in its two columns. It also returns the indices of tracks and detections that remained unassigned.

```
function [assignments, unassignedTracks, unassignedDetections] = ...
    detectionToTrackAssignment()

    nTracks = length(tracks);
    nDetections = size(centroids, 1);

    % Compute the cost of assigning each detection to each track.
    cost = zeros(nTracks, nDetections);
    for i = 1:nTracks
        cost(i, :) = distance(tracks(i).kalmanFilter, centroids);
    end

    % Solve the assignment problem.
    costOfNonAssignment = 20;
    [assignments, unassignedTracks, unassignedDetections] = ...
        assignDetectionsToTracks(cost, costOfNonAssignment);
end
```

Update Assigned Tracks

The `updateAssignedTracks` function updates each assigned track with the corresponding detection. It calls the `correct` method of `vision.KalmanFilter` to correct the location estimate. Next, it stores the new bounding box, and increases the age of the track and the total visible count by 1. Finally, the function sets the invisible count to 0.

```
function updateAssignedTracks()
    numAssignedTracks = size(assignments, 1);
    for i = 1:numAssignedTracks
        trackIdx = assignments(i, 1);
        detectionIdx = assignments(i, 2);
        centroid = centroids(detectionIdx, :);
        bbox = bboxes(detectionIdx, :);

        % Correct the estimate of the object's location
        % using the new detection.
        correct(tracks(trackIdx).kalmanFilter, centroid);

        % Replace predicted bounding box with detected
        % bounding box.
        tracks(trackIdx).bbox = bbox;

        % Update track's age.
```

```

        tracks(trackIdx).age = tracks(trackIdx).age + 1;

        % Update visibility.
        tracks(trackIdx).totalVisibleCount = ...
            tracks(trackIdx).totalVisibleCount + 1;
        tracks(trackIdx).consecutiveInvisibleCount = 0;
    end
end

```

Update Unassigned Tracks

Mark each unassigned track as invisible, and increase its age by 1.

```

function updateUnassignedTracks()
    for i = 1:length(unassignedTracks)
        ind = unassignedTracks(i);
        tracks(ind).age = tracks(ind).age + 1;
        tracks(ind).consecutiveInvisibleCount = ...
            tracks(ind).consecutiveInvisibleCount + 1;
    end
end

```

Delete Lost Tracks

The `deleteLostTracks` function deletes tracks that have been invisible for too many consecutive frames. It also deletes recently created tracks that have been invisible for too many frames overall.

```

function deleteLostTracks()
    if isempty(tracks)
        return;
    end

    invisibleForTooLong = 20;
    ageThreshold = 8;

    % Compute the fraction of the track's age for which it was visible.
    ages = [tracks(:).age];
    totalVisibleCounts = [tracks(:).totalVisibleCount];
    visibility = totalVisibleCounts ./ ages;

    % Find the indices of 'lost' tracks.
    lostInds = (ages < ageThreshold & visibility < 0.6) | ...
        [tracks(:).consecutiveInvisibleCount] >= invisibleForTooLong;

    % Delete lost tracks.
    tracks = tracks(~lostInds);
end

```

Create New Tracks

Create new tracks from unassigned detections. Assume that any unassigned detection is a start of a new track. In practice, you can use other cues to eliminate noisy detections, such as size, location, or appearance.

```

function createNewTracks()
    centroids = centroids(unassignedDetections, :);
    bboxes = bboxes(unassignedDetections, :);

```

```

for i = 1:size(centroids, 1)

    centroid = centroids(i,:);
    bbox = bboxes(i, :);

    % Create a Kalman filter object.
    kalmanFilter = configureKalmanFilter('ConstantVelocity', ...
        centroid, [200, 50], [100, 25], 100);

    % Create a new track.
    newTrack = struct(...
        'id', nextId, ...
        'bbox', bbox, ...
        'kalmanFilter', kalmanFilter, ...
        'age', 1, ...
        'totalVisibleCount', 1, ...
        'consecutiveInvisibleCount', 0);

    % Add it to the array of tracks.
    tracks(end + 1) = newTrack;

    % Increment the next id.
    nextId = nextId + 1;
end
end

```

Display Tracking Results

The `displayTrackingResults` function draws a bounding box and label ID for each track on the video frame and the foreground mask. It then displays the frame and the mask in their respective video players.

```

function displayTrackingResults()
    % Convert the frame and the mask to uint8 RGB.
    frame = im2uint8(frame);
    mask = uint8(repmat(mask, [1, 1, 3])) .* 255;

    minVisibleCount = 8;
    if ~isempty(tracks)

        % Noisy detections tend to result in short-lived tracks.
        % Only display tracks that have been visible for more than
        % a minimum number of frames.
        reliableTrackInds = ...
            [tracks(:).totalVisibleCount] > minVisibleCount;
        reliableTracks = tracks(reliableTrackInds);

        % Display the objects. If an object has not been detected
        % in this frame, display its predicted bounding box.
        if ~isempty(reliableTracks)
            % Get bounding boxes.
            bboxes = cat(1, reliableTracks.bbox);

            % Get ids.
            ids = int32([reliableTracks(:).id]);

            % Create labels for objects indicating the ones for
            % which we display the predicted rather than the actual

```



```

    % location.
    labels = cellstr(int2str(ids'));
    predictedTrackInds = ...
        [reliableTracks(:).consecutiveInvisibleCount] > 0;
    isPredicted = cell(size(labels));
    isPredicted(predictedTrackInds) = {' predicted'};
    labels = strcat(labels, isPredicted);

    % Draw the objects on the frame.
    frame = insertObjectAnnotation(frame, 'rectangle', ...
        bboxes, labels);

    % Draw the objects on the mask.
    mask = insertObjectAnnotation(mask, 'rectangle', ...
        bboxes, labels);
end
end

% Display the mask and the frame.
obj.maskPlayer.step(mask);
obj.videoPlayer.step(frame);
end

```

Summary

This example created a motion-based system for detecting and tracking multiple moving objects. Try using a different video to see if you are able to detect and track objects. Try modifying the parameters for the detection, assignment, and deletion steps.

The tracking in this example was solely based on motion with the assumption that all objects move in a straight line with constant speed. When the motion of an object significantly deviates from this model, the example may produce tracking errors. Notice the mistake in tracking the person labeled #12, when he is occluded by the tree.

The likelihood of tracking errors can be reduced by using a more complex motion model, such as constant acceleration, or by using multiple Kalman filters for every object. Also, you can incorporate other cues for associating detections over time, such as size, shape, and color.

end

Tracking Pedestrians from a Moving Car

This example shows how to track pedestrians using a camera mounted in a moving car.

Overview

This example shows how to perform automatic detection and tracking of people in a video from a moving camera. It demonstrates the flexibility of a tracking system adapted to a moving camera, which is ideal for automotive safety applications. Unlike the stationary camera example, The Motion-Based Multiple Object Tracking, this example contains several additional algorithmic steps. These steps include people detection, customized non-maximum suppression, and heuristics to identify and eliminate false alarm tracks. For more information please see “Multiple Object Tracking” on page 15-2.

This example is a function with the main body at the top and helper routines in the form of “What Are Nested Functions?” below.

```
function PedestrianTrackingFromMovingCameraExample()

% Create system objects used for reading video, loading prerequisite data file, detecting pedest
videoFile      = 'vippedtracking.mp4';
scaleDataFile  = 'pedScaleTable.mat'; % An auxiliary file that helps to determine the size of a

obj = setupSystemObjects(videoFile, scaleDataFile);

detector = peopleDetectorACF('caltech');

% Create an empty array of tracks.
tracks = initializeTracks();

% ID of the next track.
nextId = 1;

% Set the global parameters.
option.ROI      = [40 95 400 140]; % A rectangle [x, y, w, h] that limits the proces
option.scThresh = 0.3;             % A threshold to control the tolerance of error :
option.gatingThresh = 0.9;        % A threshold to reject a candidate match between
option.gatingCost = 100;          % A large value for the assignment cost matrix th
option.costOfNonAssignment = 10;   % A tuning parameter to control the likelihood o
option.timeWindowSize = 16;       % A tuning parameter to specify the number of fra
option.confidenceThresh = 2;      % A threshold to determine if a track is true pos
option.ageThresh = 8;             % A threshold to determine the minimum length rec
option.visThresh = 0.6;          % A threshold to determine the minimum visibility

% Detect people and track them across video frames.
stopFrame = 1629; % stop on an interesting frame with several pedestrians
for fNum = 1:stopFrame
    frame = readFrame(obj.reader);

    [centroids, bboxes, scores] = detectPeople();

    predictNewLocationsOfTracks();

    [assignments, unassignedTracks, unassignedDetections] = ...
        detectionToTrackAssignment();

    updateAssignedTracks();
end
```

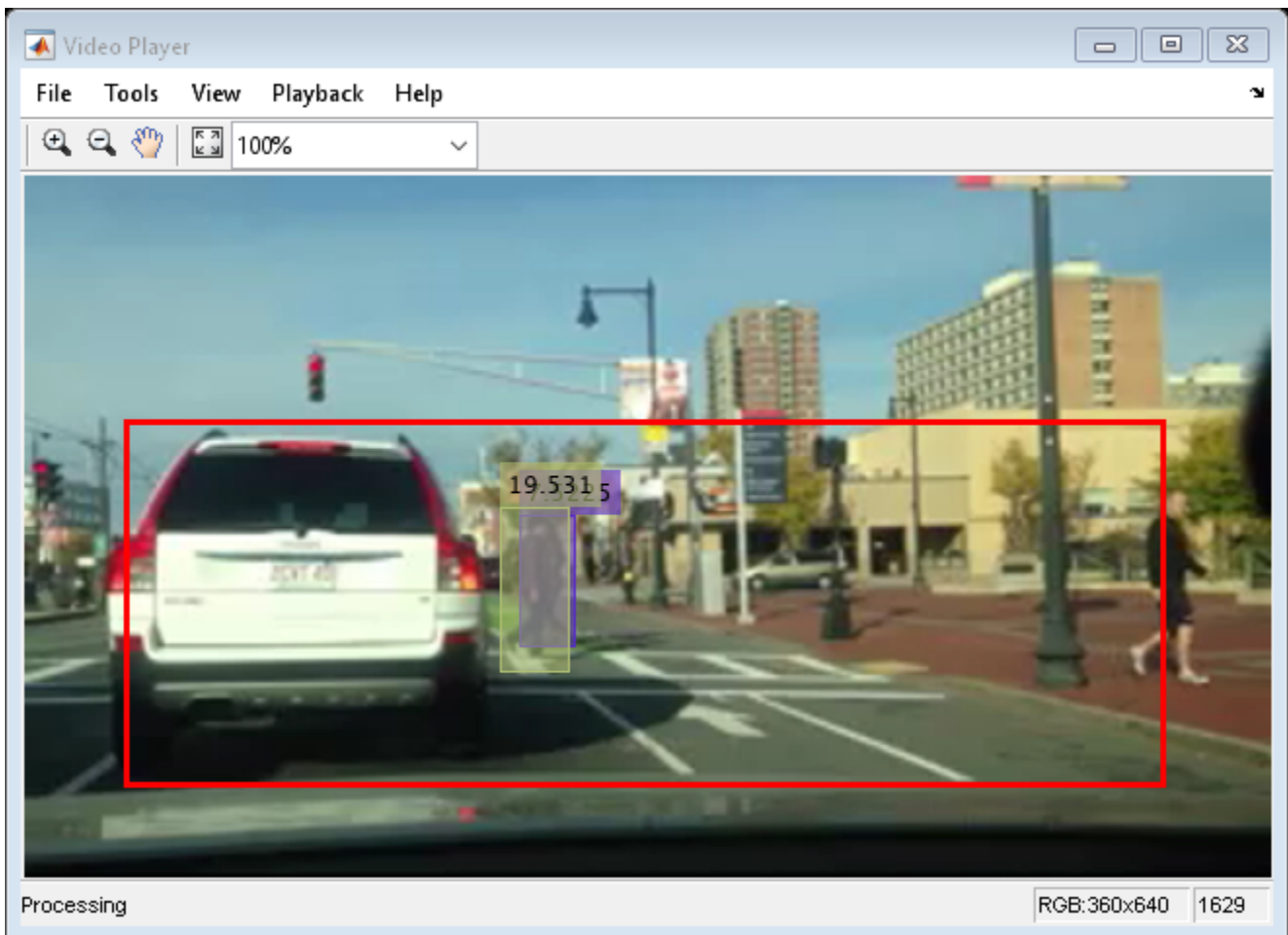
```

updateUnassignedTracks();
deleteLostTracks();
createNewTracks();

displayTrackingResults();

% Exit the loop if the video player figure is closed.
if ~isOpen(obj.videoPlayer)
    break;
end
end
end

```



Auxiliary Input and Global Parameters of the Tracking System

This tracking system requires a data file that contains information that relates the pixel location in the image to the size of the bounding box marking the pedestrian's location. This prior knowledge is stored in a vector `pedScaleTable`. The n -th entry in `pedScaleTable` represents the estimated height of an adult person in pixels. The index n references the approximate Y-coordinate of the pedestrian's feet.

To obtain such a vector, a collection of training images were taken from the same viewpoint and in a similar scene to the testing environment. The training images contained images of pedestrians at varying distances from the camera. Using the Image Labeler app, bounding boxes of the pedestrians

in the images were manually annotated. The height of the bounding boxes together with the location of the pedestrians in the image were used to generate the scale data file through regression. Here is a helper function to show the algorithmic steps to fit the linear regression model:

`helperTableOfScales.m`

There is also a set of global parameters that can be tuned to optimize the tracking performance. You can use the descriptions below to learn about how these parameters affect the tracking performance.

- `ROI` : Region-Of-Interest in the form of $[x, y, w, h]$. It limits the processing area to ground locations.
- `scThresh` : Tolerance threshold for scale estimation. When the difference between the detected scale and the expected scale exceeds the tolerance, the candidate detection is considered to be unrealistic and is removed from the output.
- `gatingThresh` : Gating parameter for the distance measure. When the cost of matching the detected bounding box and the predicted bounding box exceeds the threshold, the system removes the association of the two bounding boxes from tracking consideration.
- `gatingCost` : Value for the assignment cost matrix to discourage the possible tracking to detection assignment.
- `costOfNonAssignment` : Value for the assignment cost matrix for not assigning a detection or a track. Setting it too low increases the likelihood of creating a new track, and may result in track fragmentation. Setting it too high may result in a single track corresponding to a series of separate moving objects.
- `timeWindowSize` : Number of frames required to estimate the confidence of the track.
- `confidenceThresh` : Confidence threshold to determine if the track is a true positive.
- `ageThresh` : Minimum length of a track being a true positive.
- `visThresh` : Minimum visibility threshold to determine if the track is a true positive.

Create System Objects for the Tracking System Initialization

The `setupSystemObjects` function creates system objects used for reading and displaying the video frames and loads the scale data file.

The `pedScaleTable` vector, which is stored in the scale data file, encodes our prior knowledge of the target and the scene. Once you have the regressor trained from your samples, you can compute the expected height at every possible Y-position in the image. These values are stored in the vector. The n -th entry in `pedScaleTable` represents our estimated height of an adult person in pixels. The index n references the approximate Y-coordinate of the pedestrian's feet.

```
function obj = setupSystemObjects(videoFile,scaleDataFile)
    % Initialize Video I/O
    % Create objects for reading a video from a file, drawing the
    % detected and tracked people in each frame, and playing the video.

    % Create a video file reader.
    obj.reader = VideoReader(videoFile);

    % Create a video player.
    obj.videoPlayer = vision.VideoPlayer('Position', [29, 597, 643, 386]);

    % Load the scale data file
    ld = load(scaleDataFile, 'pedScaleTable');
    obj.pedScaleTable = ld.pedScaleTable;
end
```

Initialize Tracks

The `initializeTracks` function creates an array of tracks, where each track is a structure representing a moving object in the video. The purpose of the structure is to maintain the state of a tracked object. The state consists of information used for detection-to-track assignment, track termination, and display.

The structure contains the following fields:

- `id` : An integer ID of the track.
- `color` : The color of the track for display purpose.
- `bboxes` : A N-by-4 matrix to represent the bounding boxes of the object with the current box at the last row. Each row has a form of [x, y, width, height].
- `scores` : An N-by-1 vector to record the classification score from the person detector with the current detection score at the last row.
- `kalmanFilter` : A Kalman filter object used for motion-based tracking. We track the center point of the object in image;
- `age` : The number of frames since the track was initialized.
- `totalVisibleCount` : The total number of frames in which the object was detected (visible).
- `confidence` : A pair of two numbers to represent how confident we trust the track. It stores the maximum and the average detection scores in the past within a predefined time window.
- `predPosition` : The predicted bounding box in the next frame.

```
function tracks = initializeTracks()
    % Create an empty array of tracks
    tracks = struct(...
        'id', {}, ...
        'color', {}, ...
        'bboxes', {}, ...
        'scores', {}, ...
        'kalmanFilter', {}, ...
        'age', {}, ...
        'totalVisibleCount', {}, ...
        'confidence', {}, ...
        'predPosition', {});
end
```

Detect People

The `detectPeople` function returns the centroids, the bounding boxes, and the classification scores of the detected people. It performs filtering and non-maximum suppression on the raw output of the detector returned by `peopleDetectorACF`.

- `centroids` : An N-by-2 matrix with each row in the form of [x,y].
- `bboxes` : An N-by-4 matrix with each row in the form of [x, y, width, height].
- `scores` : An N-by-1 vector with each element is the classification score at the corresponding frame.

```
function [centroids, bboxes, scores] = detectPeople()
    % Resize the image to increase the resolution of the pedestrian.
    % This helps detect people further away from the camera.
    resizeMode = 1.5;
```

```

frame = imresize(frame, resizeMode, 'Antialiasing', false);

% Run ACF people detector within a region of interest to produce
% detection candidates.
[bboxes, scores] = detect(detector, frame, option.ROI, ...
    'WindowStride', 2, ...
    'NumScaleLevels', 4, ...
    'SelectStrongest', false);

% Look up the estimated height of a pedestrian based on location of their feet.
height = bboxes(:, 4) / resizeMode;
y = (bboxes(:, 2) - 1) / resizeMode + 1;
yfoot = min(length(obj.pedScaleTable), round(y + height));
estHeight = obj.pedScaleTable(yfoot);

% Remove detections whose size deviates from the expected size,
% provided by the calibrated scale estimation.
invalid = abs(estHeight - height) > estHeight * option.scThresh;
bboxes(invalid, :) = [];
scores(invalid, :) = [];

% Apply non-maximum suppression to select the strongest bounding boxes.
[bboxes, scores] = selectStrongestBbox(bboxes, scores, ...
    'RatioType', 'Min', 'OverlapThreshold', 0.6);

% Compute the centroids
if isempty(bboxes)
    centroids = [];
else
    centroids = [(bboxes(:, 1) + bboxes(:, 3) / 2), ...
        (bboxes(:, 2) + bboxes(:, 4) / 2)];
end
end

```

Predict New Locations of Existing Tracks

Use the Kalman filter to predict the centroid of each track in the current frame, and update its bounding box accordingly. We take the width and height of the bounding box in previous frame as our current prediction of the size.

```

function predictNewLocationsOfTracks()
    for i = 1:length(tracks)
        % Get the last bounding box on this track.
        bbox = tracks(i).bboxes(end, :);

        % Predict the current location of the track.
        predictedCentroid = predict(tracks(i).kalmanFilter);

        % Shift the bounding box so that its center is at the predicted location.
        tracks(i).predPosition = [predictedCentroid - bbox(3:4)/2, bbox(3:4)];
    end
end

```

Assign Detections to Tracks

Assigning object detections in the current frame to existing tracks is done by minimizing cost. The cost is computed using the `bboxOverlapRatio` function, and is the overlap ratio between the predicted bounding box and the detected bounding box. In this example, we assume the person will

move gradually in consecutive frames due to the high frame rate of the video and the low motion speed of a person.

The algorithm involves two steps:

Step 1: Compute the cost of assigning every detection to each track using the `bboxOverlapRatio` measure. As people move towards or away from the camera, their motion will not be accurately described by the centroid point alone. The cost takes into account the distance on the image plane as well as the scale of the bounding boxes. This prevents assigning detections far away from the camera to tracks closer to the camera, even if their centroids coincide. The choice of this cost function will ease the computation without resorting to a more sophisticated dynamic model. The results are stored in an $M \times N$ matrix, where M is the number of tracks, and N is the number of detections.

Step 2: Solve the assignment problem represented by the cost matrix using the `assignDetectionsToTracks` function. The function takes the cost matrix and the cost of not assigning any detections to a track.

The value for the cost of not assigning a detection to a track depends on the range of values returned by the cost function. This value must be tuned experimentally. Setting it too low increases the likelihood of creating a new track, and may result in track fragmentation. Setting it too high may result in a single track corresponding to a series of separate moving objects.

The `assignDetectionsToTracks` function uses the Munkres' version of the Hungarian algorithm to compute an assignment which minimizes the total cost. It returns an $M \times 2$ matrix containing the corresponding indices of assigned tracks and detections in its two columns. It also returns the indices of tracks and detections that remained unassigned.

```
function [assignments, unassignedTracks, unassignedDetections] = ...
    detectionToTrackAssignment()

    % Compute the overlap ratio between the predicted boxes and the
    % detected boxes, and compute the cost of assigning each detection
    % to each track. The cost is minimum when the predicted bbox is
    % perfectly aligned with the detected bbox (overlap ratio is one)
    predBboxes = reshape([tracks(:).predPosition], 4, []);
    cost = 1 - bboxOverlapRatio(predBboxes, bboxes);

    % Force the optimization step to ignore some matches by
    % setting the associated cost to be a large number. Note that this
    % number is different from the 'costOfNonAssignment' below.
    % This is useful when gating (removing unrealistic matches)
    % technique is applied.
    cost(cost > option.gatingThresh) = 1 + option.gatingCost;

    % Solve the assignment problem.
    [assignments, unassignedTracks, unassignedDetections] = ...
        assignDetectionsToTracks(cost, option.costOfNonAssignment);
end
```

Update Assigned Tracks

The `updateAssignedTracks` function updates each assigned track with the corresponding detection. It calls the correct method of `vision.KalmanFilter` to correct the location estimate. Next, it stores the new bounding box by taking the average of the size of recent (up to) 4 boxes, and increases the age of the track and the total visible count by 1. Finally, the function adjusts our confidence score for the track based on the previous detection scores.

```

function updateAssignedTracks()
    numAssignedTracks = size(assignments, 1);
    for i = 1:numAssignedTracks
        trackIdx = assignments(i, 1);
        detectionIdx = assignments(i, 2);

        centroid = centroids(detectionIdx, :);
        bbox = bboxes(detectionIdx, :);

        % Correct the estimate of the object's location
        % using the new detection.
        correct(tracks(trackIdx).kalmanFilter, centroid);

        % Stabilize the bounding box by taking the average of the size
        % of recent (up to) 4 boxes on the track.
        T = min(size(tracks(trackIdx).bboxes,1), 4);
        w = mean([tracks(trackIdx).bboxes(end-T+1:end, 3); bbox(3)]);
        h = mean([tracks(trackIdx).bboxes(end-T+1:end, 4); bbox(4)]);
        tracks(trackIdx).bboxes(end+1, :) = [centroid - [w, h]/2, w, h];

        % Update track's age.
        tracks(trackIdx).age = tracks(trackIdx).age + 1;

        % Update track's score history
        tracks(trackIdx).scores = [tracks(trackIdx).scores; scores(detectionIdx)];

        % Update visibility.
        tracks(trackIdx).totalVisibleCount = ...
            tracks(trackIdx).totalVisibleCount + 1;

        % Adjust track confidence score based on the maximum detection
        % score in the past 'timeWindowSize' frames.
        T = min(option.timeWindowSize, length(tracks(trackIdx).scores));
        score = tracks(trackIdx).scores(end-T+1:end);
        tracks(trackIdx).confidence = [max(score), mean(score)];
    end
end

```

Update Unassigned Tracks

The `updateUnassignedTracks` function marks each unassigned track as invisible, increases its age by 1, and appends the predicted bounding box to the track. The confidence is set to zero since we are not sure why it was not assigned to a track.

```

function updateUnassignedTracks()
    for i = 1:length(unassignedTracks)
        idx = unassignedTracks(i);
        tracks(idx).age = tracks(idx).age + 1;
        tracks(idx).bboxes = [tracks(idx).bboxes; tracks(idx).predPosition];
        tracks(idx).scores = [tracks(idx).scores; 0];

        % Adjust track confidence score based on the maximum detection
        % score in the past 'timeWindowSize' frames
        T = min(option.timeWindowSize, length(tracks(idx).scores));
        score = tracks(idx).scores(end-T+1:end);
        tracks(idx).confidence = [max(score), mean(score)];
    end
end

```


Delete Lost Tracks

The `deleteLostTracks` function deletes tracks that have been invisible for too many consecutive frames. It also deletes recently created tracks that have been invisible for many frames overall.

Noisy detections tend to result in creation of false tracks. For this example, we remove a track under following conditions:

- The object was tracked for a short time. This typically happens when a false detection shows up for a few frames and a track was initiated for it.
- The track was marked invisible for most of the frames.
- It failed to receive a strong detection within the past few frames, which is expressed as the maximum detection confidence score.

```
function deleteLostTracks()
    if isempty(tracks)
        return;
    end

    % Compute the fraction of the track's age for which it was visible.
    ages = [tracks(:).age]';
    totalVisibleCounts = [tracks(:).totalVisibleCount]';
    visibility = totalVisibleCounts ./ ages;

    % Check the maximum detection confidence score.
    confidence = reshape([tracks(:).confidence], 2, []);
    maxConfidence = confidence(:, 1);

    % Find the indices of 'lost' tracks.
    lostInds = (ages <= option.ageThresh & visibility <= option.visThresh) | ...
        (maxConfidence <= option.confidenceThresh);

    % Delete lost tracks.
    tracks = tracks(~lostInds);
end
```

Create New Tracks

Create new tracks from unassigned detections. Assume that any unassigned detection is a start of a new track. In practice, you can use other cues to eliminate noisy detections, such as size, location, or appearance.

```
function createNewTracks()
    unassignedCentroids = centroids(unassignedDetections, :);
    unassignedBboxes = bboxes(unassignedDetections, :);
    unassignedScores = scores(unassignedDetections);

    for i = 1:size(unassignedBboxes, 1)
        centroid = unassignedCentroids(i, :);
        bbox = unassignedBboxes(i, :);
        score = unassignedScores(i);

        % Create a Kalman filter object.
        kalmanFilter = configureKalmanFilter('ConstantVelocity', ...
            centroid, [2, 1], [5, 5], 100);

        % Create a new track.
    end
```

```

newTrack = struct(...
    'id', nextId, ...
    'color', 255*rand(1,3), ...
    'bboxes', bbox, ...
    'scores', score, ...
    'kalmanFilter', kalmanFilter, ...
    'age', 1, ...
    'totalVisibleCount', 1, ...
    'confidence', [score, score], ...
    'predPosition', bbox);

% Add it to the array of tracks.
tracks(end + 1) = newTrack; %#ok<AGROW>

% Increment the next id.
nextId = nextId + 1;
end
end

```

Display Tracking Results

The `displayTrackingResults` function draws a colored bounding box for each track on the video frame. The level of transparency of the box together with the displayed score indicate the confidence of the detections and tracks.

```

function displayTrackingResults()

displayRatio = 4/3;
frame = imresize(frame, displayRatio);

if ~isempty(tracks),
    ages = [tracks(:).age]';
    confidence = reshape([tracks(:).confidence], 2, [])'';
    maxConfidence = confidence(:, 1);
    avgConfidence = confidence(:, 2);
    opacity = min(0.5, max(0.1, avgConfidence/3));
    noDispInds = (ages < option.ageThresh & maxConfidence < option.confidenceThresh) | .
                (ages < option.ageThresh / 2);

    for i = 1:length(tracks)
        if ~noDispInds(i)

            % scale bounding boxes for display
            bb = tracks(i).bboxes(end, :);
            bb(:,1:2) = (bb(:,1:2)-1)*displayRatio + 1;
            bb(:,3:4) = bb(:,3:4) * displayRatio;

            frame = insertShape(frame, ...
                'FilledRectangle', bb, ...
                'Color', tracks(i).color, ...
                'Opacity', opacity(i));
            frame = insertObjectAnnotation(frame, ...
                'rectangle', bb, ...
                num2str(avgConfidence(i)), ...
                'Color', tracks(i).color);
        end
    end
end

```

```
end  
  
frame = insertShape(frame, 'Rectangle', option.ROI * displayRatio, ...  
                    'Color', [255, 0, 0], 'LineWidth', 3);  
  
step(obj.videoPlayer, frame);  
  
end  
  
end
```

Use Kalman Filter for Object Tracking

This example shows how to use the `vision.KalmanFilter` object and `configureKalmanFilter` function to track objects.

This example is a function with its main body at the top and helper routines in the form of nested functions.

```
function kalmanFilterForTracking
```

Introduction

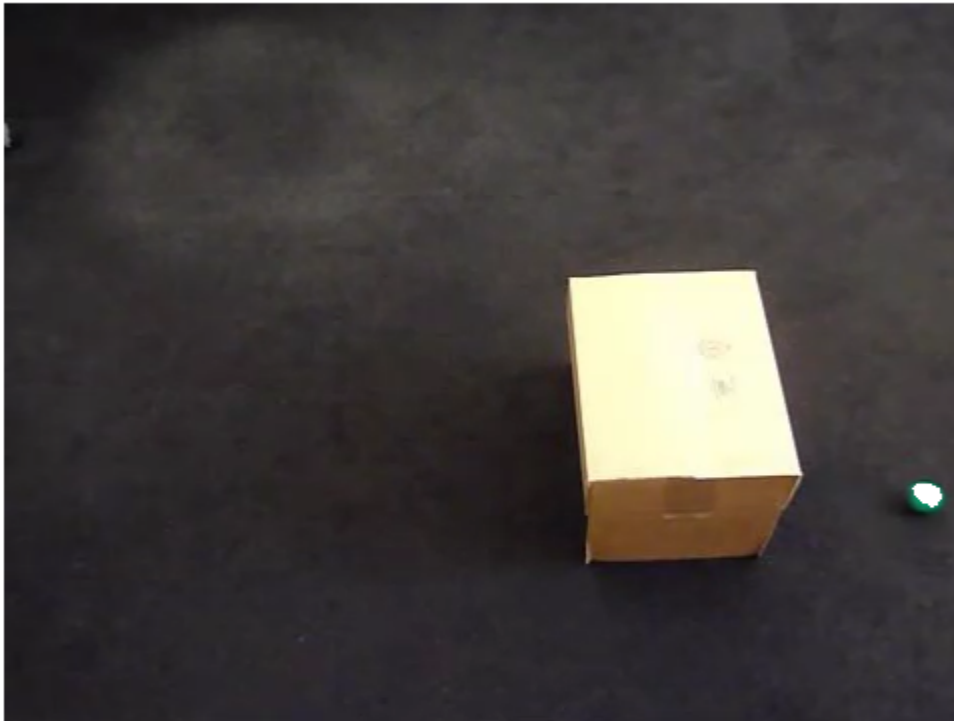
The Kalman filter has many uses, including applications in control, navigation, computer vision, and time series econometrics. This example illustrates how to use the Kalman filter for tracking objects and focuses on three important features:

- Prediction of object's future location
- Reduction of noise introduced by inaccurate detections
- Facilitating the process of association of multiple objects to their tracks

Challenges of Object Tracking

Before showing the use of Kalman filter, let us first examine the challenges of tracking an object in a video. The following video shows a green ball moving from left to right on the floor.

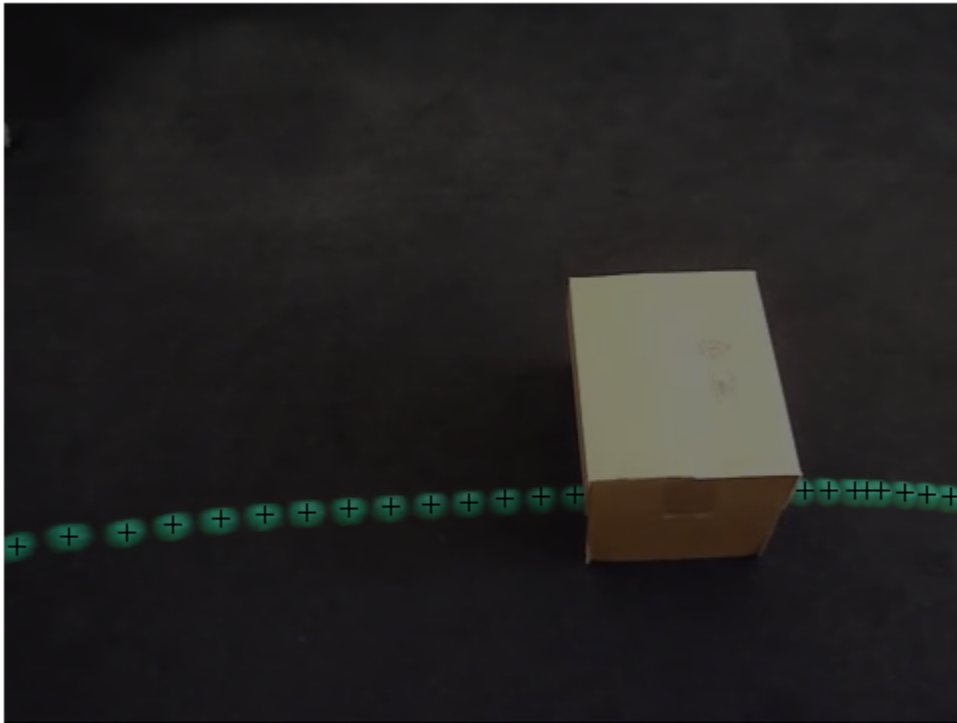
```
showDetections();
```



The white region over the ball highlights the pixels detected using `vision.ForegroundDetector`, which separates moving objects from the background. The background subtraction only finds a portion of the ball because of the low contrast between the ball and the floor. In other words, the detection process is not ideal and introduces noise.

To easily visualize the entire object trajectory, we overlay all video frames onto a single image. The "+" marks indicate the centroids computed using blob analysis.

```
showTrajectory();
```



Two issues can be observed:

- 1 The region's center is usually different from the ball's center. In other words, there is an error in the measurement of the ball's location.
- 2 The location of the ball is not available when it is occluded by the box, i.e. the measurement is missing.

Both of these challenges can be addressed by using the Kalman filter.

Track a Single Object Using Kalman Filter

Using the video which was seen earlier, the `trackSingleObject` function shows you how to:

- Create `vision.KalmanFilter` by using `configureKalmanFilter`
- Use `predict` and `correct` methods in a sequence to eliminate noise present in the tracking system
- Use `predict` method by itself to estimate ball's location when it is occluded by the box

The selection of the Kalman filter parameters can be challenging. The `configureKalmanFilter` function helps simplify this problem. More details about this can be found further in the example.

The `trackSingleObject` function includes nested helper functions. The following top-level variables are used to transfer the data between the nested functions.

```

frame          = []; % A video frame
detectedLocation = []; % The detected location
trackedLocation = []; % The tracked location
label          = ''; % Label for the ball
utilities      = []; % Utilities used to process the video

```

The procedure for tracking a single object is shown below.

```

function trackSingleObject(param)
% Create utilities used for reading video, detecting moving objects,
% and displaying the results.
utilities = createUtilities(param);

isTrackInitialized = false;
while hasFrame(utilities.videoReader)
    frame = readFrame(utilities.videoReader);

    % Detect the ball.
    [detectedLocation, isObjectDetected] = detectObject(frame);

    if ~isTrackInitialized
        if isObjectDetected
            % Initialize a track by creating a Kalman filter when the ball is
            % detected for the first time.
            initialLocation = computeInitialLocation(param, detectedLocation);
            kalmanFilter = configureKalmanFilter(param.motionModel, ...
                initialLocation, param.initialEstimateError, ...
                param.motionNoise, param.measurementNoise);

            isTrackInitialized = true;
            trackedLocation = correct(kalmanFilter, detectedLocation);
            label = 'Initial';
        else
            trackedLocation = [];
            label = '';
        end
    else
        % Use the Kalman filter to track the ball.
        if isObjectDetected % The ball was detected.
            % Reduce the measurement noise by calling predict followed by
            % correct.
            predict(kalmanFilter);
            trackedLocation = correct(kalmanFilter, detectedLocation);
            label = 'Corrected';
        else % The ball was missing.
            % Predict the ball's location.
            trackedLocation = predict(kalmanFilter);
            label = 'Predicted';
        end
    end

    annotateTrackedObject();
end % while

showTrajectory();
end

```

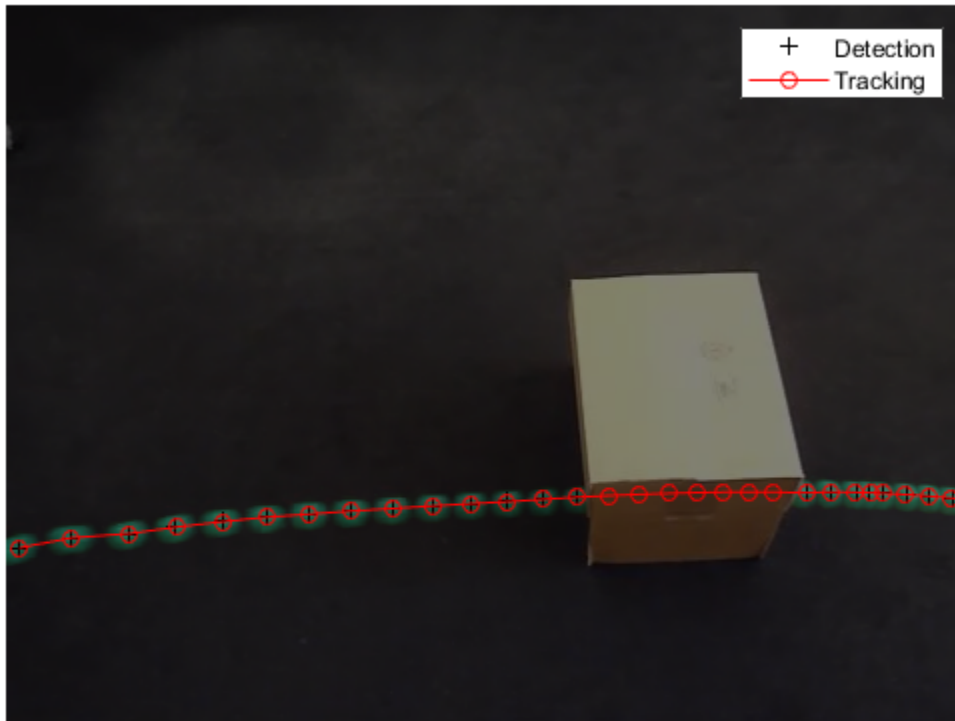
There are two distinct scenarios that the Kalman filter addresses:

- When the ball is detected, the Kalman filter first predicts its state at the current video frame, and then uses the newly detected object location to correct its state. This produces a filtered location.
- When the ball is missing, the Kalman filter solely relies on its previous state to predict the ball's current location.

You can see the ball's trajectory by overlaying all video frames.

```
param = getDefaultParameters(); % get Kalman configuration that works well
                                % for this example

trackSingleObject(param); % visualize the results
```



Explore Kalman Filter Configuration Options

Configuring the Kalman filter can be very challenging. Besides basic understanding of the Kalman filter, it often requires experimentation in order to come up with a set of suitable configuration parameters. The `trackSingleObject` function, defined above, helps you to explore the various configuration options offered by the `configureKalmanFilter` function.

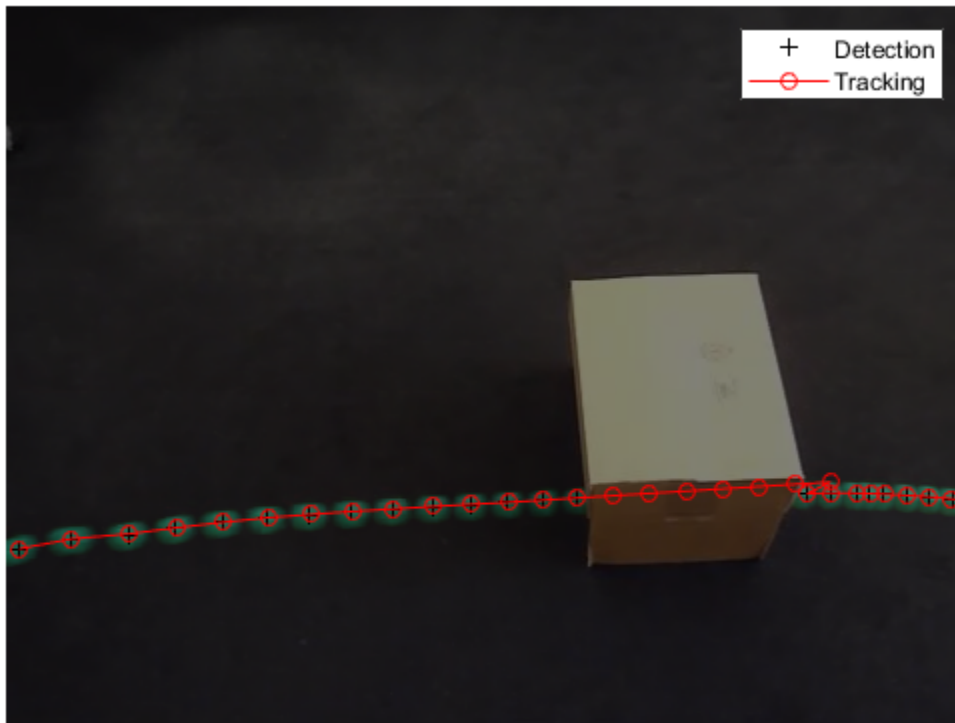
The `configureKalmanFilter` function returns a Kalman filter object. You must provide five input arguments.


```
kalmanFilter = configureKalmanFilter(MotionModel, InitialLocation,
    InitialEstimateError, MotionNoise, MeasurementNoise)
```

The **MotionModel** setting must correspond to the physical characteristics of the object's motion. You can set it to either a constant velocity or constant acceleration model. The following example illustrates the consequences of making a sub-optimal choice.

```
param = getDefaultParameters();           % get parameters that work well
param.motionModel = 'ConstantVelocity'; % switch from ConstantAcceleration
                                         % to ConstantVelocity
% After switching motion models, drop noise specification entries
% corresponding to acceleration.
param.initialEstimateError = param.initialEstimateError(1:2);
param.motionNoise          = param.motionNoise(1:2);

trackSingleObject(param); % visualize the results
```



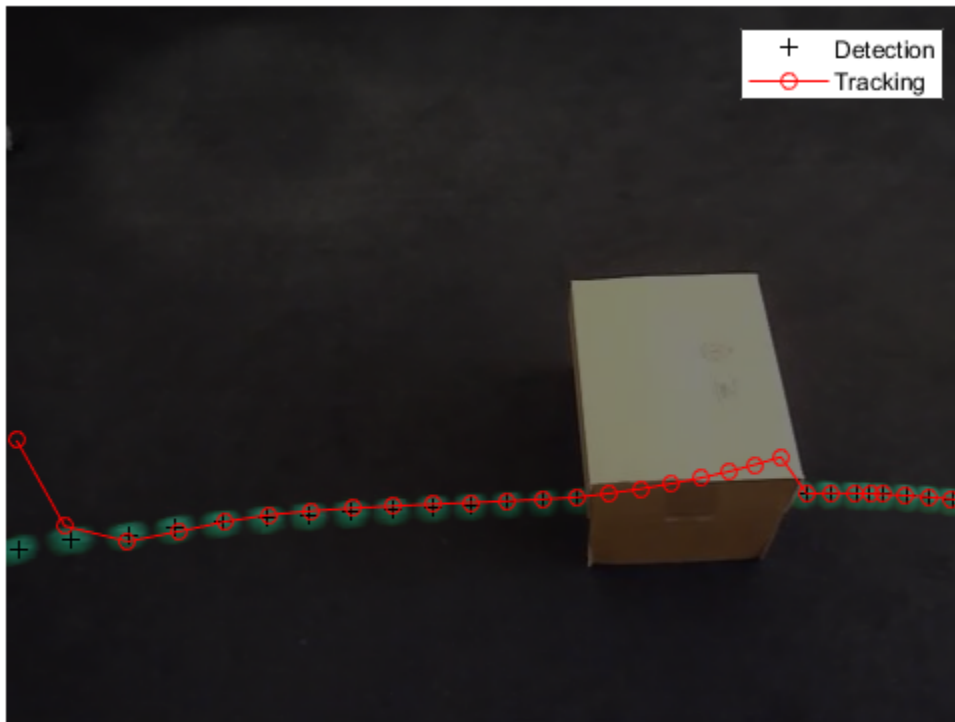
Notice that the ball emerged in a spot that is quite different from the predicted location. From the time when the ball was released, it was subject to constant deceleration due to resistance from the carpet. Therefore, constant acceleration model was a better choice. If you kept the constant velocity model, the tracking results would be sub-optimal no matter what you selected for the other values.

Typically, you would set the **InitialLocation** input to the location where the object was first detected. You would also set the **InitialEstimateError** vector to large values since the initial state may be very

noisy given that it is derived from a single detection. The following figure demonstrates the effect of misconfiguring these parameters.

```
param = getDefaultParameters(); % get parameters that work well
param.initialLocation = [0, 0]; % location that's not based on an actual detection
param.initialEstimateError = 100*ones(1,3); % use relatively small values

trackSingleObject(param); % visualize the results
```

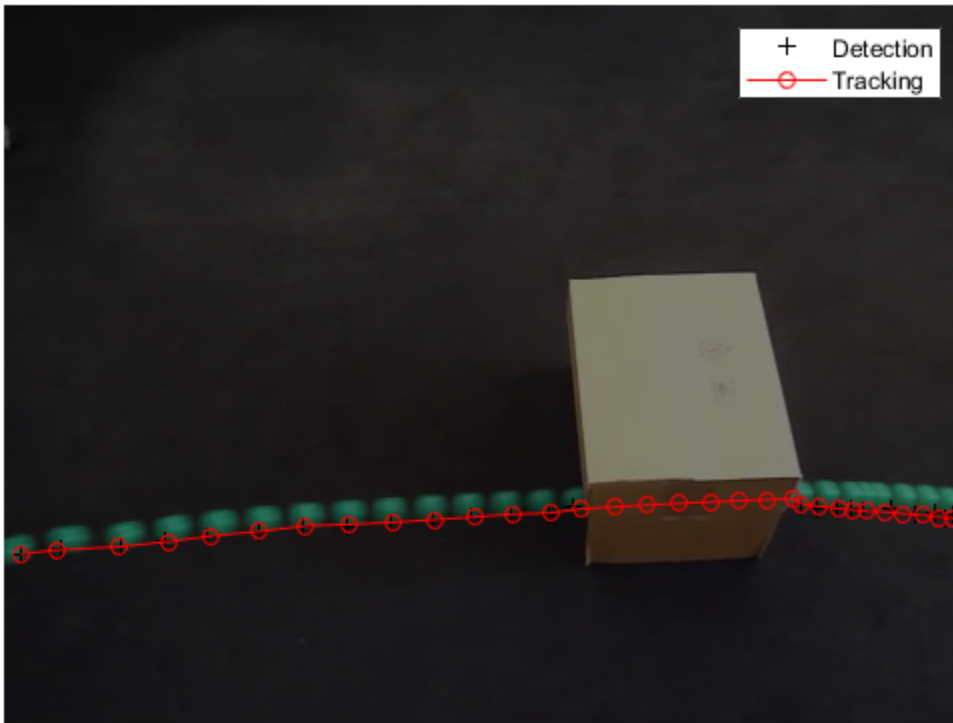


With the misconfigured parameters, it took a few steps before the locations returned by the Kalman filter align with the actual trajectory of the object.

The values for **MeasurementNoise** should be selected based on the detector's accuracy. Set the measurement noise to larger values for a less accurate detector. The following example illustrates the noisy detections of a misconfigured segmentation threshold. Increasing the measurement noise causes the Kalman filter to rely more on its internal state rather than the incoming measurements, and thus compensates for the detection noise.

```
param = getDefaultParameters();
param.segmentationThreshold = 0.0005; % smaller value resulting in noisy detections
param.measurementNoise      = 12500; % increase the value to compensate
                                % for the increase in measurement noise

trackSingleObject(param); % visualize the results
```



Typically objects do not move with constant acceleration or constant velocity. You use the **MotionNoise** to specify the amount of deviation from the ideal motion model. When you increase the motion noise, the Kalman filter relies more heavily on the incoming measurements than on its internal state. Try experimenting with **MotionNoise** parameter to learn more about its effects.

Now that you are familiar with how to use the Kalman filter and how to configure it, the next section will help you learn how it can be used for multiple object tracking.

Note: In order to simplify the configuration process in the above examples, we used the `configureKalmanFilter` function. This function makes several assumptions. See the function's documentation for details. If you require greater level of control over the configuration process, you can use the `vision.KalmanFilter` object directly.

Track Multiple Objects Using Kalman Filter

Tracking multiple objects poses several additional challenges:

- Multiple detections must be associated with the correct tracks
- You must handle new objects appearing in a scene
- Object identity must be maintained when multiple objects merge into a single detection

The `vision.KalmanFilter` object together with the `assignDetectionsToTracks` function can help to solve the problems of

- Assigning detections to tracks
- Determining whether or not a detection corresponds to a new object, in other words, track creation
- Just as in the case of an occluded single object, prediction can be used to help separate objects that are close to each other

To learn more about using Kalman filter to track multiple objects, see the example titled “Motion-Based Multiple Object Tracking” on page 7-31.

Utility Functions Used in the Example

Utility functions were used for detecting the objects and displaying the results. This section illustrates how the example implemented these functions.

Get default parameters for creating Kalman filter and for segmenting the ball.

```
function param = getDefaultParameters
    param.motionModel      = 'ConstantAcceleration';
    param.initialLocation  = 'Same as first detection';
    param.initialEstimateError = 1E5 * ones(1, 3);
    param.motionNoise      = [25, 10, 1];
    param.measurementNoise = 25;
    param.segmentationThreshold = 0.05;
end
```

Detect and annotate the ball in the video.

```
function showDetections()
    param = getDefaultParameters();
    utilities = createUtilities(param);
    trackedLocation = [];

    idx = 0;
    while hasFrame(utilities.videoReader)
        frame = readFrame(utilities.videoReader);
        detectedLocation = detectObject(frame);
        % Show the detection result for the current video frame.
        annotateTrackedObject();

        % To highlight the effects of the measurement noise, show the detection
        % results for the 40th frame in a separate figure.
        idx = idx + 1;
        if idx == 40
            combinedImage = max(repmat(utilities.foregroundMask, [1,1,3]), im2single(frame));
            figure, imshow(combinedImage);
        end
    end % while

    % Close the window which was used to show individual video frame.
    uiscopes.close('All');
end
```

Detect the ball in the current video frame.

```
function [detection, isObjectDetected] = detectObject(frame)
    grayImage = rgb2gray(im2single(frame));
    utilities.foregroundMask = step(utilities.foregroundDetector, grayImage);
```

```

detection = step(utilities.blobAnalyzer, utilities.foregroundMask);
if isempty(detection)
    isObjectDetected = false;
else
    % To simplify the tracking process, only use the first detected object.
    detection = detection(1, :);
    isObjectDetected = true;
end
end

```

Show the current detection and tracking results.

```

function annotateTrackedObject()
    accumulateResults();
    % Combine the foreground mask with the current video frame in order to
    % show the detection result.
    combinedImage = max(repmat(utilities.foregroundMask, [1,1,3]), im2single(frame));

    if ~isempty(trackedLocation)
        shape = 'circle';
        region = trackedLocation;
        region(:, 3) = 5;
        combinedImage = insertObjectAnnotation(combinedImage, shape, ...
            region, {label}, 'Color', 'red');
    end
    step(utilities.videoPlayer, combinedImage);
end

```

Show trajectory of the ball by overlaying all video frames on top of each other.

```

function showTrajectory
    % Close the window which was used to show individual video frame.
    uiscopes.close('All');

    % Create a figure to show the processing results for all video frames.
    figure; imshow(utilities.accumulatedImage/2+0.5); hold on;
    plot(utilities.accumulatedDetections(:,1), ...
        utilities.accumulatedDetections(:,2), 'k+');

    if ~isempty(utilities.accumulatedTrackings)
        plot(utilities.accumulatedTrackings(:,1), ...
            utilities.accumulatedTrackings(:,2), 'r-o');
        legend('Detection', 'Tracking');
    end
end

```

Accumulate video frames, detected locations, and tracked locations to show the trajectory of the ball.

```

function accumulateResults()
    utilities.accumulatedImage = max(utilities.accumulatedImage, frame);
    utilities.accumulatedDetections ...
        = [utilities.accumulatedDetections; detectedLocation];
    utilities.accumulatedTrackings ...
        = [utilities.accumulatedTrackings; trackedLocation];
end

```

For illustration purposes, select the initial location used by the Kalman filter.

```
function loc = computeInitialLocation(param, detectedLocation)
    if strcmp(param.initialLocation, 'Same as first detection')
        loc = detectedLocation;
    else
        loc = param.initialLocation;
    end
end
```

Create utilities for reading video, detecting moving objects, and displaying the results.

```
function utilities = createUtilities(param)
    % Create System objects for reading video, displaying video, extracting
    % foreground, and analyzing connected components.
    utilities.videoReader = VideoReader('singleball.mp4');
    utilities.videoPlayer = vision.VideoPlayer('Position', [100,100,500,400]);
    utilities.foregroundDetector = vision.ForegroundDetector(...
        'NumTrainingFrames', 10, 'InitialVariance', param.segmentationThreshold);
    utilities.blobAnalyzer = vision.BlobAnalysis('AreaOutputPort', false, ...
        'MinimumBlobArea', 70, 'CentroidOutputPort', true);

    utilities.accumulatedImage      = 0;
    utilities.accumulatedDetections = zeros(0, 2);
    utilities.accumulatedTrackings  = zeros(0, 2);
end
```

end

Detect Cars Using Gaussian Mixture Models

This example shows how to detect and count cars in a video sequence using foreground detector based on Gaussian mixture models (GMMs).

Introduction

Detecting and counting cars can be used to analyze traffic patterns. Detection is also a first step prior to performing more sophisticated tasks such as tracking or categorization of vehicles by their type.

This example shows how to use the foreground detector and blob analysis to detect and count cars in a video sequence. It assumes that the camera is stationary. The example focuses on detecting objects. To learn more about tracking objects, see the example titled “Motion-Based Multiple Object Tracking” on page 7-31.

Step 1 - Import Video and Initialize Foreground Detector

Rather than immediately processing the entire video, the example starts by obtaining an initial video frame in which the moving objects are segmented from the background. This helps to gradually introduce the steps used to process the video.

The foreground detector requires a certain number of video frames in order to initialize the Gaussian mixture model. This example uses the first 50 frames to initialize three Gaussian modes in the mixture model.

```
foregroundDetector = vision.ForegroundDetector('NumGaussians', 3, ...
    'NumTrainingFrames', 50);

videoReader = VideoReader('visiontraffic.avi');
for i = 1:150
    frame = readFrame(videoReader); % read the next video frame
    foreground = step(foregroundDetector, frame);
end
```

After the training, the detector begins to output more reliable segmentation results. The two figures below show one of the video frames and the foreground mask computed by the detector.

```
figure; imshow(frame); title('Video Frame');
```

Video Frame



```
figure; imshow(foreground); title('Foreground');
```

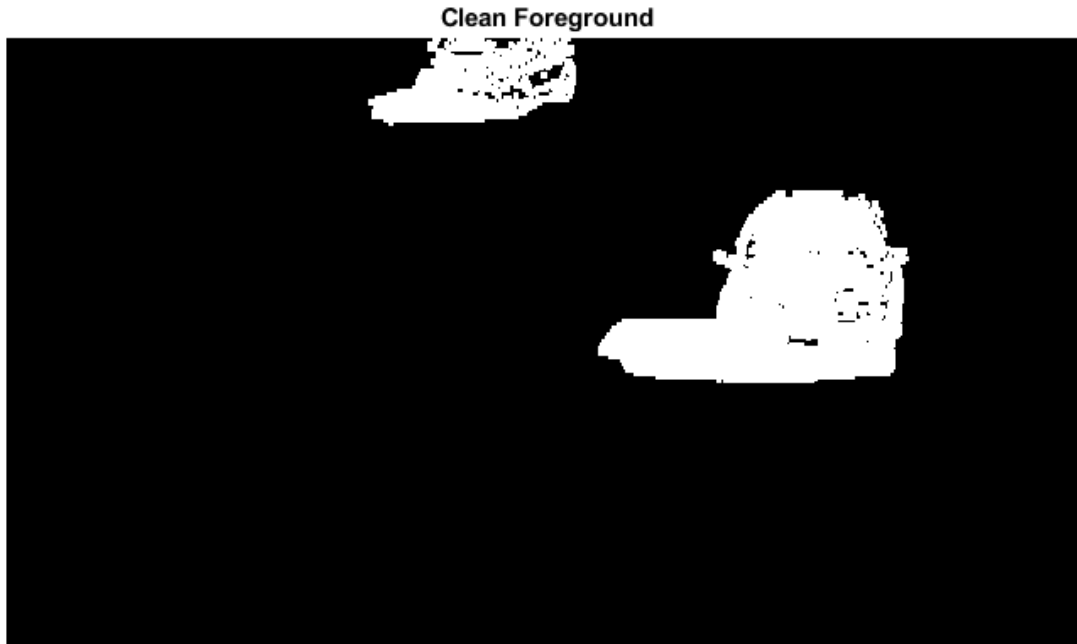
Foreground



Step 2 - Detect Cars in an Initial Video Frame

The foreground segmentation process is not perfect and often includes undesirable noise. The example uses morphological opening to remove the noise and to fill gaps in the detected objects.

```
se = strel('square', 3);
filteredForeground = imopen(foreground, se);
figure; imshow(filteredForeground); title('Clean Foreground');
```



Next, find bounding boxes of each connected component corresponding to a moving car by using `vision.BlobAnalysis` object. The object further filters the detected foreground by rejecting blobs which contain fewer than 150 pixels.

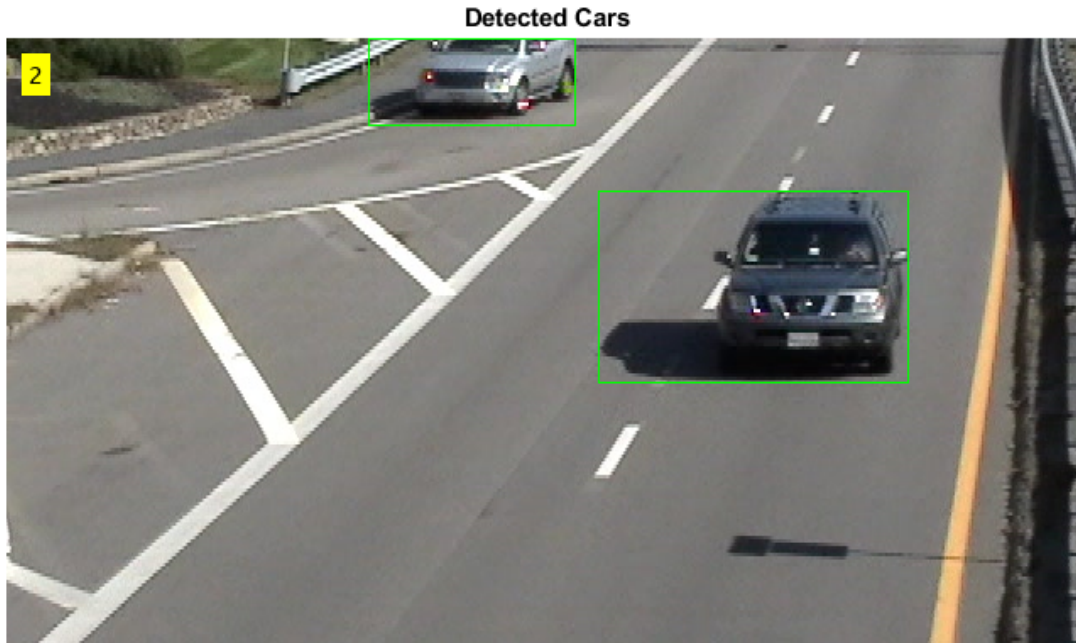
```
blobAnalysis = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
    'AreaOutputPort', false, 'CentroidOutputPort', false, ...
    'MinimumBlobArea', 150);
bbox = step(blobAnalysis, filteredForeground);
```

To highlight the detected cars, we draw green boxes around them.

```
result = insertShape(frame, 'Rectangle', bbox, 'Color', 'green');
```

The number of bounding boxes corresponds to the number of cars found in the video frame. Display the number of found cars in the upper left corner of the processed video frame.

```
numCars = size(bbox, 1);
result = insertText(result, [10 10], numCars, 'BoxOpacity', 1, ...
    'FontSize', 14);
figure; imshow(result); title('Detected Cars');
```



Step 3 - Process the Rest of Video Frames

In the final step, we process the remaining video frames.

```

videoPlayer = vision.VideoPlayer('Name', 'Detected Cars');
videoPlayer.Position(3:4) = [650,400]; % window size: [width, height]
se = strel('square', 3); % morphological filter for noise removal

while hasFrame(videoReader)

    frame = readFrame(videoReader); % read the next video frame

    % Detect the foreground in the current video frame
    foreground = step(foregroundDetector, frame);

    % Use morphological opening to remove noise in the foreground
    filteredForeground = imopen(foreground, se);

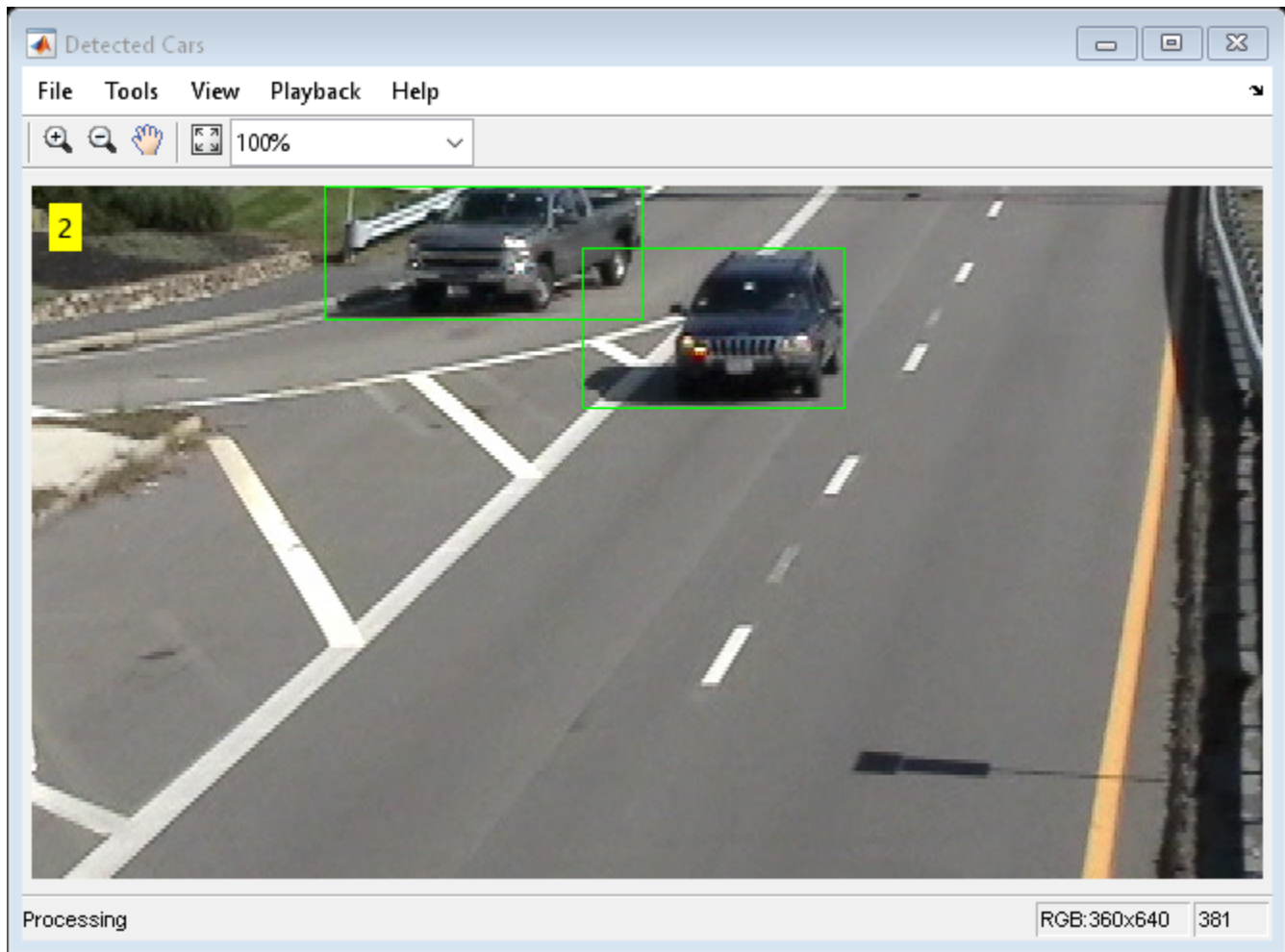
    % Detect the connected components with the specified minimum area, and
    % compute their bounding boxes
    bbox = step(blobAnalysis, filteredForeground);

    % Draw bounding boxes around the detected cars
    result = insertShape(frame, 'Rectangle', bbox, 'Color', 'green');

    % Display the number of cars found in the video frame
    numCars = size(bbox, 1);
    result = insertText(result, [10 10], numCars, 'BoxOpacity', 1, ...
        'FontSize', 14);

```

```
step(videoPlayer, result); % display the results  
end
```



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

Featured Examples

- “Localize and Read Multiple Barcodes in Image” on page 8-2
- “Monocular Visual Odometry” on page 8-22
- “Detect and Track Vehicles Using Lidar Data” on page 8-35
- “Semantic Segmentation Using Dilated Convolutions” on page 8-54
- “Define Custom Pixel Classification Layer with Tversky Loss” on page 8-58
- “Track a Face in Scene” on page 8-65
- “Create 3-D Stereo Display” on page 8-70
- “Measure Distance from Stereo Camera to a Face” on page 8-71
- “Reconstruct 3-D Scene from Disparity Map” on page 8-72
- “Visualize Stereo Pair of Camera Extrinsic Parameters” on page 8-75
- “Remove Distortion from an Image Using the Camera Parameters Object” on page 8-78

Localize and Read Multiple Barcodes in Image

This example shows how to use the `readBarcode` function from the Computer Vision Toolbox™ to detect and decode 1-D and 2-D barcodes in an image. Barcodes are widely used to encode data in a visual, machine-readable format. They are useful in many applications such as item identification, warehouse inventory tracking, and compliance tracking. For 1-D barcodes, the `readBarcode` function returns the location of the barcode endpoints. For 2-D barcodes, the function returns the locations of the finder patterns. This example uses two approaches for localizing multiple barcodes in an image. One approach is clustering-based, which is more robust to different imaging conditions and requires the Statistics and Machine Learning Toolbox™. The second approach uses a segmentation-based workflow and might require parameter tuning based on the imaging conditions.

Barcode Detection using the `readBarcode` Function

Read a QR code from an image.

```
I = imread("barcodeQR.jpg");

% Search the image for a QR Code.
[msg, ~, loc] = readBarcode(I);

% Annotate the image with the decoded message.
xyText = loc(2,:);
Imsg = insertText(I, xyText, msg, "BoxOpacity", 1, "FontSize", 25);

% Insert filled circles at the finder pattern locations.
Imsg = insertShape(Imsg, "FilledCircle", [loc, ...
    repmat(10, length(loc), 1)], "Color", "red", "Opacity", 1);

% Display image.
imshow(Imsg)
```



Read a 1-D barcode from an image.

```
I = imread("barcode1D.jpg");

% Read the 1-D barcode and determine the format..
[msg, format, locs] = readBarcode(I);

% Display the detected message and format.
disp("Detected format and message: " + format + ", " + msg)

Detected format and message: EAN-13, 1234567890128

% Insert a line to show the scan row of the barcode.
xyBegin = locs(1,:); imSize = size(I);
I = insertShape(I,"Line",[1 xyBegin(2) imSize(2) xyBegin(2)], ...
    "LineWidth", 7);

% Insert markers at the end locations of the barcode.
I = insertShape(I, "FilledCircle", [locs, ...
    repmat(10, length(locs), 1)], "Color", "red", "Opacity", 1);
```

```
% Display image.  
imshow(I)
```



Improving Barcode Detection

For a successful detection, the barcode must be clearly visible. The barcode must also be as closely aligned to a horizontal or vertical position as possible. The `readBarcode` function is inherently more robust to rotations for 2-D or matrix codes than it is to 1-D or linear barcodes. For example, the barcode cannot be detected in this image.

```
I = imread("rotated1DBarcode.jpg");
```

```
% Display the image.  
imshow(I)
```




```
% Pass the image to the readBarcode function.  
readBarcode(I)
```

```
ans =  
''
```

Rotate the image using the `imrotate` so that the barcode is roughly horizontal. Use `readBarcode` on the rotated image.

```
% Rotate the image by 30 degrees clockwise.  
Irot = imrotate(I, -30);
```

```
% Display the rotated image.  
imshow(Irot)
```



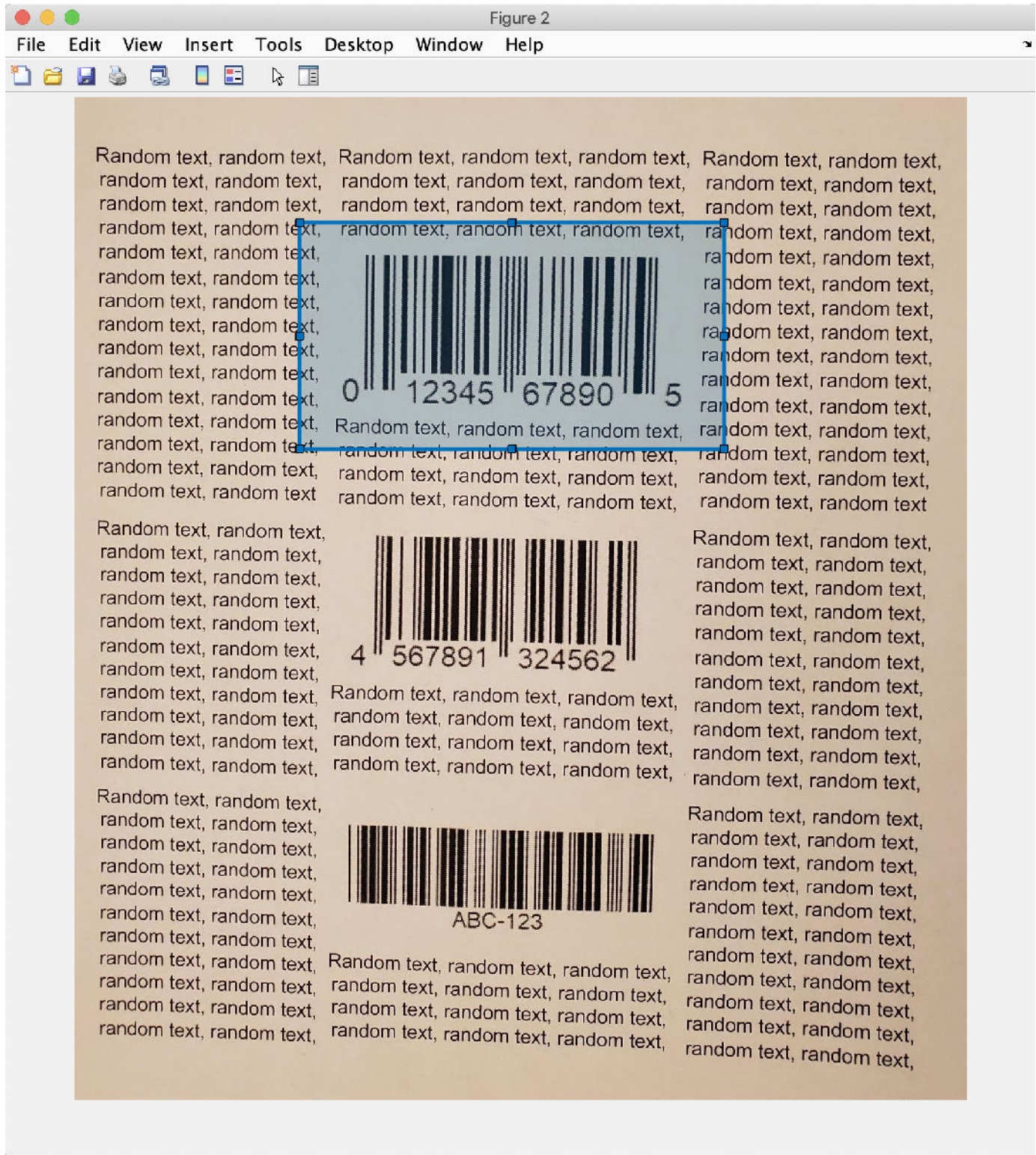
```
% Pass the rotated image to the readBarcode function.  
readBarcode(Irot)
```

```
ans =  
"012345678905"
```

Detect Multiple Barcodes

The `readBarcode` function detects only a single barcode in each image. In order to detect multiple barcodes, you must specify a region-of-interest (ROI). To specify an ROI, you can use the `drawrectangle` function to interactively determine the ROIs. You can also use image analysis techniques to detect the ROI of multiple barcodes in the image.

Interactively determine ROIs



```
I = imread("multiple1DBarcodes.jpg");
```

Use the `drawrectangle` function to draw and obtain rectangle parameters.

```
roi1 = drawrectangle;

pos = roi1.Position;

% ROIs obtained using drawrectangle
roi = [180 100 330 180
       180 320 330 180
       180 550 330 180];

imSize = size(I);
for i = 1:size(roi,1)
    [msg, format, locs] = readBarcode(I, roi(i,:));
    disp("Decoded format and message: " + format + ", " + msg)

    % Insert a line to indicate the scan row of the barcode.
    xyBegin = locs(1,:);
    I = insertShape(I,"Line",[1 xyBegin(2) imSize(2) xyBegin(2)], ...
                    "LineWidth", 5);

    % Annotate image with decoded message.
    I = insertText(I, xyBegin, msg, "BoxOpacity", 1, "FontSize", 20);
end

Decoded format and message: UPC-A, 012345678905
Decoded format and message: EAN-13, 4567891324562
Decoded format and message: CODE-39, ABC-123

imshow(I)
```

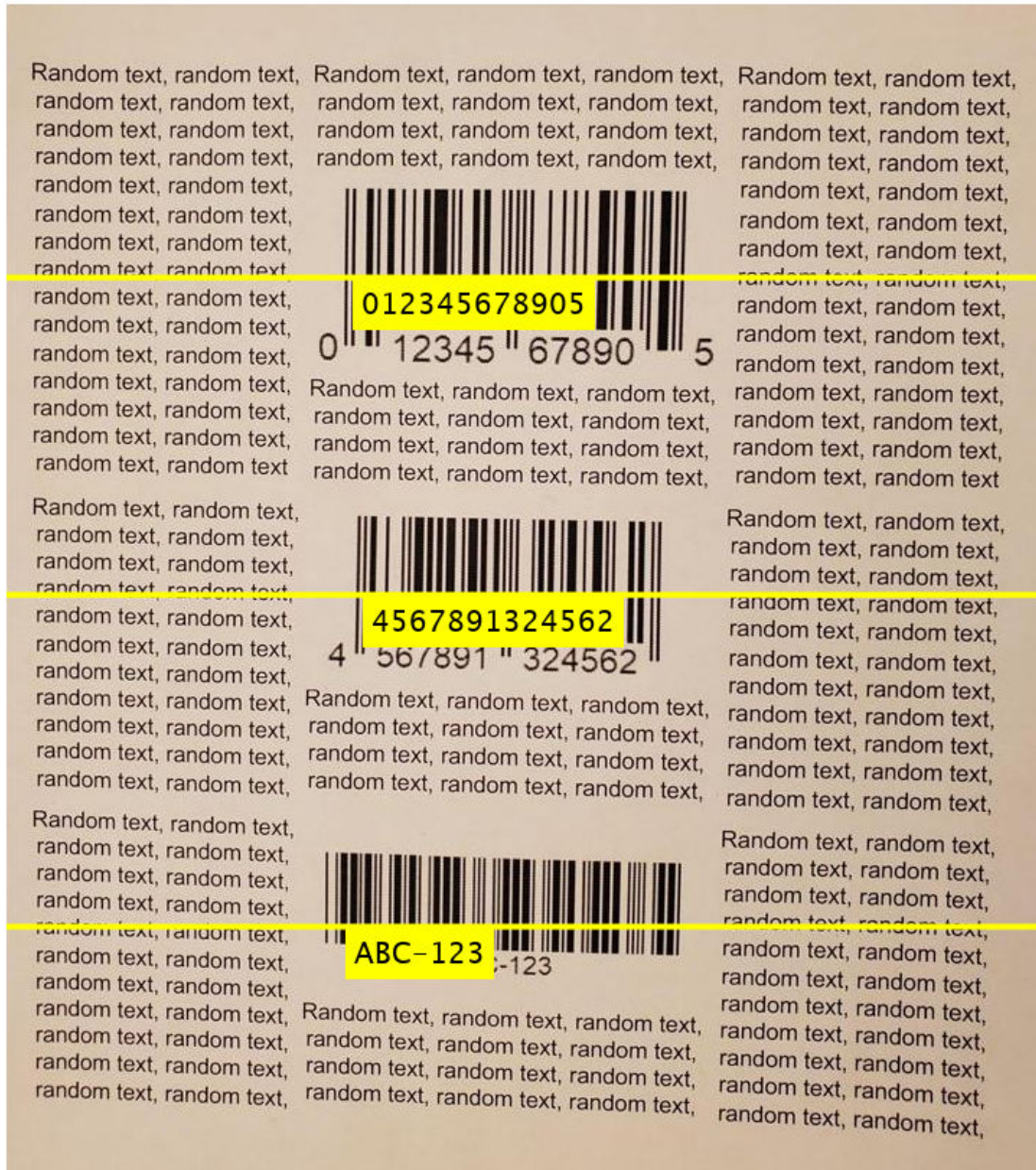
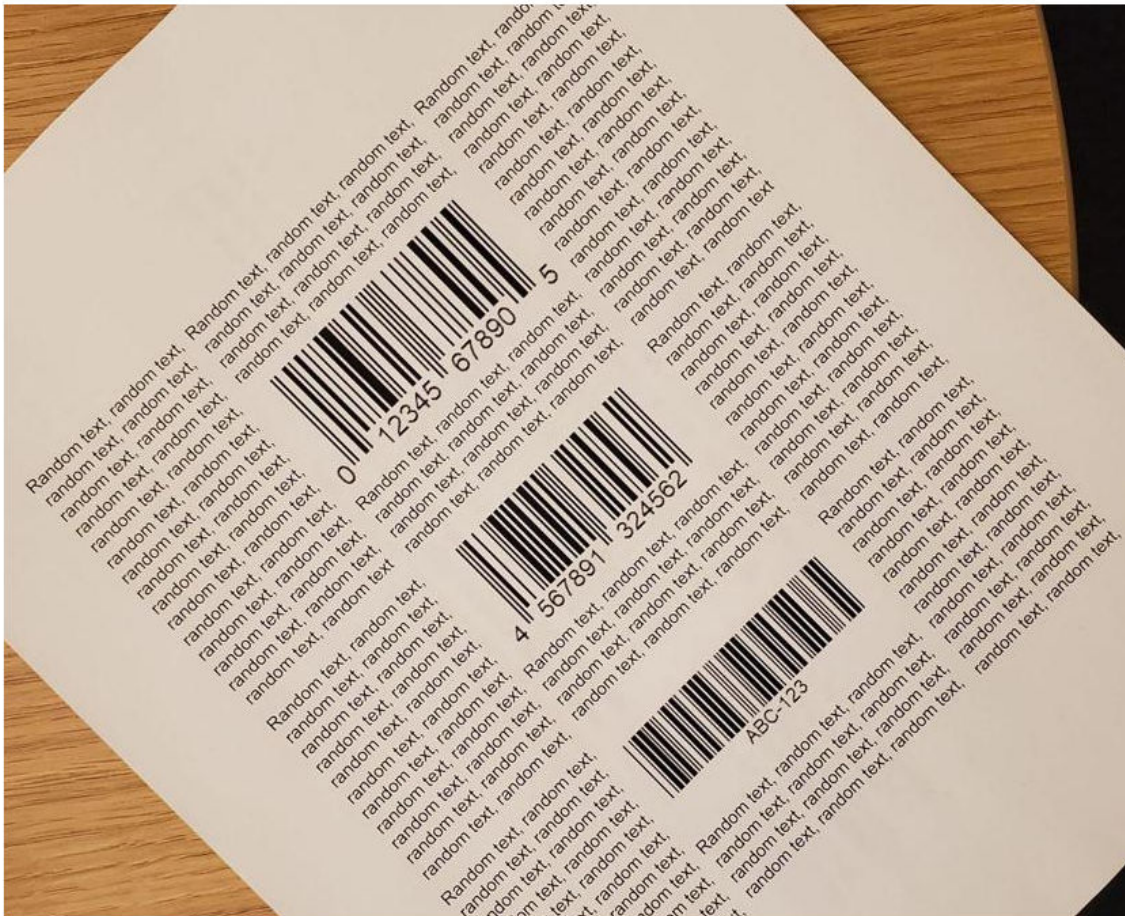


Image analysis to determine ROIs

Use image analysis techniques to automate the detection of multiple barcodes. This requires localizing multiple barcodes in an image, determining their orientation, and correcting for the orientation. Without preprocessing, barcodes cannot be detected in the image containing multiple rotated barcodes.

```
I = imread("multiple1DBarcodesRotated.jpg");
Igray = rgb2gray(I);

% Display the image.
imshow(I)
```



```
% Pass the unprocessed image to the readBarcode function.
readBarcode(Igray, '1D')
```

```
ans =
''
```

Detection on the unprocessed image resulted in no detection.

Step 1: Detect candidate regions for the barcodes using MSER

Detect regions of interest in the image using the `detectMSERFeatures` function. Then, you can eliminate regions of interest based on a specific criteria such as the aspect ratio. You can use the binary image from the filtered results for further processing.

```
% Detect MSER features.
[~, cc] = detectMSERFeatures(Igray);

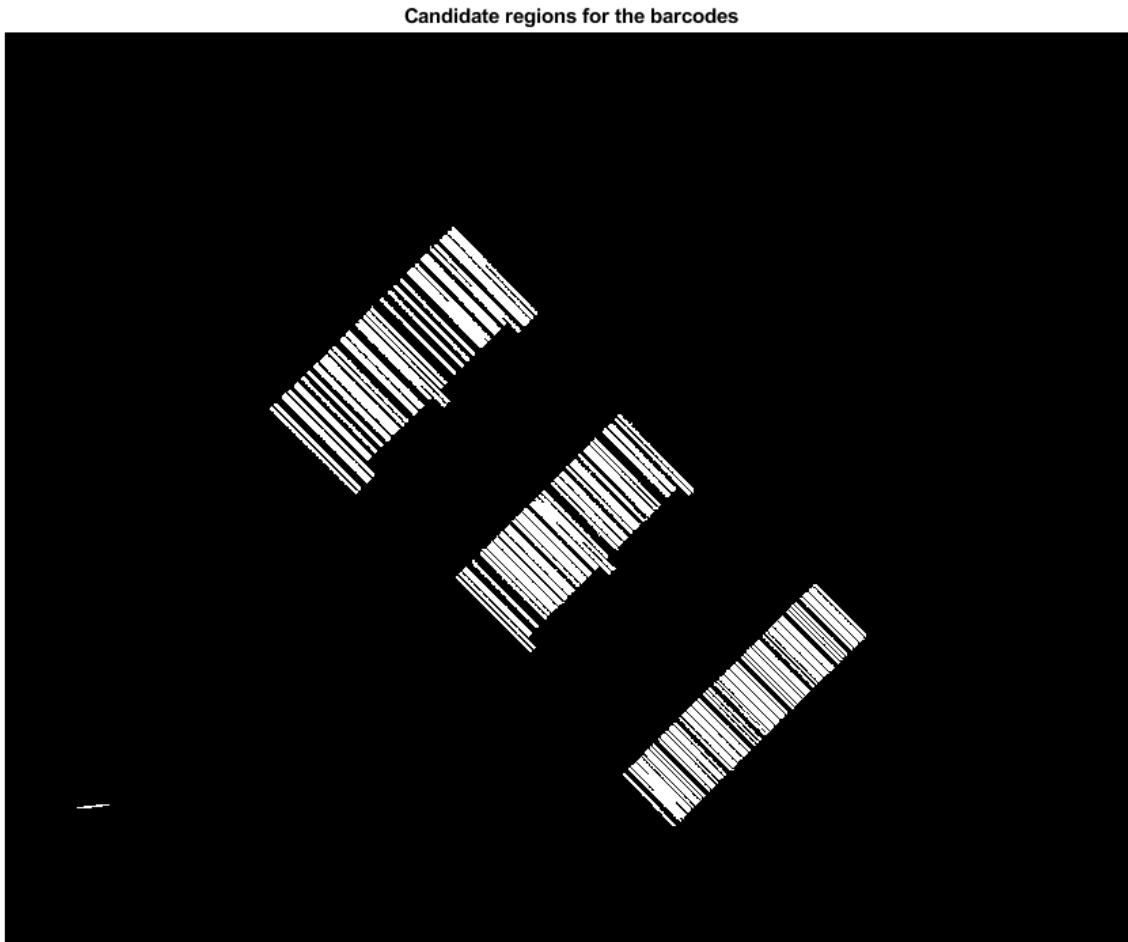
% Compute region properties MajorAxisLength and MinorAxisLength.
regionStatistics = regionprops(cc, 'MajorAxisLength', 'MinorAxisLength');

% Filter out components that have a low aspect ratio as unsuitable
% candidates for the bars in the barcode.
minAspectRatio = 10;
candidateRegions = find((regionStatistics.MajorAxisLength)./regionStatistics.MinorAxisLength)

% Binary image to store the filtered components.
BW = false(size(Igray));

% Update the binary image.
for i = 1:length(candidateRegions)
    BW(cc.PixelIdxList{candidateRegions(i)}) = true;
end

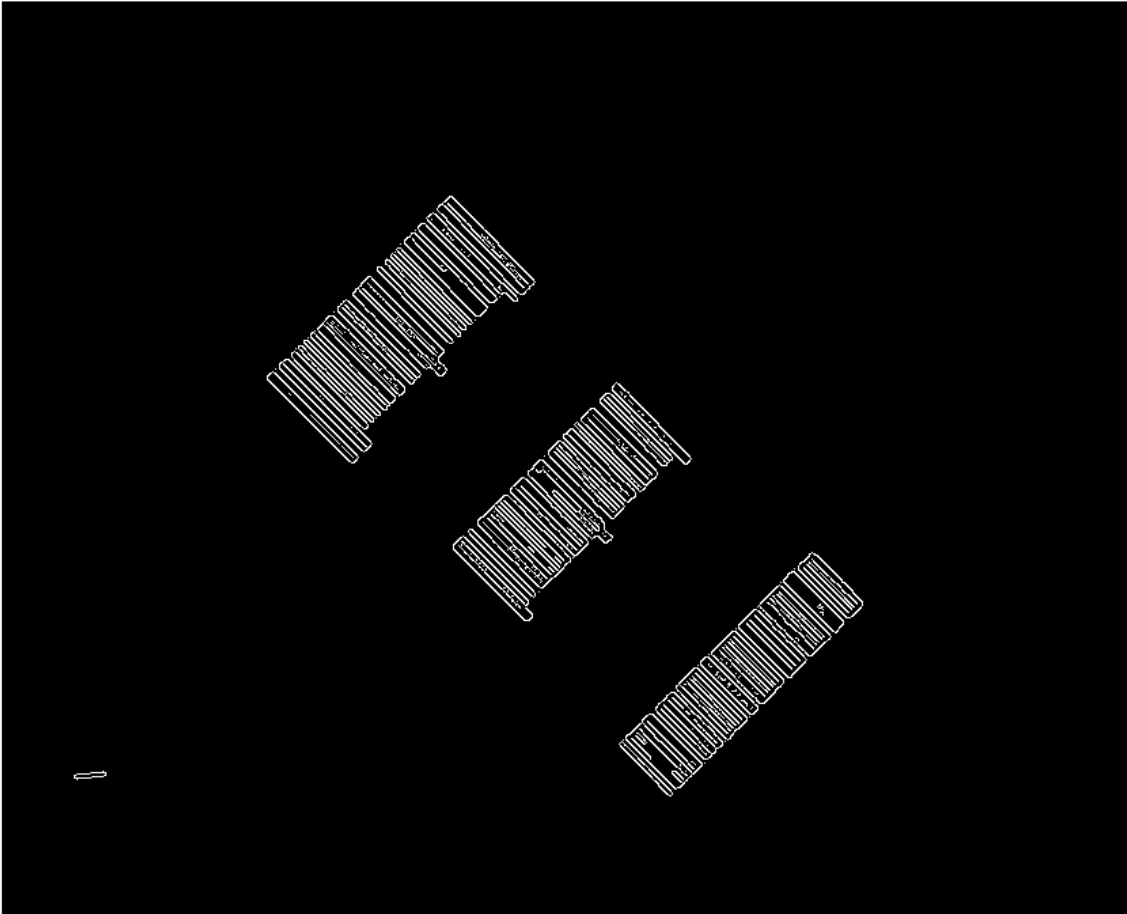
% Display the binary image with the filtered components.
imshow(BW)
title("Candidate regions for the barcodes")
```



Step 2: Extract barcode line segments using hough transform

Detect prominent edges in the image using the `edge` function. Then use the hough transform to find lines of interest. The lines represent possible candidates for the vertical bars in the barcode.

```
% Perform hough transform.  
BW = edge(BW, 'canny');  
[H,T,R] = hough(BW);  
  
% Display the result of the edge detection operation.  
imshow(BW)
```

```

% Determine the size of the suppression neighborhood.
reductionRatio = 500;
nhSize = floor(size(H)/reductionRatio);
idx = mod(nhSize,2) < 1;
nhSize(idx) = nhSize(idx) + 1;

% Identify the peaks in the Hough transform.
P = houghpeaks(H,length(candidateRegions),'NHoodSize',nhSize);

% Detect the lines based on the detected peaks.
lines = houghlines(BW,T,R,P);

% Display the lines detected using the houghlines function.
Ithoughlines = ones(size(BW));

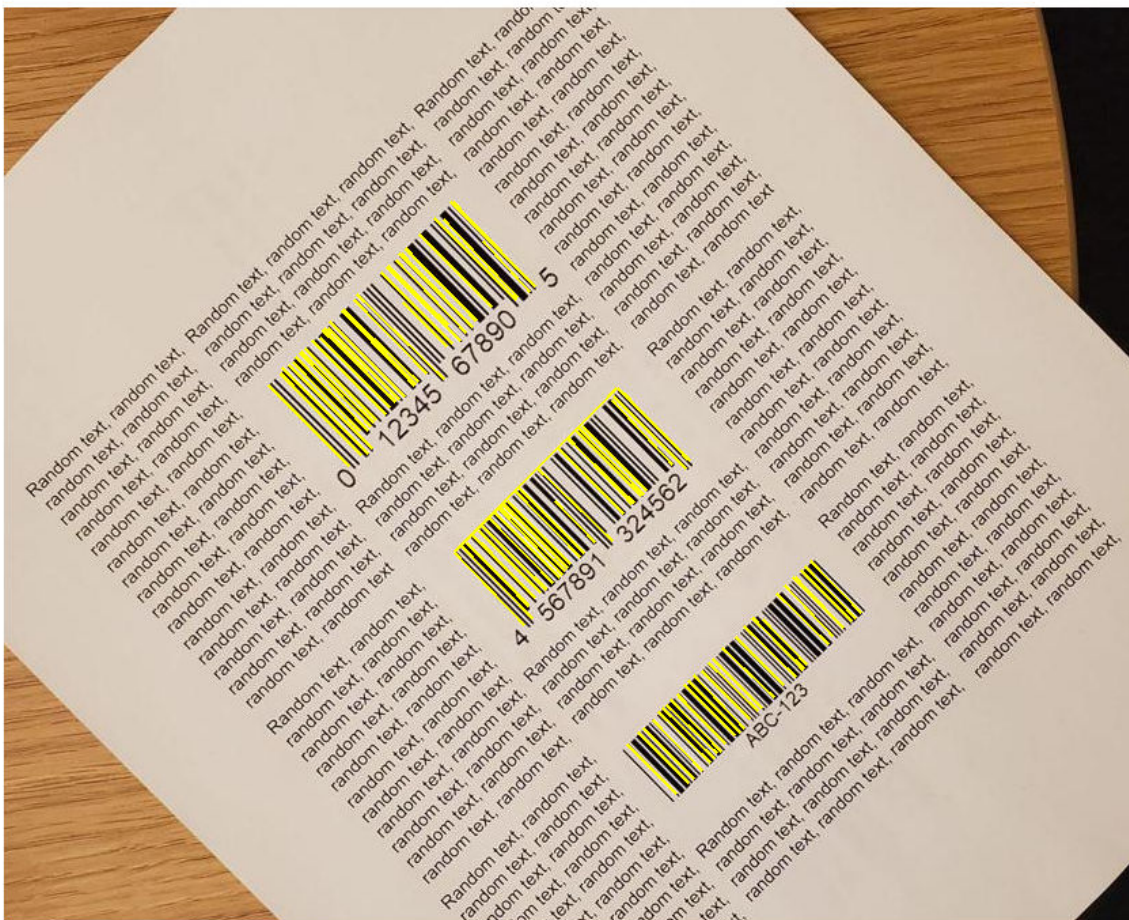
% Start and end points of the detected lines.
startPts = reshape([lines(:).point1], 2, length(lines))';
endPts = reshape([lines(:).point2], 2, length(lines))';
    
```

```

Ithoughlines = insertShape(Ithoughlines, 'Line', [startPts, endPts], ...
    'LineWidth', 2, 'Color', 'green');

% Display the original image overlaid with the detected lines.
Ibarlines = imoverlay(I, ~Ithoughlines(:,:,1));
imshow(Ibarlines)

```



Step 3: Localize barcodes in image

After extracting the line segments, two methods are presented for localizing the individual barcodes in the image:

- Method 1: A clustering-based technique that uses functionalities from the Statistics and Machine Learning Toolbox™ to identify individual barcodes. This technique is more robust to outliers that were detected using the image analysis techniques above. It can also be extended to a wide range of imaging conditions without having to tune parameters.
- Method 2: A segmentation-based workflow to separate the individual barcodes. This method uses other image analysis techniques to localize and rotation correct the extracted barcodes. While this works fairly well, it might require some parameter tuning to prevent detection of outliers.

Method 1: Clustering based workflow

There are two steps in this workflow:

1. Determine bisectors of barcode line segments

While it is common practice to directly use the lines (that were obtained using the Hough transform) to localize the barcode, this method uses the lines to further detect the perpendicular bisectors for each of the lines. The bisector lines are represented as points in cartesian space, which makes them suitable for identifying individual barcodes. Using the bisectors make the detection of the individual barcodes more robust, since it results in less misclassifications of lines that are similar but belonging to different barcodes.

2. Perform clustering on the bisectors to identify the individual barcodes

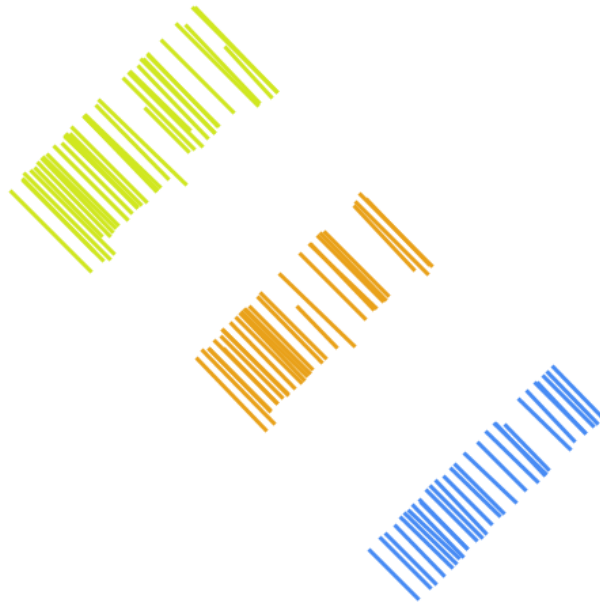
Since all of the bars in a barcode are approximately parallel to each other, the bisectors of each of these bars should ideally be the same line, and their corresponding points should therefore cluster around a single point. In practice, these bisectors will vary from segment to segment, but still remain similar enough to allow the use of a density-based clustering algorithm. The result of performing this clustering operation is a set of clusters, each of which points to a separate barcode. This example uses the `dbscan` (Statistics and Machine Learning Toolbox) function, which does not require prior knowledge of the number of clusters. The different clusters (barcodes) are visualized in this example.

The example checks for a Statistics and Machine Learning Toolbox™ license. If a license is found, the example uses the clustering method. Otherwise, the example uses the segmentation method.

```
useClustering = license('test','statistics_toolbox');

if useClustering
    [boundingBox, orientation, Iclusters] = clusteringLocalization(lines, size(I));

    % Display the detected clusters.
    imshow(Iclusters)
else
    disp("The clustering based workflow requires a license for the Statistics and Machine Learning Toolbox™.")
end
```



Method 2: Segmentation based workflow

Having removed the background noise and variation, the detected vertical bars are grouped into individual barcodes using morphological operations, like `imdilate`. The example uses the `regionprops` function to determine the bounding box and orientation for each of the barcodes. The results are used to crop the individual barcodes from the original image and to orient them to be roughly horizontal.

```
if ~useClustering
    [boundingBox, orientation, Idilated] = segmentationLocalization(Ithoughlines);

    % Display the dilated image.
    imshow(Idilated)
end
```

Step 4: Crop the Barcodes and correct their rotation

The barcodes are cropped from the original image using the bounding boxes obtained from the segmentation. The orientation results are used to align the barcodes to be approximately horizontal.

```

% Localize and rotate the barcodes in the image.
correctedImages = cell(1, length(orientation));

% Store the cropped and rotation corrected images of the barcodes.
for i = 1:length(orientation)

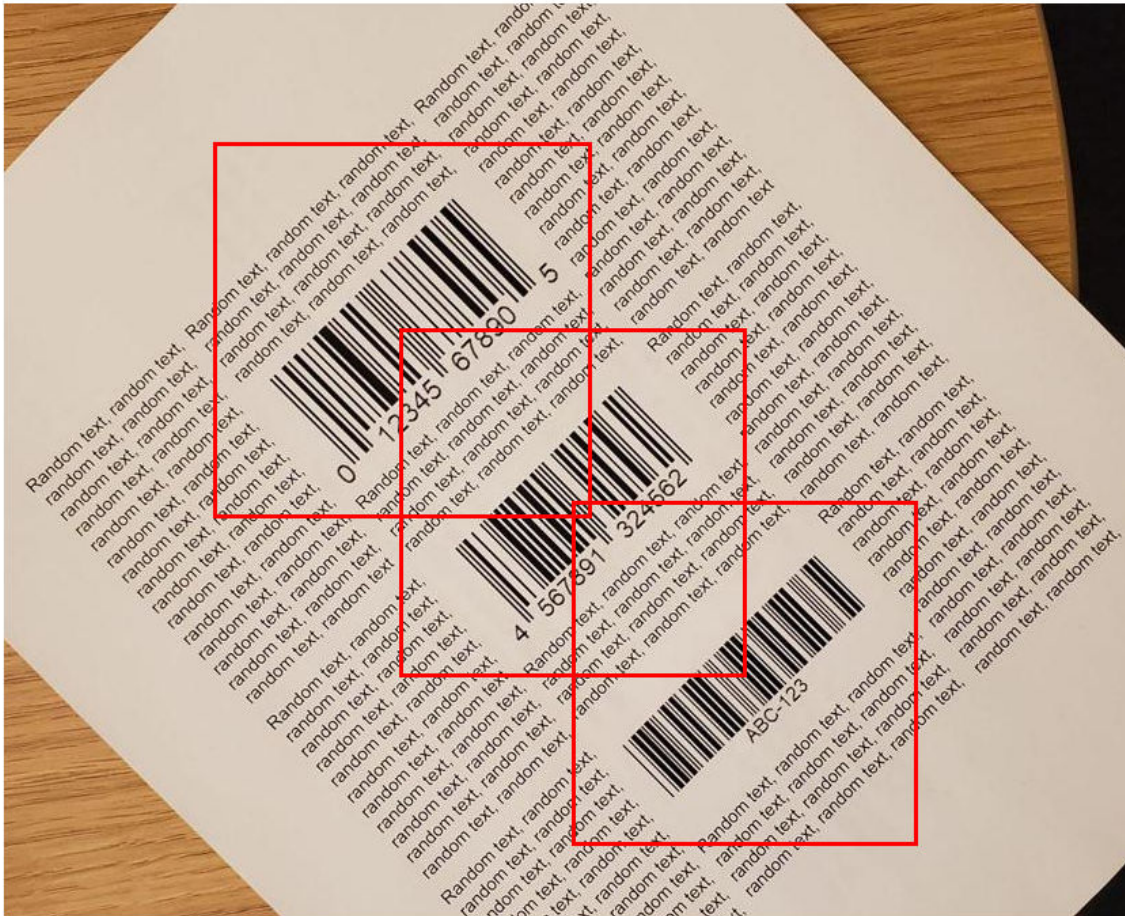
    I = insertShape(I, 'Rectangle', boundingBox(i,:), 'LineWidth',3, 'Color', 'red');

    if orientation(i) > 0
        orientation(i) = -(90 - orientation(i));
    else
        orientation(i) = 90 + orientation(i);
    end

    % Crop the barcode from the original image and rotate it using the
    % detected orientation.
    correctedImages{i} = imrotate(imcrop(Igray,boundingBox(i,:)), orientation(i));
end

% Display the image with the localized barcodes.
imshow(I)

```



Step 5: Detect barcodes in the cropped and rotation corrected images

The cropped and rotation corrected images of the barcodes are then used with the `readBarcode` function to decode them.

```
% Pass each of the images to the readBarcode function.
for i = 1:length(correctedImages)
    [msg, format, ~] = readBarcode(correctedImages{i}, '1D');
    disp("Decoded format and message: " + format + ", " + msg)
end
```

```
Decoded format and message: UPC-A, 012345678905
Decoded format and message: EAN-13, 4567891324562
Decoded format and message: CODE-39, ABC-123
```

This example showed how the `readBarcode` function can be used to detect, decode and localize barcodes in an image. While the function works well when the alignment of the barcodes is roughly horizontal or vertical, it needs additional pre-processing when the barcodes appear rotated. The preprocessing steps detailed above is a good starting point to work with multiple barcodes that are not aligned in an image.

Supporting Functions

clusteringLocalization uses a clustering-based workflow to localize individual barcodes.

```
function [boundingBox, orientation, lclusters] = clusteringLocalization(lines, imSize)

%-----
% Determine Bisectors of Barcode Line Segments
%-----

% Table to store the properties of the bisectors of the detected lines.
linesBisector = array2table(zeros(length(lines), 4), 'VariableNames', {'theta', 'rho', 'x', 'y'})

% Use the orientation values of the lines to determine the orientation.
% values of the bisectors
idxNeg = find([lines.theta] < 0);
idxPos = find([lines.theta] >= 0);

negAngles = 90 + [lines(idxNeg).theta];
linesBisector.theta(idxNeg) = negAngles;

posAngles = [lines(idxPos).theta] - 90;
linesBisector.theta(idxPos) = posAngles;

% Determine the midpoints of the detected lines.
midPts = zeros(length(lines),2);

% Determine the 'rho' values of the bisectors.
for i = 1:length(lines)
    midPts(i,:) = (lines(i).point1 + lines(i).point2)/2;
    linesBisector.rho(i) = abs(midPts(i,2) - tand(lines(i).theta) * midPts(i,1))/...
        ((tand(lines(i).theta)^2 + 1) ^ 0.5);
end

% Update the [x,y] locations of the bisectors using their polar
% coordinates.
[linesBisector.x, linesBisector.y] = pol2cart(deg2rad(linesBisector.theta),linesBisector.rho,'ro

%-----
% Perform Clustering on the Bisectors to Identity the Individual Barcodes
%-----

% Store the [x,y] data of the bisectors to be used for clustering.
X = [linesBisector.x,linesBisector.y];

% Get pairwise distance between the points
D = pdist2(X,X);

% Perform density-based spatial clustering to separate the different
% barcodes in the image.
searchRadius = max(imSize/5);
minPoints = 10;
idx = dbscan(D,searchRadius, minPoints);

% Identify the number of clusters (barcodes).
numClusters = unique(idx(idx > 0));

% Store the endpoints of the detected lines.
```

```

dataXY = cell(1, length(numClusters));

% Image to show the detected clusters (barcodes).
Iclusters = ones(imSize);

for i = 1:length(numClusters)
    classIdx = find(idx == i);

    rgbColor = rand(1,3);
    startPts = reshape([lines(classIdx).point1], 2, length(classIdx));
    endPts = reshape([lines(classIdx).point2], 2, length(classIdx));

    % Insert lines corresponding to the current cluster (barcode).
    Iclusters = insertShape(Iclusters, 'Line', [startPts, endPts], ...
        'LineWidth', 2, 'Color', rgbColor);

    % Update the endpoints of the lines in each cluster (barcode).
    dataXY{i} = [startPts; endPts];
end

%-----
% Localization parameters for the barcode
%-----

orientation = zeros(1,length(numClusters));
boundingBox = zeros(length(numClusters), 4);

% Padding the cropped images of barcodes.
padding = 40;

% Determine the ROI and orientation of the individual clusters (barcodes).
for i = 1:length(numClusters)

    % Bounding box coordinates with padding.
    x1 = min(dataXY{i}(:,1)) - padding;
    x2 = max(dataXY{i}(:,1)) + padding;
    y1 = min(dataXY{i}(:,2)) - padding;
    y2 = max(dataXY{i}(:,2)) + padding;

    boundingBox(i,:) = [x1, y1, x2-x1, y2-y1];

    % Orientation of the barcode.
    orientation(i) = mean(linesBisector.theta(idx == i));

end

end

segmentationLocalization uses a segmentation-based workflow to localize individual barcodes.

function [boundingBox, orientation, Idilated] = segmentationLocalization(Ithoughlines)

%-----
% Use image dilation to separate the barcodes
%-----

% Create binary image with the detected lines.
Ibw = ~Ithoughlines(:,:,1);

```



```

Ibw(Ibw > 0) = true;

% Dilate the image using a disk structuring element.
diskRadius = 10; % Might need tuning depending on the input image.
se = strel('disk', diskRadius);
Idilated = imdilate(Ibw, se);

%-----
% Localization parameters for the barcode
%-----

% Compute region properties Orientation and BoundingBox.
regionStatistics = regionprops(Idilated, 'Orientation', 'BoundingBox');

% Padding for the cropped images of barcodes.
padding = 40;

boundingBox = zeros(length(regionStatistics), 4);

for idx = 1:length(regionStatistics)

    boundingBox(idx,:) = regionStatistics(idx).BoundingBox;

    % Bounding box coordinates with padding.
    boundingBox(idx,1) = boundingBox(idx,1) - padding;
    boundingBox(idx,2) = boundingBox(idx,2) - padding;
    boundingBox(idx,3) = boundingBox(idx,3) + 2*padding;
    boundingBox(idx,4) = boundingBox(idx,4) + 2*padding;

end

orientation = [regionStatistics(:).Orientation];

end
    
```

References

[1] Creusot, Clement, et al. "Real-time Barcode Detection in the Wild." IEEE Winter Conference on Applications of Computer Vision, 2015.

Monocular Visual Odometry

Visual odometry is the process of determining the location and orientation of a camera by analyzing a sequence of images. Visual odometry is used in a variety of applications, such as mobile robots, self-driving cars, and unmanned aerial vehicles. This example shows you how to estimate the trajectory of a single calibrated camera from a sequence of images.

Overview

This example shows how to estimate the trajectory of a calibrated camera from a sequence of 2-D views. This example uses images from the New Tsukuba Stereo Dataset created at Tsukuba University's CVLAB. (<https://cvlab.cs.tsukuba.ac.jp>). The dataset consists of synthetic images, generated using computer graphics, and includes the ground truth camera poses.

Without additional information, the trajectory of a monocular camera can only be recovered up to an unknown scale factor. Monocular visual odometry systems used on mobile robots or autonomous vehicles typically obtain the scale factor from another sensor (e.g. wheel odometer or GPS), or from an object of a known size in the scene. This example computes the scale factor from the ground truth.

The example is divided into three parts:

- 1 Estimating the pose of the second view relative to the first view.** Estimate the pose of the second view by estimating the essential matrix and decomposing it into camera location and orientation.
- 2 Bootstrapping estimating camera trajectory using global bundle adjustment.** Eliminate outliers using the epipolar constraint. Find 3D-to-2D correspondences between points triangulated from the previous two views and the current view. Compute the world camera pose for the current view by solving the perspective-n-point (PnP) problem. Estimating the camera poses inevitably results in errors, which accumulate over time. This effect is called *the drift*. To reduce the drift, the example refines all the poses estimated so far using bundle adjustment.
- 3 Estimating remaining camera trajectory using windowed bundle adjustment.** With each new view the time it takes to refine all the poses increases. Windowed bundle adjustment is a way to reduce computation time by only optimizing the last n views, rather than the entire trajectory. Computation time is further reduced by not calling bundle adjustment for every view.

Read Input Image Sequence and Ground Truth

This example uses images from the New Tsukuba Stereo Dataset created at Tsukuba University's CVLAB. If you use these images in your own work or publications, please cite the following papers:

[1] Martin Peris Martorell, Atsuto Maki, Sarah Martull, Yasuhiro Ohkawa, Kazuhiro Fukui, "Towards a Simulation Driven Stereo Vision System". Proceedings of ICPR, pp.1038-1042, 2012.

[2] Sarah Martull, Martin Peris Martorell, Kazuhiro Fukui, "Realistic CG Stereo Image Dataset with Ground Truth Disparity Maps", Proceedings of ICPR workshop TrakMark2012, pp.40-42, 2012.

```
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', 'NewTsukuba'));
% Load ground truth camera poses.
load(fullfile(toolboxdir('vision'), 'visiondata', 'visualOdometryGroundTruth.mat'));
```

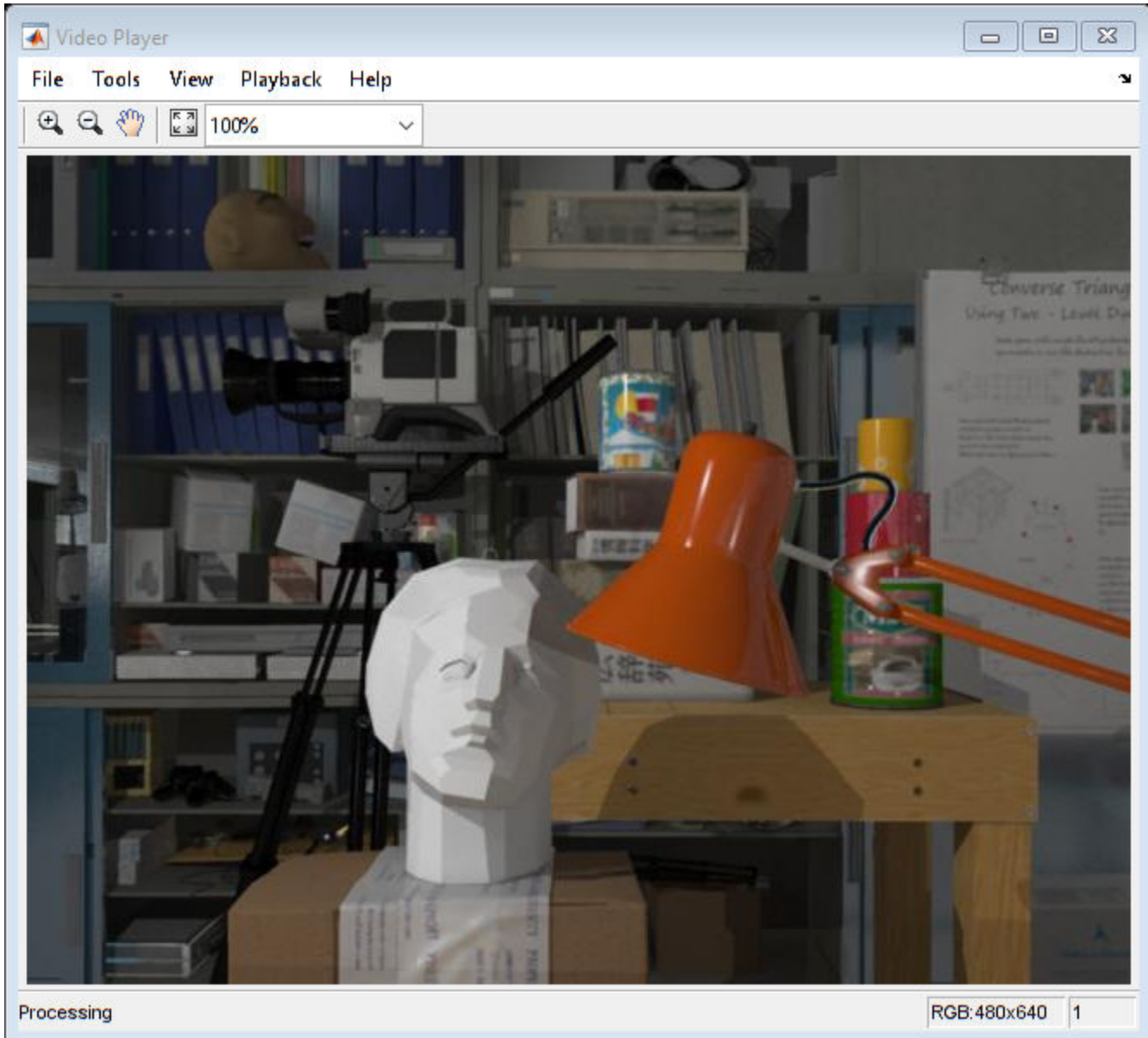
Create a View Set Containing the First View of the Sequence

Use an `imageviewset` object to store and manage the image points and the camera pose associated with each view, as well as point matches between pairs of views. Once you populate an

imageviewset object, you can use it to find point tracks across multiple views and retrieve the camera poses to be used by triangulateMultiview and bundleAdjustment functions.

```
% Create an empty imageviewset object to manage the data associated with each view.
vSet = imageviewset;

% Read and display the first image.
Irgb = readimage(images, 1);
player = vision.VideoPlayer('Position', [20, 400, 650, 510]);
step(player, Irgb);
```



```
% Create the camera intrinsics object using camera intrinsics from the
% New Tsukuba dataset.
focalLength = [615 615]; % specified in units of pixels
```

```
principalPoint = [320 240]; % in pixels [x, y]
imageSize      = size(Irgb,[1,2]); % in pixels [mrows, ncols]
intrinsics     = cameraIntrinsics(focalLength, principalPoint, imageSize);
```

Convert to gray scale and undistort. In this example, undistortion has no effect, because the images are synthetic, with no lens distortion. However, for real images, undistortion is necessary.

```
prevI = undistortImage(rgb2gray(Irgb), intrinsics);

% Detect features.
prevPoints = detectSURFFeatures(prevI, 'MetricThreshold', 500);

% Select a subset of features, uniformly distributed throughout the image.
numPoints = 200;
prevPoints = selectUniform(prevPoints, numPoints, size(prevI));

% Extract features. Using 'Upright' features improves matching quality if
% the camera motion involves little or no in-plane rotation.
prevFeatures = extractFeatures(prevI, prevPoints, 'Upright', true);

% Add the first view. Place the camera associated with the first view
% at the origin, oriented along the Z-axis.
viewId = 1;
vSet = addView(vSet, viewId, rigid3d(eye(3), [0 0 0]), 'Points', prevPoints);
```

Plot Initial Camera Pose

Create two graphical camera objects representing the estimated and the actual camera poses based on ground truth data from the New Tsukuba dataset.

```
% Setup axes.
figure
axis([-220, 50, -140, 20, -50, 300]);

% Set Y-axis to be vertical pointing down.
view(gca, 3);
set(gca, 'CameraUpVector', [0, -1, 0]);
camorbit(gca, -120, 0, 'data', [0, 1, 0]);

grid on
xlabel('X (cm)');
ylabel('Y (cm)');
zlabel('Z (cm)');
hold on

% Plot estimated camera pose.
cameraSize = 7;
camPose = poses(vSet);
camEstimated = plotCamera(camPose, 'Size', cameraSize, ...
    'Color', 'g', 'Opacity', 0);

% Plot actual camera pose.
camActual = plotCamera('Size', cameraSize, 'AbsolutePose', ...
    rigid3d(groundTruthPoses.Orientation{1}, groundTruthPoses.Location{1}), ...
    'Color', 'b', 'Opacity', 0);

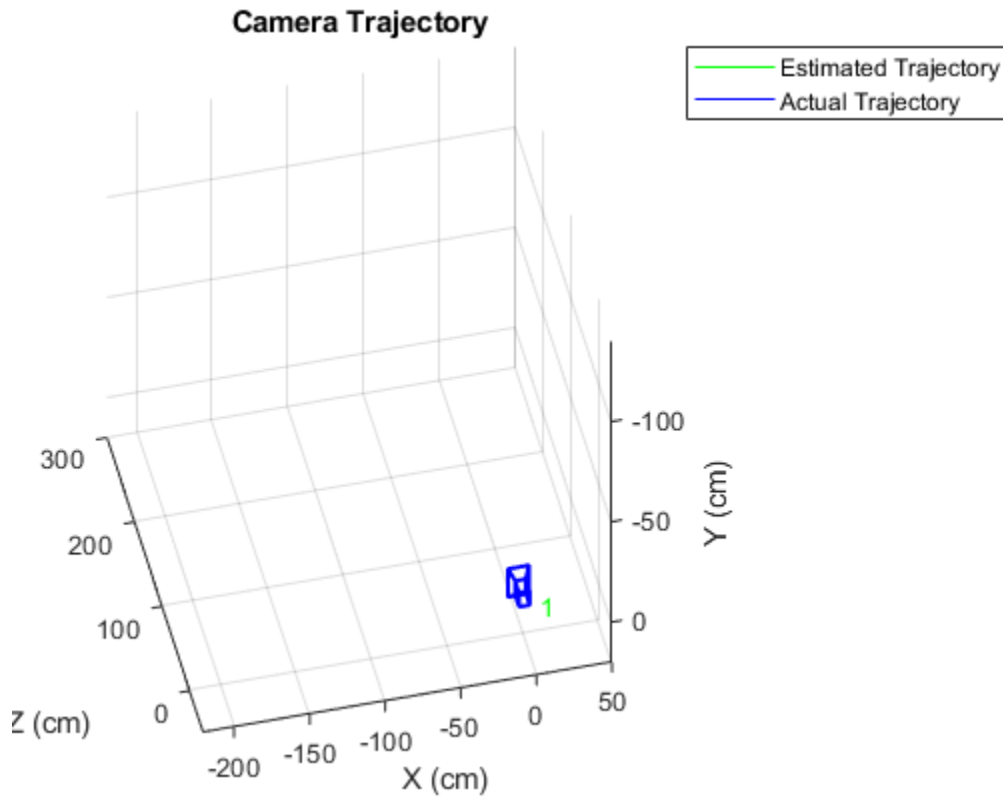
% Initialize camera trajectories.
trajectoryEstimated = plot3(0, 0, 0, 'g-');
```

```

trajectoryActual = plot3(0, 0, 0, 'b-');

legend('Estimated Trajectory', 'Actual Trajectory');
title('Camera Trajectory');

```



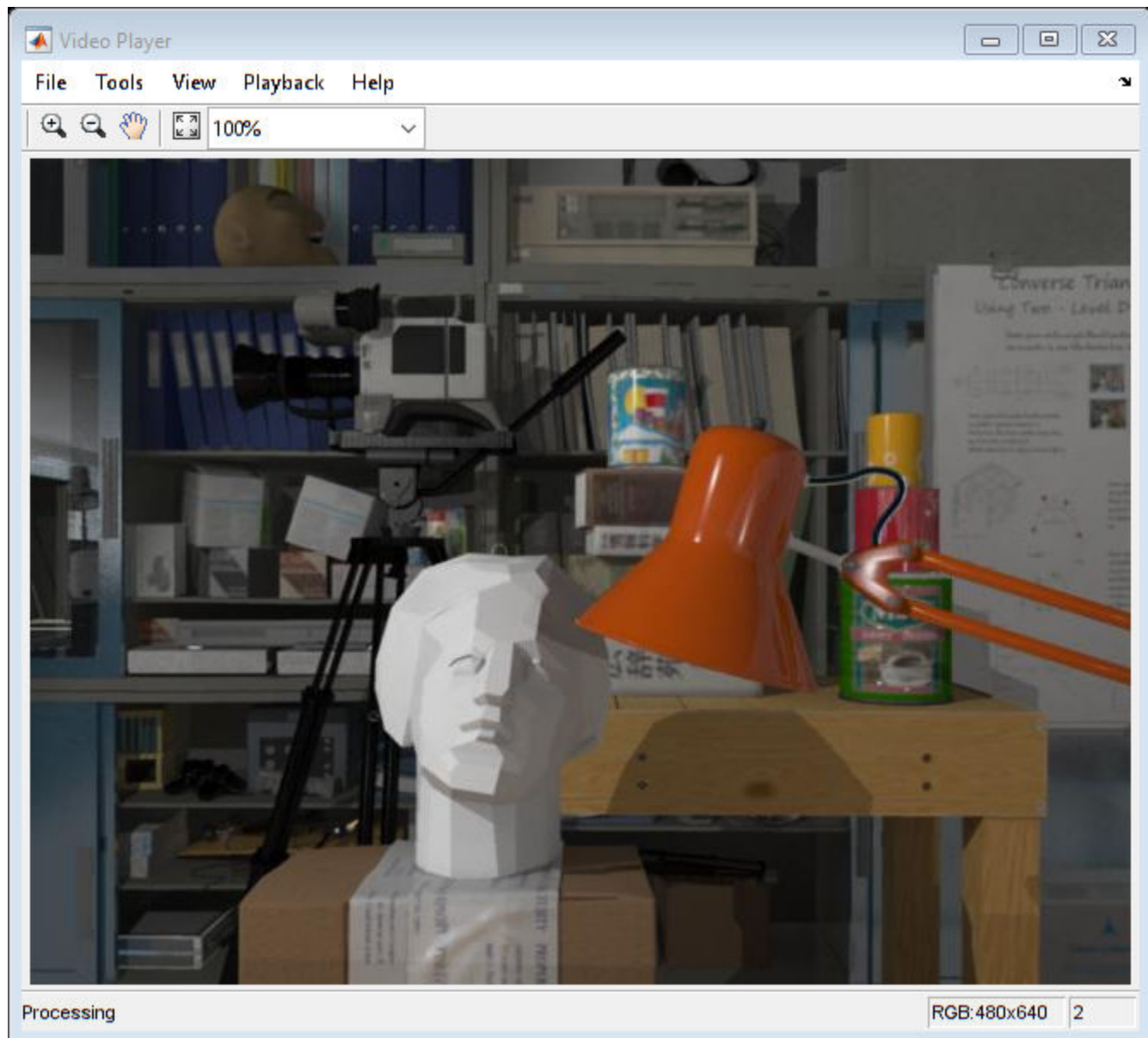
Estimate the Pose of the Second View

Detect and extract features from the second view, and match them to the first view using `helperDetectAndMatchFeatures`. Estimate the pose of the second view relative to the first view using `helperEstimateRelativePose`, and add it to the `imageviewset`.

```

% Read and display the image.
viewId = 2;
Irgb = readimage(images, viewId);
step(player, Irgb);

```



```

% Convert to gray scale and undistort.
I = undistortImage(rgb2gray(Irgb), intrinsics);

% Match features between the previous and the current image.
[currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(...
    prevFeatures, I);

% Estimate the pose of the current view relative to the previous view.
[orient, loc, inlierIdx] = helperEstimateRelativePose(...
    prevPoints(indexPairs(:,1)), currPoints(indexPairs(:,2)), intrinsics);

% Exclude epipolar outliers.
indexPairs = indexPairs(inlierIdx, :);

```

```
% Add the current view to the view set.
vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);
```

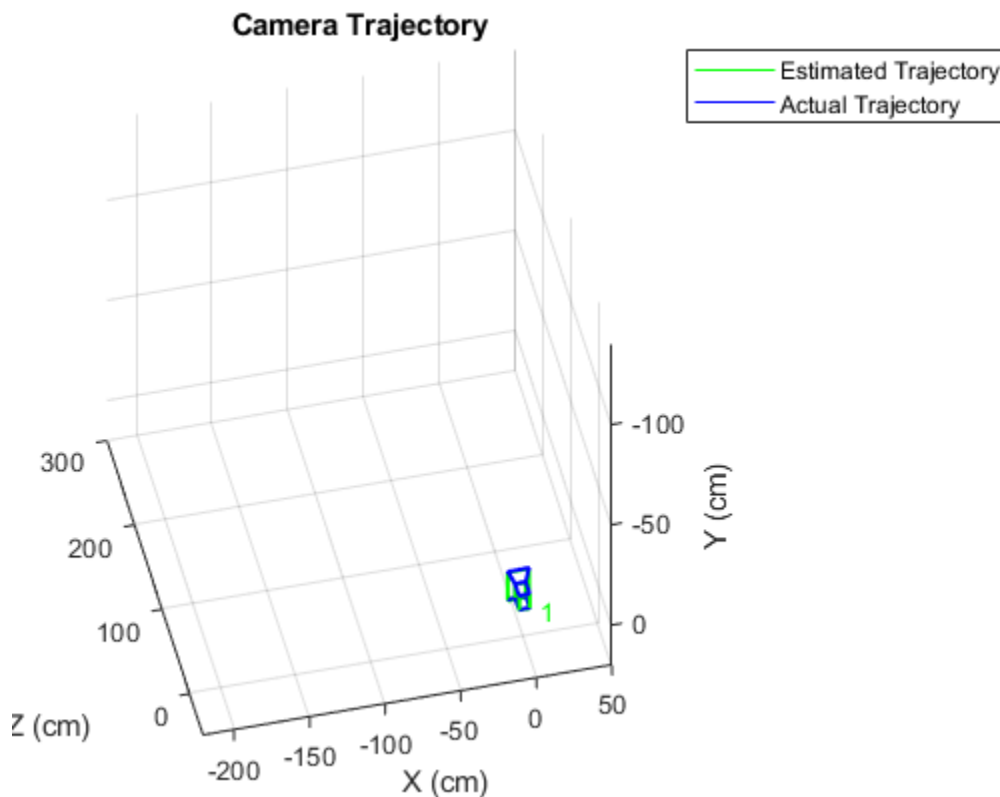
```
% Store the point matches between the previous and the current views.
vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);
```

The location of the second view relative to the first view can only be recovered up to an unknown scale factor. Compute the scale factor from the ground truth using `helperNormalizeViewSet`, simulating an external sensor, which would be used in a typical monocular visual odometry system.

```
vSet = helperNormalizeViewSet(vSet, groundTruthPoses);
```

Update camera trajectory plots using `helperUpdateCameraPlots` and `helperUpdateCameraTrajectories`.

```
helperUpdateCameraPlots(viewId, camEstimated, camActual, poses(vSet), ...
    groundTruthPoses);
helperUpdateCameraTrajectories(viewId, trajectoryEstimated, trajectoryActual, ...
    poses(vSet), groundTruthPoses);
```



```
prevI = I;
prevFeatures = currFeatures;
prevPoints = currPoints;
```

Bootstrap Estimating Camera Trajectory Using Global Bundle Adjustment

Find 3D-to-2D correspondences between world points triangulated from the previous two views and image points from the current view. Use `helperFindEpipolarInliers` to find the matches that satisfy the epipolar constraint, and then use `helperFind3Dto2DCorrespondences` to triangulate 3-D points from the previous two views and find the corresponding 2-D points in the current view.

Compute the world camera pose for the current view by solving the perspective-n-point (PnP) problem using `estimateWorldCameraPose`. For the first 15 views, use global bundle adjustment to refine the entire trajectory. Using global bundle adjustment for a limited number of views bootstraps estimating the rest of the camera trajectory, and it is not prohibitively expensive.

```
for viewId = 3:15
    % Read and display the next image
    Irgb = readimage(images, viewId);
    step(player, Irgb);

    % Convert to gray scale and undistort.
    I = undistortImage(rgb2gray(Irgb), intrinsics);

    % Match points between the previous and the current image.
    [currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(...
        prevFeatures, I);

    % Eliminate outliers from feature matches.
    inlierIdx = helperFindEpipolarInliers(prevPoints(indexPairs(:,1)),...
        currPoints(indexPairs(:, 2)), intrinsics);
    indexPairs = indexPairs(inlierIdx, :);

    % Triangulate points from the previous two views, and find the
    % corresponding points in the current view.
    [worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet,...
        intrinsics, indexPairs, currPoints);

    % Since RANSAC involves a stochastic process, it may sometimes not
    % reach the desired confidence level and exceed maximum number of
    % trials. Disable the warning when that happens since the outcomes are
    % still valid.
    warningstate = warning('off','vision:ransac:maxTrialsReached');

    % Estimate the world camera pose for the current view.
    [orient, loc] = estimateWorldCameraPose(imagePoints, worldPoints, intrinsics);

    % Restore the original warning state
    warning(warningstate)

    % Add the current view to the view set.
    vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);

    % Store the point matches between the previous and the current views.
    vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);

    tracks = findTracks(vSet); % Find point tracks spanning multiple views.

    camPoses = poses(vSet); % Get camera poses for all views.

    % Triangulate initial locations for the 3-D world points.
```



```

xyzPoints = triangulateMultiview(tracks, camPoses, intrinsics);

% Refine camera poses using bundle adjustment.
[~, camPoses] = bundleAdjustment(xyzPoints, tracks, camPoses, ...
    intrinsics, 'PointsUndistorted', true, 'AbsoluteTolerance', 1e-12, ...
    'RelativeTolerance', 1e-12, 'MaxIterations', 200, 'FixedViewID', 1);

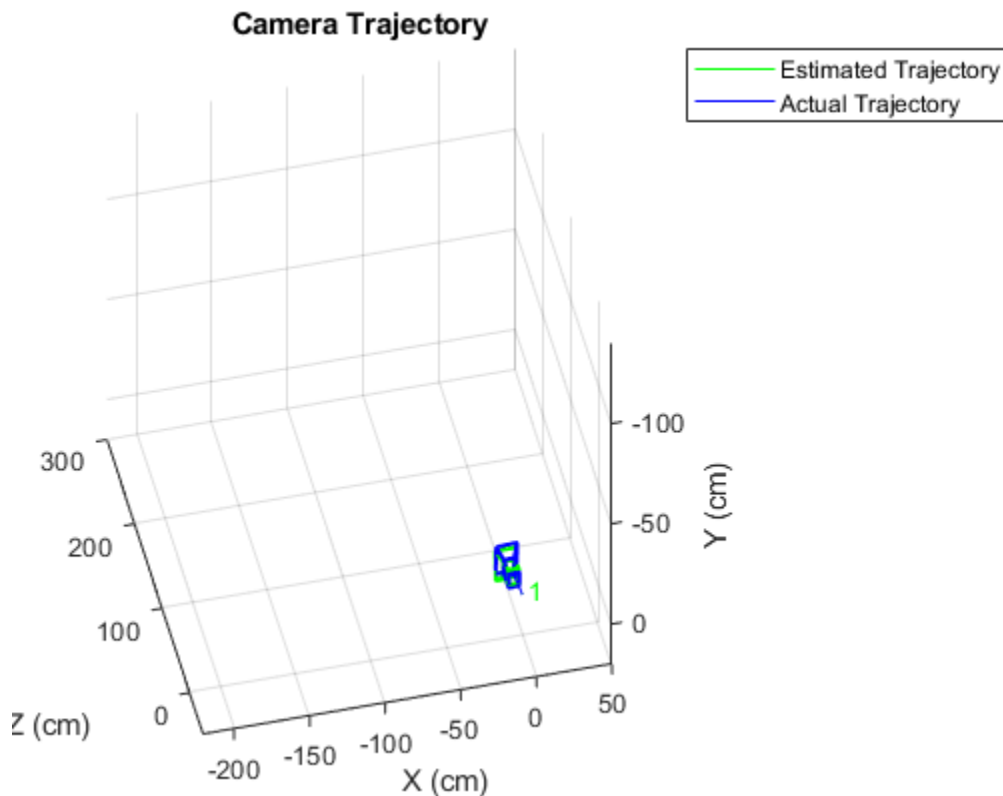
vSet = updateView(vSet, camPoses); % Update view set.

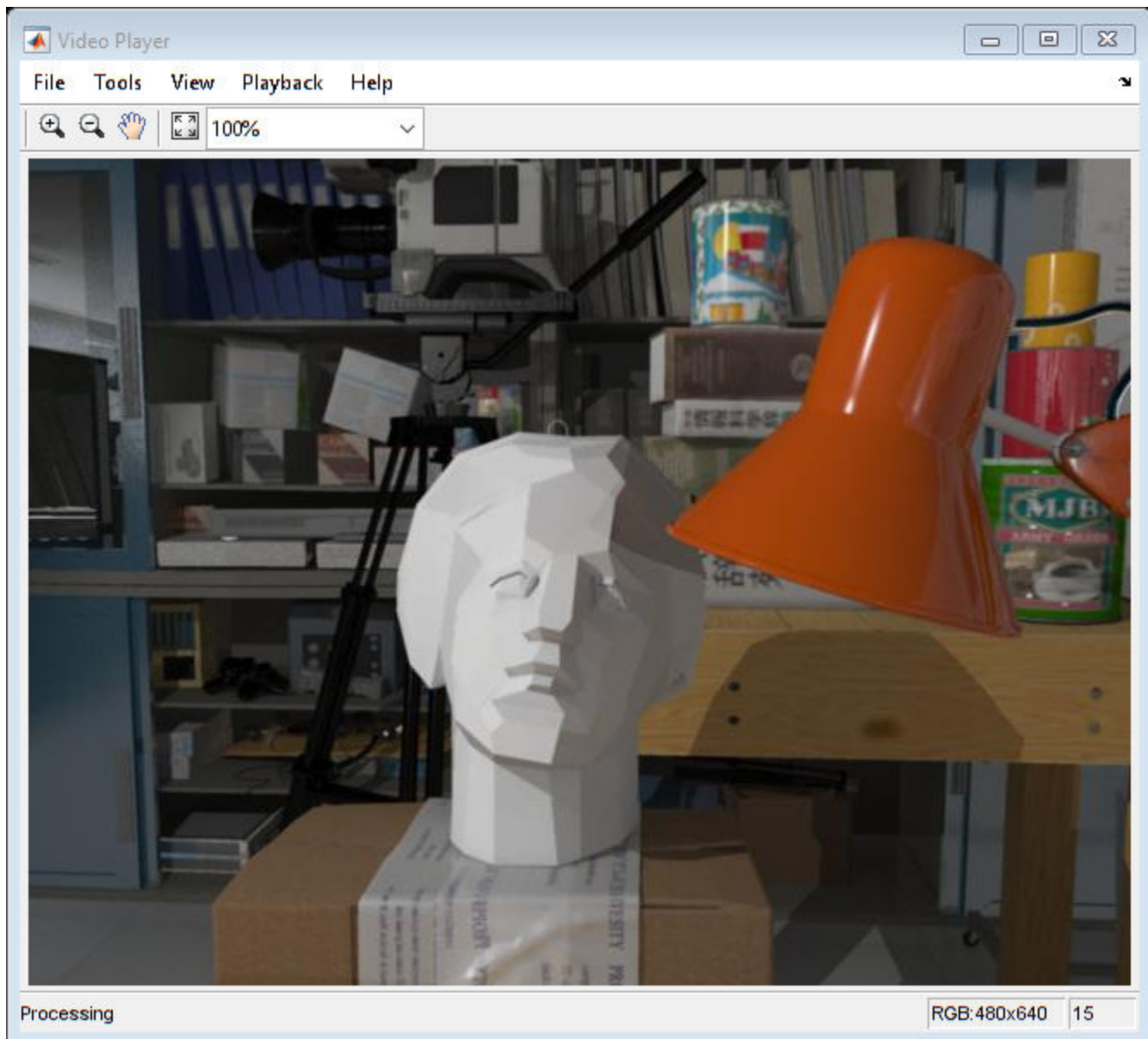
% Bundle adjustment can move the entire set of cameras. Normalize the
% view set to place the first camera at the origin looking along the
% Z-axis and adjust the scale to match that of the ground truth.
vSet = helperNormalizeViewSet(vSet, groundTruthPoses);

% Update camera trajectory plot.
helperUpdateCameraPlots(viewId, camEstimated, camActual, poses(vSet), ...
    groundTruthPoses);
helperUpdateCameraTrajectories(viewId, trajectoryEstimated, ...
    trajectoryActual, poses(vSet), groundTruthPoses);

prevI = I;
prevFeatures = currFeatures;
prevPoints = currPoints;
end

```





Estimate Remaining Camera Trajectory Using Windowed Bundle Adjustment

Estimate the remaining camera trajectory by using windowed bundle adjustment to only refine the last 15 views, in order to limit the amount of computation. Furthermore, bundle adjustment does not have to be called for every view, because `estimateWorldCameraPose` computes the pose in the same units as the 3-D points. This section calls bundle adjustment for every 7th view. The window size and the frequency of calling bundle adjustment have been chosen experimentally.

```
for viewId = 16:numel(images.Files)
    % Read and display the next image
    Irgb = readimage(images, viewId);
    step(player, Irgb);

    % Convert to gray scale and undistort.
```

```

I = undistortImage(rgb2gray(Irgb), intrinsics);

% Match points between the previous and the current image.
[currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(...
    prevFeatures, I);

% Triangulate points from the previous two views, and find the
% corresponding points in the current view.
[worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet, ...
    intrinsics, indexPairs, currPoints);

% Since RANSAC involves a stochastic process, it may sometimes not
% reach the desired confidence level and exceed maximum number of
% trials. Disable the warning when that happens since the outcomes are
% still valid.
warningstate = warning('off', 'vision:ransac:maxTrialsReached');

% Estimate the world camera pose for the current view.
[orient, loc] = estimateWorldCameraPose(imagePoints, worldPoints, intrinsics);

% Restore the original warning state
warning(warningstate)

% Add the current view and connection to the view set.
vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);
vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);

% Refine estimated camera poses using windowed bundle adjustment. Run
% the optimization every 7th view.
if mod(viewId, 7) == 0
    % Find point tracks in the last 15 views and triangulate.
    windowSize = 15;
    startFrame = max(1, viewId - windowSize);
    tracks = findTracks(vSet, startFrame:viewId);
    camPoses = poses(vSet, startFrame:viewId);
    [xyzPoints, reprojErrors] = triangulateMultiview(tracks, camPoses, intrinsics);

    % Hold the first two poses fixed, to keep the same scale.
    fixedIds = [startFrame, startFrame+1];

    % Exclude points and tracks with high reprojection errors.
    idx = reprojErrors < 2;

    [~, camPoses] = bundleAdjustment(xyzPoints(idx, :), tracks(idx), ...
        camPoses, intrinsics, 'FixedViewIDs', fixedIds, ...
        'PointsUndistorted', true, 'AbsoluteTolerance', 1e-12, ...
        'RelativeTolerance', 1e-12, 'MaxIterations', 200);

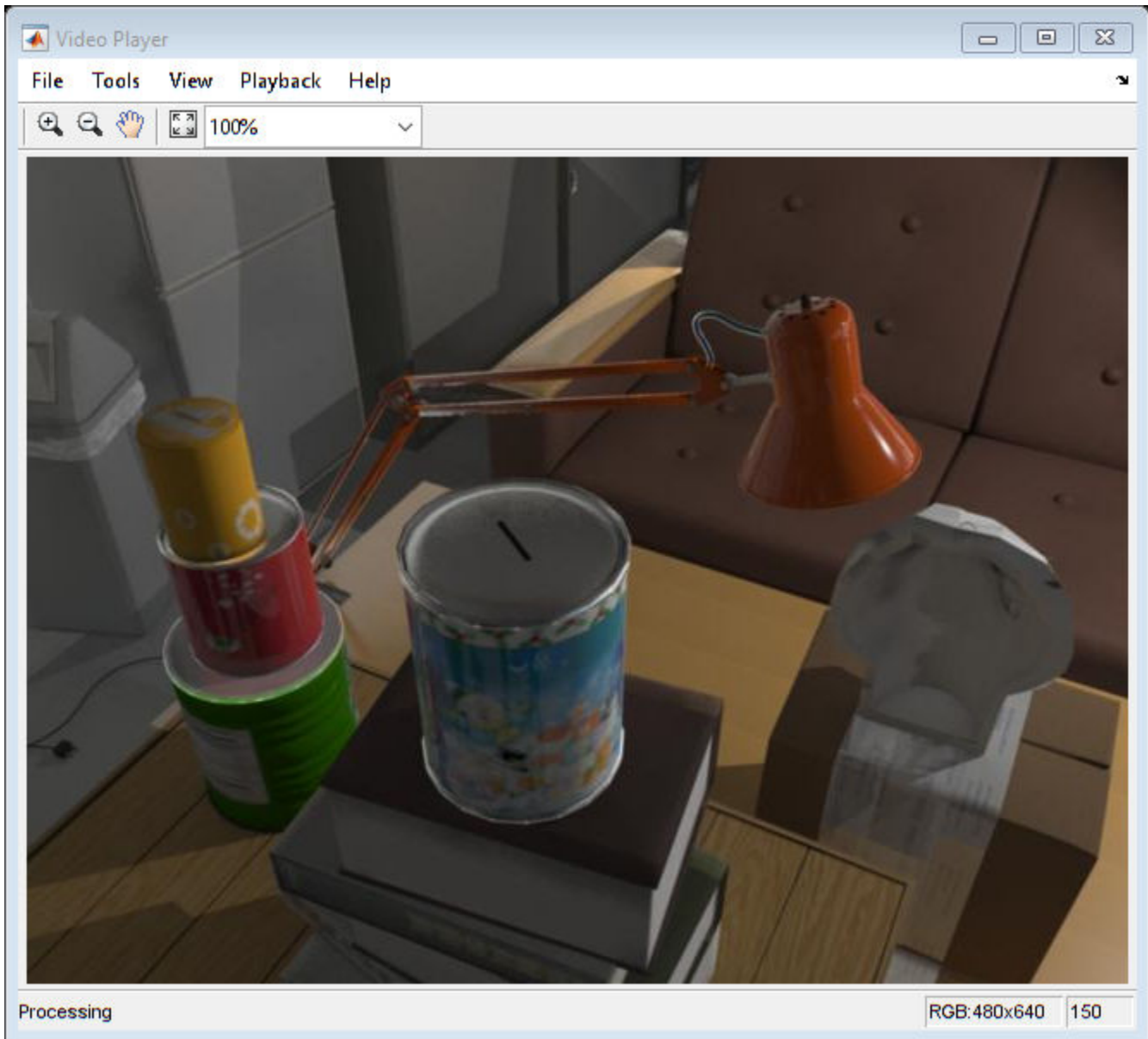
    vSet = updateView(vSet, camPoses); % Update view set.
end

% Update camera trajectory plot.
helperUpdateCameraPlots(viewId, camEstimated, camActual, poses(vSet), ...
    groundTruthPoses);
helperUpdateCameraTrajectories(viewId, trajectoryEstimated, ...
    trajectoryActual, poses(vSet), groundTruthPoses);

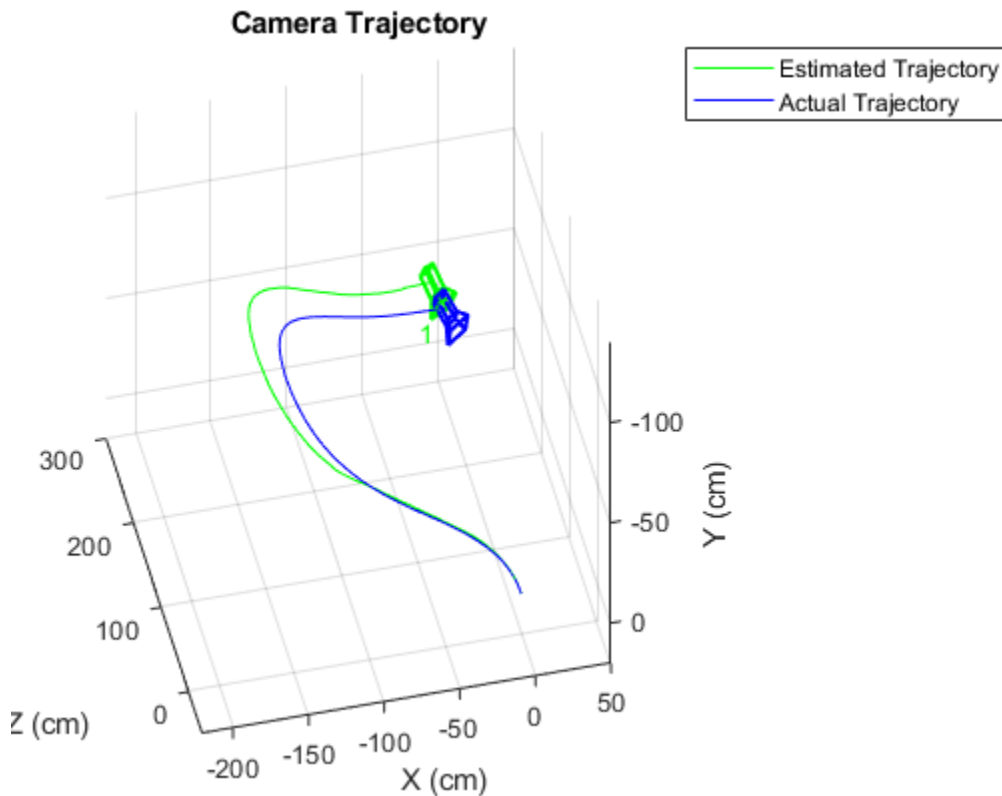
prevI = I;

```

```
    prevFeatures = currFeatures;  
    prevPoints  = currPoints;  
end
```



hold off



Summary

This example showed how to estimate the trajectory of a calibrated monocular camera from a sequence of views. Notice that the estimated trajectory does not exactly match the ground truth. Despite the non-linear refinement of camera poses, errors in camera pose estimation accumulate, resulting in drift. In visual odometry systems this problem is typically addressed by fusing information from multiple sensors, and by performing loop closure.

References

- [1] Martin Peris Martorell, Atsuto Maki, Sarah Martull, Yasuhiro Ohkawa, Kazuhiro Fukui, "Towards a Simulation Driven Stereo Vision System". Proceedings of ICPR, pp.1038-1042, 2012.
- [2] Sarah Martull, Martin Peris Martorell, Kazuhiro Fukui, "Realistic CG Stereo Image Dataset with Ground Truth Disparity Maps", Proceedings of ICPR workshop TrakMark2012, pp.40-42, 2012.
- [3] M.I.A. Lourakis and A.A. Argyros (2009). "SBA: A Software Package for Generic Sparse Bundle Adjustment". ACM Transactions on Mathematical Software (ACM) 36 (1): 1-30.
- [4] R. Hartley, A. Zisserman, "Multiple View Geometry in Computer Vision," Cambridge University Press, 2003.
- [5] B. Triggs; P. McLauchlan; R. Hartley; A. Fitzgibbon (1999). "Bundle Adjustment: A Modern Synthesis". Proceedings of the International Workshop on Vision Algorithms. Springer-Verlag. pp. 298-372.

[6] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, "Complete Solution Classification for the Perspective-Three-Point Problem," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 25, no. 8, pp. 930-943, 2003.

Detect and Track Vehicles Using Lidar Data

This example shows you how to track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Lidar sensors report measurements as a point cloud. The example illustrates the workflow in MATLAB® for processing the point cloud and tracking the objects. For a Simulink® version of the example, refer to “Track Vehicles Using Lidar Data in Simulink” (Sensor Fusion and Tracking Toolbox). The lidar data used in this example is recorded from a highway driving scenario. In this example, you use the recorded data to track vehicles with a joint probabilistic data association (JPDA) tracker and an interacting multiple model (IMM) approach.

3-D Bounding Box Detector Model

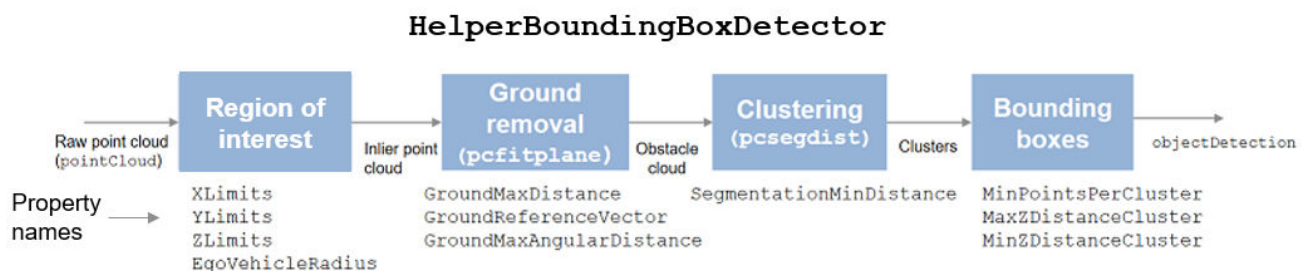
Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrians. For more details about segmentation of lidar data into objects such as the ground plane and obstacles, refer to the “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox) example. In this example, the point clouds belonging to obstacles are further classified into clusters using the `pcsegdist` function, and each cluster is converted to a bounding box detection with the following format:

$$[x \ y \ z \ l \ w \ h]$$

x , y and z refer to the x-, y- and z-positions of the bounding box and l , w and h refer to its length, width, and height, respectively.

The bounding box is fit onto each cluster by using minimum and maximum of coordinates of points in each dimension. The detector is implemented by a supporting class `HelperBoundingBoxDetector`, which wraps around point cloud segmentation and clustering functionalities. An object of this class accepts a `pointCloud` input and returns a list of `objectDetection` objects with bounding box measurements.

The diagram shows the processes involved in the bounding box detector model and the Computer Vision Toolbox™ functions used to implement each process. It also shows the properties of the supporting class that control each process.



The lidar data is available at the following link: <https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleData.zip>

Download the data into your temporary directory, whose location is specified by MATLAB's `tempdir` function. If you want to place the files in a different folder, change the directory name in subsequent instructions.

```
% Load data if unavailable. The lidar data is stored as a cell array of
% pointCloud objects.
if ~exist('lidarData','var')
    dataURL = 'https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleD';
    datasetFolder = fullfile(tempdir,'LidarExampleDataset');
    if ~exist(datasetFolder,'dir')
        unzip(dataURL,datasetFolder);
    end
    % Specify initial and final time for simulation.
    initTime = 0;
    finalTime = 35;
    [lidarData, imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime);
end

% Set random seed to generate reproducible results.
S = rng(2018);

% A bounding box detector model.
detectorModel = HelperBoundingBoxDetector(...
    'XLimits',[-50 75],...           % min-max
    'YLimits',[-5 5],...           % min-max
    'ZLimits',[-2 5],...           % min-max
    'SegmentationMinDistance',1.6,... % minimum Euclidian distance
    'MinDetectionsPerCluster',1,... % minimum points per cluster
    'MeasurementNoise',eye(6),... % measurement noise in detection report
    'GroundMaxDistance',0.3);      % maximum distance of ground points from ground plane
```

Target State and Sensor Measurement Model

The first step in tracking an object is defining its state, and the models that define the transition of state and the corresponding measurement. These two sets of equations are collectively known as the state-space model of the target. To model the state of vehicles for tracking using lidar, this example uses a cuboid model with following convention:

$$x = [x_{kin} \ \theta \ l \ w \ h]$$

x_{kin} refers to the portion of the state that controls the kinematics of the motion center, and θ is the yaw angle. The length, width, height of the cuboid are modeled as a constants, whose estimates evolve in time during correction stages of the filter.

In this example, you use two state-space models: a constant velocity (cv) cuboid model and a constant turn-rate (ct) cuboid model. These models differ in the way they define the kinematic part of the state, as described below:

$$x_{cv} = [x \ \dot{x} \ y \ \dot{y} \ z \ \dot{z} \ \theta \ l \ w \ h]$$

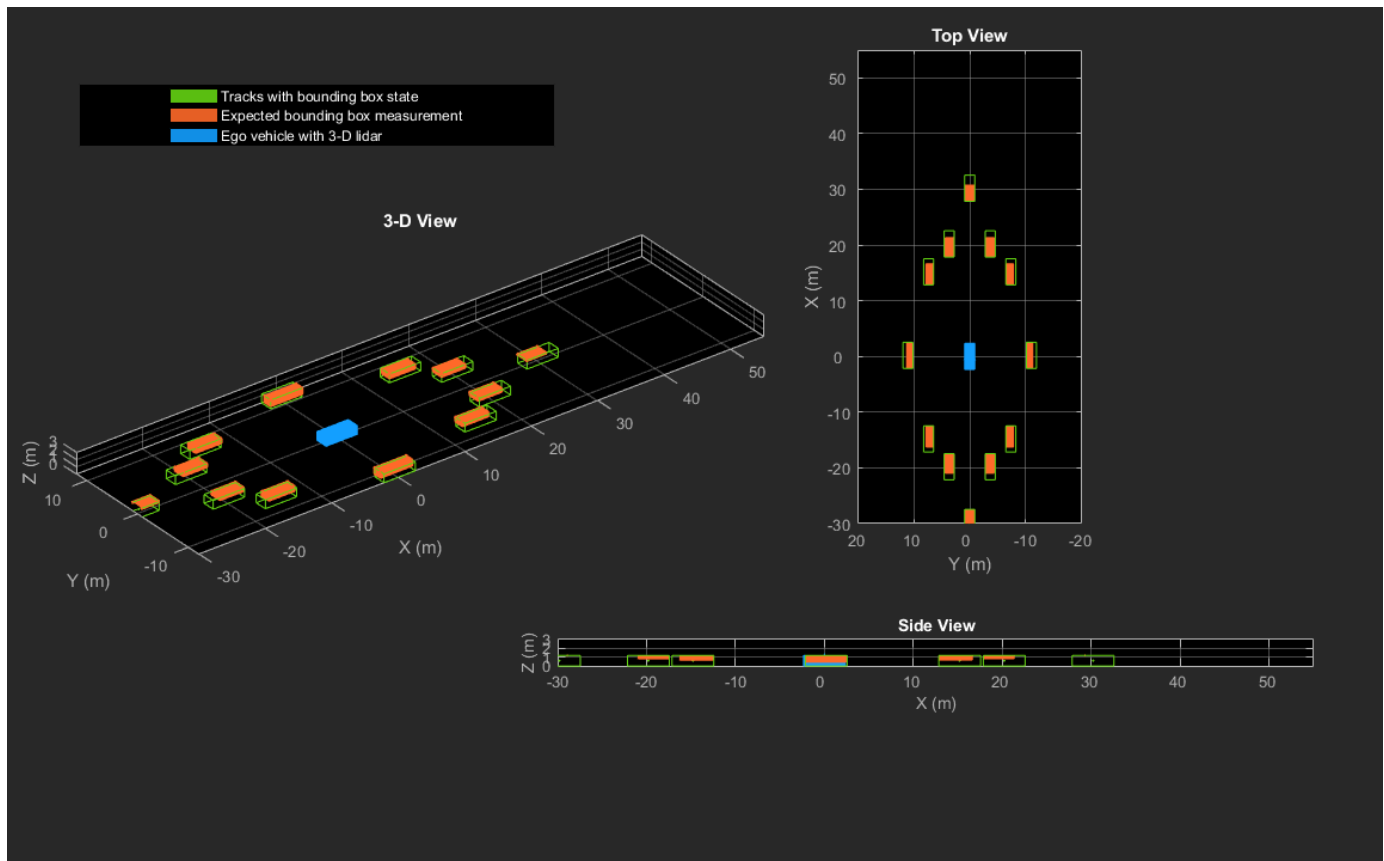
$$x_{ct} = [x \ \dot{x} \ y \ \dot{y} \ \dot{\theta} \ z \ \dot{z} \ \theta \ l \ w \ h]$$

For information about their state transition, refer to the `helperConstvelCuboid` and `helperConstturnCuboid` functions used in this example.

The `helperCvmeasCuboid` and `helperCtmeasCuboid` measurement models describe how the sensor perceives the constant velocity and constant turn-rate states respectively, and they return

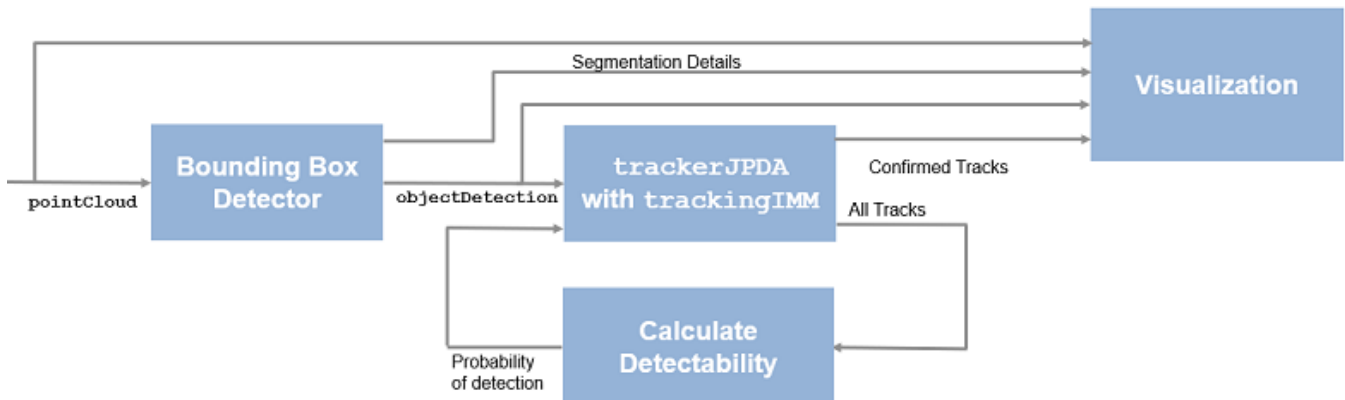
bounding box measurements. Because the state contains information about size of the target, the measurement model includes the effect of center-point offset and bounding box shrinkage, as perceived by the sensor, due to effects like self-occlusion [1]. This effect is modeled by a shrinkage factor that is directly proportional to the distance from the tracked vehicle to the sensor.

The image below demonstrates the measurement model operating at different state-space samples. Notice the modeled effects of bounding box shrinkage and center-point offset as the objects move around the ego vehicle.



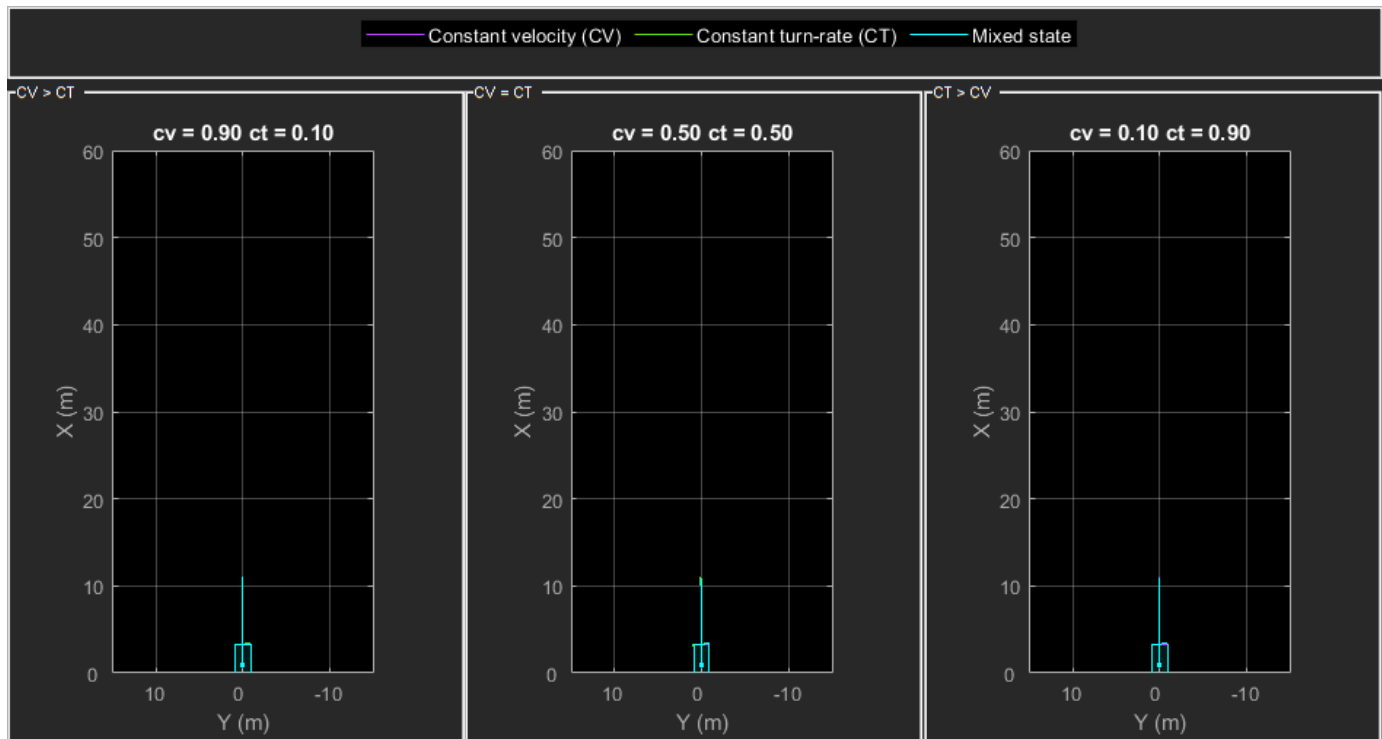
Set Up Tracker and Visualization

The image below shows the complete workflow to obtain a list of tracks from a pointCloud input.

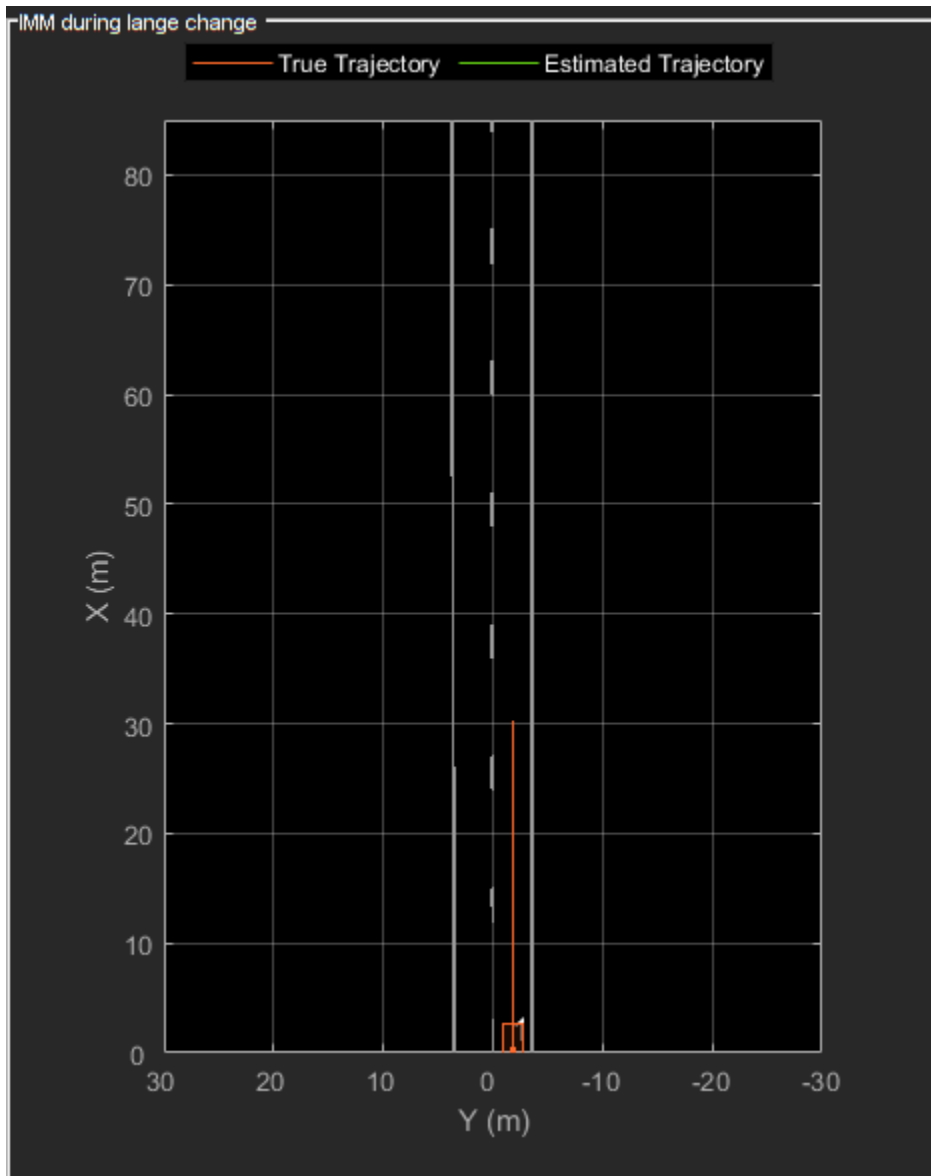


Now, set up the tracker and the visualization used in the example.

A joint probabilistic data association tracker (`trackerJPDA`) coupled with an IMM filter (`trackingIMM`) is used to track objects in this example. The IMM filter uses a constant velocity and constant turn-rate model and is initialized using the supporting function, `helperInitIMMFilter`, included with this example. The IMM approach helps a track to switch between motion models and thus achieve good estimation accuracy during events like maneuvering or lane changing. The animation below shows the effect of mixing the constant velocity and constant turn-rate model during prediction stages of the IMM filter.



The IMM filter updates the probability of each model when it is corrected with detections from the object. The animation below shows the estimated trajectory of a vehicle during a lane change event and the corresponding estimated probabilities of each model.



Set the `HasDetectableTrackIDsInput` property of the tracker as `true`, which enables you to specify a state-dependent probability of detection. The detection probability of a track is calculated by the `helperCalcDetectability` function, listed at the end of this example.

```
assignmentGate = [50 100]; % Assignment threshold;
confThreshold = [7 10]; % Confirmation threshold for history logic
delThreshold = [8 10]; % Deletion threshold for history logic
Kc = 1e-5; % False-alarm rate per unit volume

% IMM filter initialization function
filterInitFcn = @helperInitIMMFilter;

% A joint probabilistic data association tracker with IMM filter
tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
    'TrackLogic','History',...
    'AssignmentThreshold',assignmentGate,...
```

```
'ClutterDensity',Kc,...
'ConfirmationThreshold',confThreshold,...
'DeletionThreshold',delThreshold,...
'HasDetectableTrackIDsInput',true,...
'InitializationThreshold',0);
```

The visualization is divided into these main categories:

- 1 Lidar Preprocessing and Tracking - This display shows the raw point cloud, segmented ground, and obstacles. It also shows the resulting detections from the detector model and the tracks of vehicles generated by the tracker.
- 2 Ego Vehicle Display - This display shows the 2-D bird's-eye view of the scenario. It shows the obstacle point cloud, bounding box detections, and the tracks generated by the tracker. For reference, it also displays the image recorded from a camera mounted on the ego vehicle and its field of view.
- 3 Tracking Details - This display shows the scenario zoomed around the ego vehicle. It also shows finer tracking details, such as error covariance in estimated position of each track and its motion model probabilities, denoted by cv and ct.

```
% Create display
displayObject = HelperLidarExampleDisplay(imageData{1},...
'PositionIndex',[1 3 6],...
'VelocityIndex',[2 4 7],...
'DimensionIndex',[9 10 11],...
'YawIndex',8,...
'MovieName','',... % Specify a movie name to record a movie.
'RecordGIF',false); % Specify true to record new GIFs
```

Loop Through Data

Loop through the recorded lidar data, generate detections from the current point cloud using the detector model and then process the detections using the tracker.

```
time = 0; % Start time
dT = 0.1; % Time step

% Initiate all tracks.
allTracks = struct([]);

% Initiate variables for comparing MATLAB and MEX simulation.
numTracks = zeros(numel(lidarData),2);

% Loop through the data
for i = 1:numel(lidarData)
    % Update time
    time = time + dT;

    % Get current lidar scan
    currentLidar = lidarData{i};

    % Generator detections from lidar scan.
    [detections,obstacleIndices,groundIndices,croppedIndices] = detectorModel(currentLidar,time)

    % Calculate detectability of each track.
    detectableTracksInput = helperCalcDetectability(allTracks,[1 3 6]);

    % Pass detections to track.
```

```

[confirmedTracks,tentativeTracks,allTracks] = tracker(detections,time,detectableTracksInput)
numTracks(i,1) = numel(confirmedTracks);

% Get model probabilities from IMM filter of each track using
% getTrackFilterProperties function of the tracker.
modelProbs = zeros(2,numel(confirmedTracks));
for k = 1:numel(confirmedTracks)
    c1 = getTrackFilterProperties(tracker,confirmedTracks(k).TrackID,'ModelProbabilities');
    modelProbs(:,k) = c1{1};
end

% Update display
if isvalid(displayObject.PointCloudProcessingDisplay.ObstaclePlotter)
    % Get current image scan for reference image
    currentImage = imageData{i};

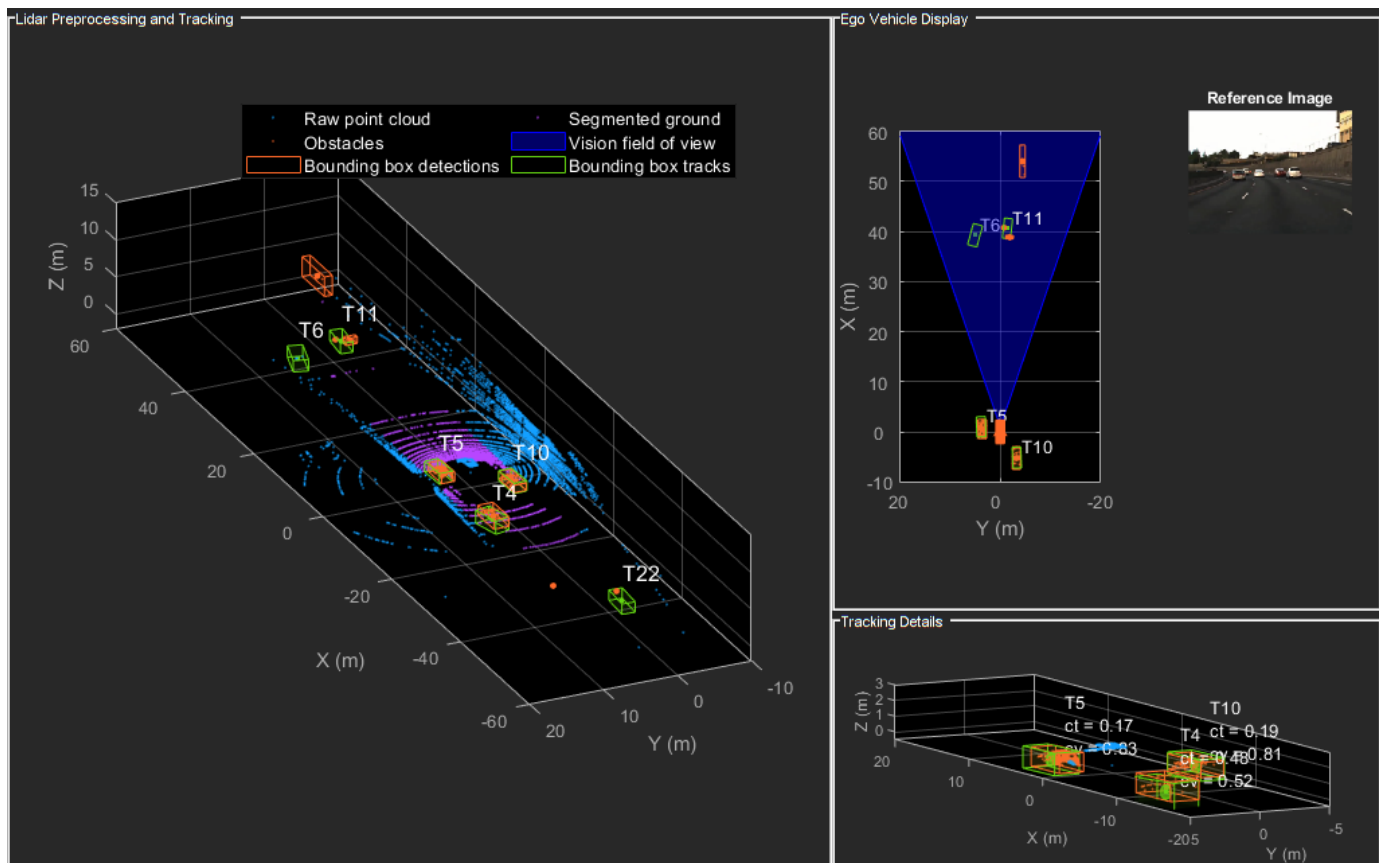
    % Update display object
    displayObject(detections,confirmedTracks,currentLidar,obstacleIndices,...
        groundIndices,croppedIndices,currentImage,modelProbs);
end

% Snap a figure at time = 18
if abs(time - 18) < dT/2
    snapnow(displayObject);
end

% Write movie if requested
if ~isempty(displayObject.MovieName)
    writeMovie(displayObject);
end

% Write new GIFs if requested.
if displayObject.RecordGIF
    % second input is start frame, third input is end frame and last input
    % is a character vector specifying the panel to record.
    writeAnimatedGIF(displayObject,10,170,'trackMaintenance','ego');
    writeAnimatedGIF(displayObject,310,330,'jpda','processing');
    writeAnimatedGIF(displayObject,150,180,'imm','details');
end

```



The figure above shows the three displays at time = 18 seconds. The tracks are represented by green bounding boxes. The bounding box detections are represented by orange bounding boxes. The detections also have orange points inside them, representing the point cloud segmented as obstacles. The segmented ground is shown in purple. The cropped or discarded point cloud is shown in blue.

Generate C Code

You can generate C code from the MATLAB® code for the tracking and the preprocessing algorithm using MATLAB Coder™. C code generation enables you to accelerate MATLAB code for simulation. To generate C code, the algorithm must be restructured as a MATLAB function, which can be compiled into a MEX file or a shared library. For this purpose, the point cloud processing algorithm and the tracking algorithm is restructured into a MATLAB function, `mexLidarTracker`. Some variables are defined as `persistent` to preserve their state between multiple calls to the function (see `persistent`). The inputs and outputs of the function can be observed in the function description provided in the "Supporting Files" section at the end of this example.

MATLAB coder requires specifying the properties of all the input arguments. An easy way to do this is by defining the input properties by example at the command line using the `-args` option. For more information, see "Define Input Properties by Example at the Command Line" (MATLAB Coder). Note that the top-level input arguments cannot be objects of the `handle` class. Therefore, the function accepts the `x`, `y` and `z` locations of the point cloud as an input. From the stored point cloud, this information can be extracted using the `Location` property of the `pointCloud` object. This information is also directly available as the raw data from the lidar sensor.

```

% Input lists
inputExample = {lidarData{1}.Location, 0};

% Create configuration for MEX generation
cfg = coder.config('mex');

% Replace cfg with the following to generate static library and perform
% software-in-the-loop simulation. This requires Embedded Coder license.
%
% cfg = coder.config('lib'); % Static library
% cfg.VerificationMode = 'SIL'; % Software-in-the-loop

% Generate code if file does not exist.
if ~exist('mexLidarTracker_mex','file')
    h = msgbox({'Generating code. This may take a few minutes...'; 'This message box will close w
    % -config allows specifying the codegen configuration
    % -o allows specifying the name of the output file
    codegen -config cfg -o mexLidarTracker_mex mexLidarTracker -args inputExample
    close(h);
else
    clear mexLidarTracker_mex;
end

```

Rerun simulation with MEX Code

Rerun the simulation using the generated MEX code, mexLidarTracker_mex.

```

% Reset time
time = 0;

for i = 1:numel(lidarData)
    time = time + dT;

    currentLidar = lidarData{i};

    [detectionsMex, obstacleIndicesMex, groundIndicesMex, croppedIndicesMex, ...
    confirmedTracksMex, modelProbsMex] = mexLidarTracker_mex(currentLidar.Location, time);

    % Record data for comparison with MATLAB execution.
    numTracks(i,2) = numel(confirmedTracksMex);
end

```

Compare results between MATLAB and MEX Execution

```
disp(isequal(numTracks(:,1), numTracks(:,2)));
```

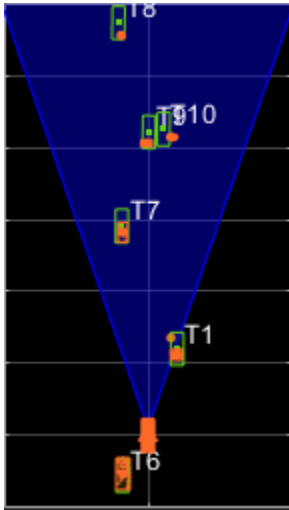
```
1
```

Notice that the number of confirmed tracks is the same for MATLAB and MEX code execution. This assures that the lidar preprocessing and tracking algorithm returns the same results with generated C code as with the MATLAB code.

Results

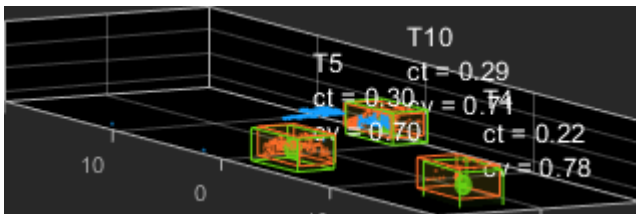
Now, analyze different events in the scenario and understand how the combination of lidar measurement model, joint probabilistic data association, and interacting multiple model filter, helps achieve a good estimation of the vehicle tracks.

Track Maintenance



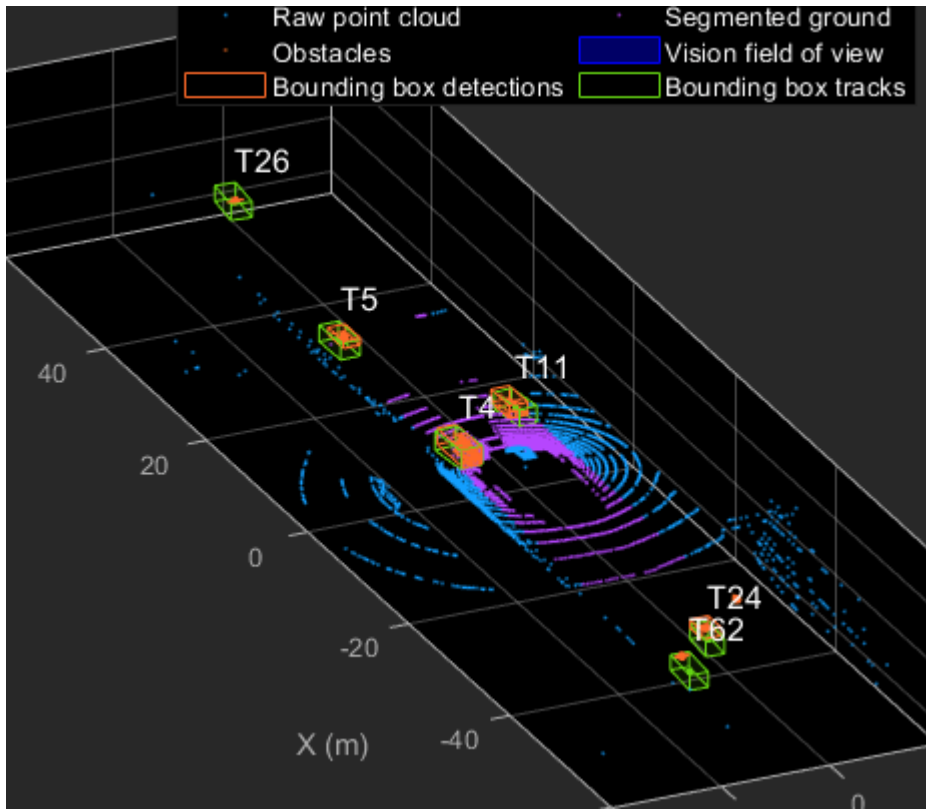
The animation above shows the simulation between time = 3 seconds and time = 16 seconds. Notice that tracks such as T10 and T6 maintain their IDs and trajectory during the time span. However, track T9 is lost because the tracked vehicle was missed (not detected) for a long time by the sensor. Also, notice that the tracked objects are able to maintain their shape and kinematic center by positioning the detections onto the visible portions of the vehicles. For example, as Track T7 moves forward, bounding box detections start to fall on its visible rear portion and the track maintains the actual size of the vehicle. This illustrates the offset and shrinkage effect modeled in the measurement functions.

Capturing Maneuvers



The animation shows that using an IMM filter helps the tracker to maintain tracks on maneuvering vehicles. Notice that the vehicle tracked by T4 changes lanes behind the ego vehicle. The tracker is able to maintain a track on the vehicle during this maneuvering event. Also notice in the display that its probability of following the constant turn model, denoted by ct , increases during the lane change maneuver.

Joint Probabilistic Data Association



This animation shows that using a joint probabilistic data association tracker helps in maintaining tracks during ambiguous situations. Here, vehicles tracked by T24 and T62, have a low probability of detection due to their large distance from the sensor. Notice that the tracker is able to maintain tracks during events when one of the vehicles is not detected. During the event, the tracks first coalesce, which is a known phenomenon in JPDA, and then separate as soon as the vehicle was detected again.

Summary

This example showed how to use a JPDA tracker with an IMM filter to track objects using a lidar sensor. You learned how a raw point cloud can be preprocessed to generate detections for conventional trackers, which assume one detection per object per sensor scan. You also learned how to define a cuboid model to describe the kinematics, dimensions, and measurements of extended objects being tracked by the JPDA tracker. In addition, you generated C code from the algorithm and verified its execution results with the MATLAB simulation.

Supporting Files

helperLidarModel

This function defines the lidar model to simulate shrinkage of the bounding box measurement and center-point offset. This function is used in the `helperCvmeasCuboid` and `helperCtmeasCuboid` functions to obtain bounding box measurement from the state.

```
function meas = helperLidarModel(pos,dim,yaw)
% This function returns the expected bounding box measurement given an
% object's position, dimension, and yaw angle.
```

```
% Copyright 2019 The MathWorks, Inc.

% Get x,y and z.
x = pos(1,:);
y = pos(2,:);
z = pos(3,:) - 2; % lidar mounted at height = 2 meters.

% Get spherical measurement.
[az,~,r] = cart2sph(x,y,z);

% Shrink rate
s = 3/50; % 3 meters radial length at 50 meters.
sz = 2/50; % 2 meters height at 50 meters.

% Get length, width and height.
L = dim(1,:);
W = dim(2,:);
H = dim(3,:);

az = az - deg2rad(yaw);

% Shrink length along radial direction.
Lshrink = min(L,abs(s*r.*(cos(az))));
Ls = L - Lshrink;

% Shrink width along radial direction.
Wshrink = min(W,abs(s*r.*(sin(az))));
Ws = W - Wshrink;

% Shrink height.
Hshrink = min(H,sz*r);
Hs = H - Hshrink;

% Measurement is given by a min-max detector hence length and width must be
% projected along x and y.
Lmeas = Ls.*cosd(yaw) + Ws.*sind(yaw);
Wmeas = Ls.*sind(yaw) + Ws.*cosd(yaw);

% Similar shift is for x and y directions.
shiftX = Lshrink.*cosd(yaw) + Wshrink.*sind(yaw);
shiftY = Lshrink.*sind(yaw) + Wshrink.*cosd(yaw);
shiftZ = Hshrink;

% Modeling the affect of box origin offset
x = x - sign(x).*shiftX/2;
y = y - sign(y).*shiftY/2;
z = z + shiftZ/2 + 2;

% Measurement format
meas = [x;y;z;Lmeas;Wmeas;Hs];

end
```

helperInverseLidarModel

This function defines the inverse lidar model to initiate a tracking filter using a lidar bounding box measurement. This function is used in the `helperInitIMMFilter` function to obtain state estimates from a bounding box measurement.

```
function [pos,posCov,dim,dimCov,yaw,yawCov] = helperInverseLidarModel(meas,measCov)
% This function returns the position, dimension, yaw using a bounding
% box measurement.

% Copyright 2019 The MathWorks, Inc.

% Shrink rate.
s = 3/50;
sz = 2/50;

% x,y and z of measurement
x = meas(1,:);
y = meas(2,:);
z = meas(3,:);

[az,~,r] = cart2sph(x,y,z);

% Shift x and y position.
Lshrink = abs(s*r.*(cos(az)));
Wshrink = abs(s*r.*(sin(az)));
Hshrink = sz*r;

shiftX = Lshrink;
shiftY = Wshrink;
shiftZ = Hshrink;

x = x + sign(x).*shiftX/2;
y = y + sign(y).*shiftY/2;
z = z + sign(z).*shiftZ/2;

pos = [x;y;z];
posCov = measCov(1:3,1:3,:);

yaw = zeros(1,numel(x),'like',x);
yawCov = ones(1,1,numel(x),'like',x);

% Dimensions are initialized for a standard passenger car with low
% uncertainty.
dim = [4.7;1.8;1.4];
dimCov = 0.01*eye(3);
end
```

HelperBoundingBoxDetector

This is the supporting class `HelperBoundingBoxDetector` to accept a point cloud input and return a list of `objectDetection`

```
classdef HelperBoundingBoxDetector < matlab.System
    % HelperBoundingBoxDetector A helper class to segment the point cloud
    % into bounding box detections.
    % The step call to the object does the following things:
```

```
%
% 1. Removes point cloud outside the limits.
% 2. From the survived point cloud, segments out ground
% 3. From the obstacle point cloud, forms clusters and puts bounding
%    box on each cluster.

% Cropping properties
properties
    % XLimits XLimits for the scene
    XLimits = [-70 70];
    % YLimits YLimits for the scene
    YLimits = [-6 6];
    % ZLimits ZLimits for the scene
    ZLimits = [-2 10];
end

% Ground Segmentation Properties
properties
    % GroundMaxDistance Maximum distance of point to the ground plane
    GroundMaxDistance = 0.3;
    % GroundReferenceVector Reference vector of ground plane
    GroundReferenceVector = [0 0 1];
    % GroundMaxAngularDistance Maximum angular distance of point to reference vector
    GroundMaxAngularDistance = 5;
end

% Bounding box Segmentation properties
properties
    % SegmentationMinDistance Distance threshold for segmentation
    SegmentationMinDistance = 1.6;
    % MinDetectionsPerCluster Minimum number of detections per cluster
    MinDetectionsPerCluster = 2;
    % MaxZDistanceCluster Maximum Z-coordinate of cluster
    MaxZDistanceCluster = 3;
    % MinZDistanceCluster Minimum Z-coordinate of cluster
    MinZDistanceCluster = -3;
end

% Ego vehicle radius to remove ego vehicle point cloud.
properties
    % EgoVehicleRadius Radius of ego vehicle
    EgoVehicleRadius = 3;
end

properties
    % MeasurementNoise Measurement noise for the bounding box detection
    MeasurementNoise = blkdiag(eye(3),eye(3));
end

properties (Nontunable)
    MeasurementParameters = struct.empty(0,1);
end

methods
    function obj = HelperBoundingBoxDetector(varargin)
        setProperties(obj,nargin,varargin{:})
    end
end
```

```

methods (Access = protected)
    function [bboxDets,obstacleIndices,groundIndices,croppedIndices] = stepImpl(obj,currentPC
        % Crop point cloud
        [pcSurvived,survivedIndices,croppedIndices] = cropPointCloud(currentPointCloud,obj.X
        % Remove ground plane
        [pcObstacles,obstacleIndices,groundIndices] = removeGroundPlane(pcSurvived,obj.Ground
        % Form clusters and get bounding boxes
        detBBoxes = getBoundingBoxes(pcObstacles,obj.SegmentationMinDistance,obj.MinDetection
        % Assemble detections
        if isempty(obj.MeasurementParameters)
            measParams = {};
        else
            measParams = obj.MeasurementParameters;
        end
        bboxDets = assembleDetections(detBBoxes,obj.MeasurementNoise,measParams,time);
    end
end
end

function detections = assembleDetections(bboxes,measNoise,measParams,time)
% This method assembles the detections in objectDetection format.
numBoxes = size(bboxes,2);
detections = cell(numBoxes,1);
for i = 1:numBoxes
    detections{i} = objectDetection(time,cast(bboxes(:,i),'double'),...
        'MeasurementNoise',double(measNoise),'ObjectAttributes',struct,...
        'MeasurementParameters',measParams);
end
end

function bboxes = getBoundingBoxes(ptCloud,minDistance,minDetsPerCluster,maxZDistance,minZDistance)
% This method fits bounding boxes on each cluster with some basic
% rules.
% Cluster must have atleast minDetsPerCluster points.
% Its mean z must be between maxZDistance and minZDistance.
% length, width and height are calculated using min and max from each
% dimension.
[labels,numClusters] = pcsegdist(ptCloud,minDistance);
pointData = ptCloud.Location;
bboxes = nan(6,numClusters,'like',pointData);
isValidCluster = false(1,numClusters);
for i = 1:numClusters
    thisPointData = pointData(labels == i,:);
    meanPoint = mean(thisPointData,1);
    if size(thisPointData,1) > minDetsPerCluster && ...
        meanPoint(3) < maxZDistance && meanPoint(3) > minZDistance
        xMin = min(thisPointData(:,1));
        xMax = max(thisPointData(:,1));
        yMin = min(thisPointData(:,2));
        yMax = max(thisPointData(:,2));
        zMin = min(thisPointData(:,3));
        zMax = max(thisPointData(:,3));
        l = (xMax - xMin);
        w = (yMax - yMin);
        h = (zMax - zMin);
        x = (xMin + xMax)/2;
        y = (yMin + yMax)/2;
    end
end

```

```

        z = (zMin + zMax)/2;
        bboxes(:,i) = [x y z l w h]';
        isValidCluster(i) = l < 20; % max length of 20 meters
    end
end
bboxes = bboxes(:,isValidCluster);
end

function [ptCloudOut,obstacleIndices,groundIndices] = removeGroundPlane(ptCloudIn,maxGroundDist,
% This method removes the ground plane from point cloud using
% pcfplane.
[-,groundIndices,outliers] = pcfplane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDist);
ptCloudOut = select(ptCloudIn,outliers);
obstacleIndices = currentIndices(outliers);
groundIndices = currentIndices(groundIndices);
end

function [ptCloudOut,indices,croppedIndices] = cropPointCloud(ptCloudIn,xLim,yLim,zLim,egoVehicleLocation)
% This method selects the point cloud within limits and removes the
% ego vehicle point cloud using findNeighborsInRadius
locations = ptCloudIn.Location;
locations = reshape(locations,[],3);
insideX = locations(:,1) < xLim(2) & locations(:,1) > xLim(1);
insideY = locations(:,2) < yLim(2) & locations(:,2) > yLim(1);
insideZ = locations(:,3) < zLim(2) & locations(:,3) > zLim(1);
inside = insideX & insideY & insideZ;

% Remove ego vehicle
nearIndices = findNeighborsInRadius(ptCloudIn,[0 0 0],egoVehicleRadius);
nonEgoIndices = true(ptCloudIn.Count,1);
nonEgoIndices(nearIndices) = false;
validIndices = inside & nonEgoIndices;
indices = find(validIndices);
croppedIndices = find(~validIndices);
ptCloudOut = select(ptCloudIn,indices);
end

```

mexLidarTracker

This function implements the point cloud preprocessing display and the tracking algorithm using a functional interface for code generation.

```

function [detections,obstacleIndices,groundIndices,croppedIndices,...
confirmedTracks, modelProbs] = mexLidarTracker(ptCloudLocations,time)

persistent detectorModel tracker detectableTracksInput currentNumTracks

if isempty(detectorModel) || isempty(tracker) || isempty(detectableTracksInput) || isempty(currentNumTracks)
% Use the same starting seed as MATLAB to reproduce results in SIL
% simulation.
rng(2018);
end

```

```

% A bounding box detector model.
detectorModel = HelperBoundingBoxDetector(...
    'XLimits', [-50 75],...           % min-max
    'YLimits', [-5 5],...           % min-max
    'ZLimits', [-2 5],...           % min-max
    'SegmentationMinDistance', 1.6,... % minimum Euclidian distance
    'MinDetectionsPerCluster', 1,... % minimum points per cluster
    'MeasurementNoise', eye(6),... % measurement noise in detection report.
    'GroundMaxDistance', 0.3);      % maximum distance of ground points from

assignmentGate = [50 100]; % Assignment threshold;
confThreshold = [7 10]; % Confirmation threshold for history logic
delThreshold = [8 10]; % Deletion threshold for history logic
Kc = 1e-5; % False-alarm rate per unit volume

filterInitFcn = @helperInitIMMFilter;

tracker = trackerJPDA('FilterInitializationFcn', filterInitFcn, ...
    'TrackLogic', 'History', ...
    'AssignmentThreshold', assignmentGate, ...
    'ClutterDensity', Kc, ...
    'ConfirmationThreshold', confThreshold, ...
    'DeletionThreshold', delThreshold, ...
    'HasDetectableTrackIDsInput', true, ...
    'InitializationThreshold', 0, ...
    'MaxNumTracks', 30);

detectableTracksInput = zeros(tracker.MaxNumTracks, 2);

currentNumTracks = 0;
end

ptCloud = pointCloud(ptCloudLocations);

% Detector model
[detections, obstacleIndices, groundIndices, croppedIndices] = detectorModel(ptCloud, time);

% Call tracker
[confirmedTracks, ~, allTracks] = tracker(detections, time, detectableTracksInput(1:currentNumTracks));
% Update the detectability input
currentNumTracks = numel(allTracks);
detectableTracksInput(1:currentNumTracks, :) = helperCalcDetectability(allTracks, [1 3 6]);

% Get model probabilities
modelProbs = zeros(2, numel(confirmedTracks));
if isLocked(tracker)
    for k = 1:numel(confirmedTracks)
        c1 = getTrackFilterProperties(tracker, confirmedTracks(k).TrackID, 'ModelProbabilities');
        probs = c1{1};
        modelProbs(1, k) = probs(1);
        modelProbs(2, k) = probs(2);
    end
end
end
end

```

helperCalcDetectability

The function calculate the probability of detection for each track. This function is used to generate the "DetectableTracksIDs" input for the trackerJPDA.

```
function detectableTracksInput = helperCalcDetectability(tracks,posIndices)
% This is a helper function to calculate the detection probability of
% tracks for the lidar tracking example. It may be removed in a future
% release.

% Copyright 2019 The MathWorks, Inc.

% The bounding box detector has low probability of segmenting point clouds
% into bounding boxes are distances greater than 40 meters. This function
% models this effect using a state-dependent probability of detection for
% each tracker. After a maximum range, the Pd is set to a high value to
% enable deletion of track at a faster rate.
if isempty(tracks)
    detectableTracksInput = zeros(0,2);
    return;
end
rMax = 75;
rAmbig = 40;
stateSize = numel(tracks(1).State);
posSelector = zeros(3,stateSize);
posSelector(1,posIndices(1)) = 1;
posSelector(2,posIndices(2)) = 1;
posSelector(3,posIndices(3)) = 1;
pos = getTrackPositions(tracks,posSelector);
if coder.target('MATLAB')
    trackIDs = [tracks.TrackID];
else
    trackIDs = zeros(1,numel(tracks),'uint32');
    for i = 1:numel(tracks)
        trackIDs(i) = tracks(i).TrackID;
    end
end
end
[~,~,r] = cart2sph(pos(:,1),pos(:,2),pos(:,3));
probDetection = 0.9*ones(numel(tracks),1);
probDetection(r > rAmbig) = 0.4;
probDetection(r > rMax) = 0.99;
detectableTracksInput = [double(trackIDs(:)) probDetection(:)];
end
```

loadLidarAndImageData

Stitches Lidar and Camera data for processing using initial and final time specified.

```
function [lidarData,imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime)
initFrame = max(1,floor(initTime*10));
lastFrame = min(350,ceil(finalTime*10));
load (fullfile(datasetFolder,'imageData_35seconds.mat'),'allImageData');
imageData = allImageData(initFrame:lastFrame);

numFrames = lastFrame - initFrame + 1;
lidarData = cell(numFrames,1);

% Each file contains 70 frames.
```



```
initFileIndex = floor(initFrame/70) + 1;
lastFileIndex = ceil(lastFrame/70);

frameIndices = [1:70:numFrames numFrames + 1];

counter = 1;
for i = initFileIndex:lastFileIndex
    startFrame = frameIndices(counter);
    endFrame = frameIndices(counter + 1) - 1;
    load(fullfile(datasetFolder,['lidarData_',num2str(i)]),'currentLidarData');
    lidarData(startFrame:endFrame) = currentLidarData(1:(endFrame + 1 - startFrame));
    counter = counter + 1;
end
end
```

References

[1] Arya Senna Abdul Rachman, Arya. "3D-LIDAR Multi Object Tracking for Autonomous Driving: Multi-target Detection and Tracking under Urban Road Uncertainties." (2017).

Semantic Segmentation Using Dilated Convolutions

Train a semantic segmentation network using dilated convolutions.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” on page 14-43.

Semantic segmentation networks like DeepLab [1] make extensive use of dilated convolutions (also known as atrous convolutions) because they can increase the receptive field of the layer (the area of the input which the layers can see) without increasing the number of parameters or computations.

Load Training Data

The example uses a simple dataset of 32-by-32 triangle images for illustration purposes. The dataset includes accompanying pixel label ground truth data. Load the training data using an `imageDatastore` and a `pixelLabelDatastore`.

```
dataFolder = fullfile(toolboxdir('vision'),'visiondata','triangleImages');
imageFolderTrain = fullfile(dataFolder,'trainingImages');
labelFolderTrain = fullfile(dataFolder,'trainingLabels');
```

Create an `imageDatastore` for the images.

```
imdsTrain = imageDatastore(imageFolderTrain);
```

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

```
classNames = ["triangle" "background"];
labels = [255 0];
pxdsTrain = pixelLabelDatastore(labelFolderTrain,classNames,labels)
```

```
pxdsTrain =
  PixelLabelDatastore with properties:
        Files: {200x1 cell}
  ClassNames: {2x1 cell}
    ReadSize: 1
    ReadFcn: @readDatastoreImage
AlternateFileSystemRoots: {}
```

Create Semantic Segmentation Network

This example uses a simple semantic segmentation network based on dilated convolutions.

Create a data source for training data and get the pixel counts for each label.

```
pximdsTrain = pixelLabelImageDatastore(imdsTrain,pxdsTrain);
tbl = countEachLabel(pximdsTrain)
```

```
tbl=2x3 table
      Name          PixelCount  ImagePixelCount
      _____  _____  _____
    {'triangle' }      10326      2.048e+05
```

```
{'background'}    1.9447e+05    2.048e+05
```

The majority of pixel labels are for background. This class imbalance biases the learning process in favor of the dominant class. To fix this, use class weighting to balance the classes. You can use several methods to compute class weights. One common method is inverse frequency weighting where the class weights are the inverse of the class frequencies. This method increases the weight given to under represented classes. Calculate the class weights using inverse frequency weighting.

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

Create a network for pixel classification by using an image input layer with an input size corresponding to the size of the input images. Next, specify three blocks of convolution, batch normalization, and ReLU layers. For each convolutional layer, specify 32 3-by-3 filters with increasing dilation factors and pad the inputs so they are the same size as the outputs by setting the 'Padding' option to 'same'. To classify the pixels, include a convolutional layer with K 1-by-1 convolutions, where K is the number of classes, followed by a softmax layer and a pixelClassificationLayer with the inverse class weights.

```
inputSize = [32 32 1];
filterSize = 3;
numFilters = 32;
numClasses = numel(classNames);

layers = [
    imageInputLayer(inputSize)

    convolution2dLayer(filterSize,numFilters,'DilationFactor',1,'Padding','same')
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(filterSize,numFilters,'DilationFactor',2,'Padding','same')
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(filterSize,numFilters,'DilationFactor',4,'Padding','same')
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(1,numClasses)
    softmaxLayer
    pixelClassificationLayer('Classes',classNames,'ClassWeights',classWeights)];
```

Train Network

Specify the training options.

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

Train the network using trainNetwork.

```
net = trainNetwork(pximdsTrain,layers,options);
```

Training on single CPU.
Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:03	91.62%	1.6825	0.0010
17	50	00:00:58	88.56%	0.2393	0.0010
34	100	00:01:58	92.08%	0.1672	0.0010
50	150	00:02:57	93.17%	0.1472	0.0010
67	200	00:03:55	94.15%	0.1313	0.0010
84	250	00:04:51	94.47%	0.1167	0.0010
100	300	00:05:47	95.04%	0.1100	0.0010

Test Network

Load the test data. Create an `imageDatastore` for the images. Create a `pixelLabelDatastore` for the ground truth pixel labels.

```
imageFolderTest = fullfile(dataFolder, 'testImages');
imdsTest = imageDatastore(imageFolderTest);
labelFolderTest = fullfile(dataFolder, 'testLabels');
pxdsTest = pixelLabelDatastore(labelFolderTest, classNames, labels);
```

Make predictions using the test data and trained network.

```
pxdsPred = semanticseg(imdsTest, net, 'MiniBatchSize', 32, 'WriteLocation', tempdir);
```

Running semantic segmentation network

```
-----
* Processed 100 images.
```

Evaluate the prediction accuracy using `evaluateSemanticSegmentation`.

```
metrics = evaluateSemanticSegmentation(pxdsPred, pxdsTest);
```

Evaluating semantic segmentation results

```
-----
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 100 images.
* Finalizing... Done.
* Data set metrics:
```

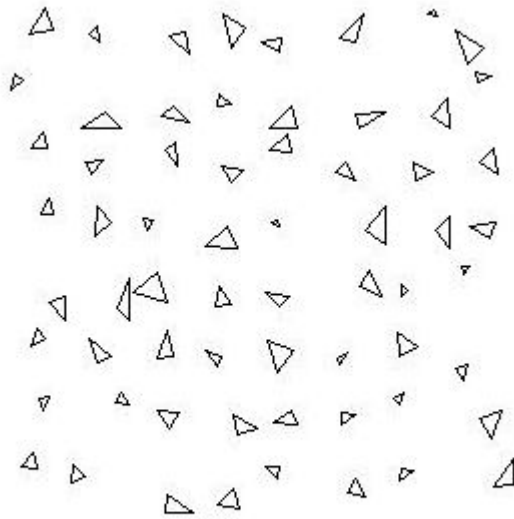
GlobalAccuracy	MeanAccuracy	MeanIoU	WeightedIoU	MeanBFScore
0.95237	0.97352	0.72081	0.92889	0.46416

For more information on evaluating semantic segmentation networks, see `evaluateSemanticSegmentation`.

Segment New Image

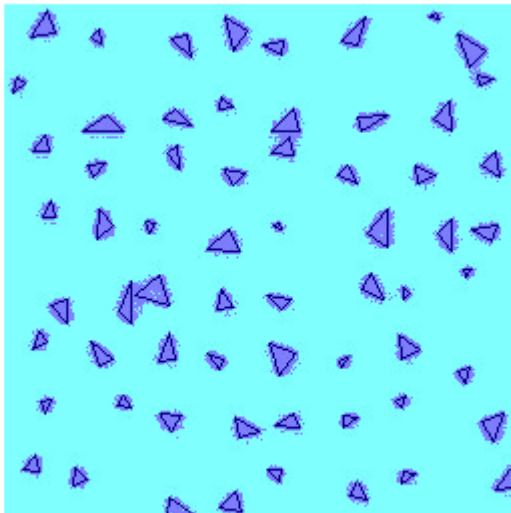
Read and display the test image `triangleTest.jpg`.

```
imgTest = imread('triangleTest.jpg');
figure
imshow(imgTest)
```



Segment the test image using `semanticseg` and display the results using `labeloverlay`.

```
C = semanticseg(imgTest,net);  
B = labeloverlay(imgTest,C);  
figure  
imshow(B)
```



Define Custom Pixel Classification Layer with Tversky Loss

This example shows how to define and create a custom pixel classification layer that uses Tversky loss.

This layer can be used to train semantic segmentation networks. To learn more about creating custom deep learning layers, see “Define Custom Deep Learning Layers” (Deep Learning Toolbox).

Tversky Loss

The Tversky loss is based on the Tversky index for measuring overlap between two segmented images [1 on page 8-0]. The Tversky index TI_c between one image Y and the corresponding ground truth T is given by

$$TI_c = \frac{\sum_{m=1}^M Y_{cm} T_{cm}}{\sum_{m=1}^M Y_{cm} T_{cm} + \alpha \sum_{m=1}^M Y_{cm} T_{\bar{c}m} + \beta \sum_{m=1}^M Y_{\bar{c}m} T_{cm}}$$

- c corresponds to the class and \bar{c} corresponds to not being in class c .
- M is the number of elements along the first two dimensions of Y .
- α and β are weighting factors that control the contribution that false positives and false negatives for each class make to the loss.

The loss L over the number of classes C is given by

$$L = \sum_{c=1}^C 1 - TI_c$$

Classification Layer Template

Copy the classification layer template into a new file in MATLAB®. This template outlines the structure of a classification layer and includes the functions that define the layer behavior. The rest of the example shows how to complete the `tverskyPixelClassificationLayer`.

```
classdef tverskyPixelClassificationLayer < nnet.layer.ClassificationLayer

    properties
        % Optional properties
    end

    methods

        function loss = forwardLoss(layer, Y, T)
            % Layer forward loss function goes here
        end

    end
end
```

Declare Layer Properties

By default, custom output layers have the following properties:

- **Name** - Layer name, specified as a character vector or a string scalar. To include this layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with this layer and Name is set to ' ', then the software automatically assigns a name at training time.
- **Description** - One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a Layer array. If you do not specify a layer description, then the software displays the layer class name.
- **Type** - Type of the layer, specified as a character vector or a string scalar. The value of Type appears when the layer is displayed in a Layer array. If you do not specify a layer type, then the software displays 'Classification layer' or 'Regression layer'.

Custom classification layers also have the following property:

- **Classes** - Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or 'auto'. If Classes is 'auto', then the software automatically sets the classes at training time. If you specify a string array or cell array of character vectors `str`, then the software sets the classes of the output layer to `categorical(str, str)`. The default value is 'auto'.

If the layer has no other properties, then you can omit the properties section.

The Tversky loss requires a small constant value to prevent division by zero. Specify the property, `Epsilon`, to hold this value. It also requires two variable properties `Alpha` and `Beta` that control the weighting of false positives and false negatives, respectively.

```
classdef tverskyPixelClassificationLayer < nnet.layer.ClassificationLayer
    properties(Constant)
        % Small constant to prevent division by zero.
        Epsilon = 1e-8;
    end

    properties
        % Default weighting coefficients for false positives and false negatives
        Alpha = 0.5;
        Beta = 0.5;
    end

    ...
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

Specify an optional input argument name to assign to the Name property at creation.

```
function layer = tverskyPixelClassificationLayer(name, alpha, beta)
    % layer = tverskyPixelClassificationLayer(name) creates a Tversky
    % pixel classification layer with the specified name.

    % Set layer name
    layer.Name = name;

    % Set layer properties
    layer.Alpha = alpha;
```

```

    layer.Beta = beta;

    % Set layer description
    layer.Description = 'Tversky loss';
end

```

Create Forward Loss Function

Create a function named `forwardLoss` that returns the weighted cross entropy loss between the predictions made by the network and the training targets. The syntax for `forwardLoss` is `loss = forwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` represents the training targets.

For semantic segmentation problems, the dimensions of `T` match the dimension of `Y`, where `Y` is a 4-D array of size H-by-W-by-K-by-N, where `K` is the number of classes, and `N` is the mini-batch size.

The size of `Y` depends on the output of the previous layer. To ensure that `Y` is the same size as `T`, you must include a layer that outputs the correct size before the output layer. For example, to ensure that `Y` is a 4-D array of prediction scores for `K` classes, you can include a fully connected layer of size `K` or a convolutional layer with `K` filters followed by a softmax layer before the output layer.

```

function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the Tversky loss between
    % the predictions Y and the training targets T.

    Pcnot = 1-Y;
    Gcnot = 1-T;
    TP = sum(sum(Y.*T,1),2);
    FP = sum(sum(Y.*Gcnot,1),2);
    FN = sum(sum(Pcnot.*T,1),2);

    numer = TP + layer.Epsilon;
    denom = TP + layer.Alpha*FP + layer.Beta*FN + layer.Epsilon;

    % Compute Tversky index
    lossTic = 1 - numer./denom;
    lossTI = sum(lossTic,3);

    % Return average Tversky index loss
    N = size(Y,4);
    loss = sum(lossTI)/N;
end

```

Backward Loss Function

As the `forwardLoss` function fully supports automatic differentiation, there is no need to create a function for the backward loss.

For a list of functions that support automatic differentiation, see “List of Functions with `dlarray` Support” (Deep Learning Toolbox).

Completed Layer

The completed layer is provided in `tverskyPixelClassificationLayer.m`.

```

classdef tverskyPixelClassificationLayer < nnet.layer.ClassificationLayer
    % This layer implements the Tversky loss function for training

```



```

% semantic segmentation networks.

% References
% Salehi, Seyed Sadegh Mohseni, Deniz Erdogmus, and Ali Gholipour.
% "Tversky loss function for image segmentation using 3D fully
% convolutional deep networks." International Workshop on Machine
% Learning in Medical Imaging. Springer, Cham, 2017.
% -----

properties(Constant)
    % Small constant to prevent division by zero.
    Epsilon = 1e-8;
end

properties
    % Default weighting coefficients for False Positives and False
    % Negatives
    Alpha = 0.5;
    Beta = 0.5;
end

methods

function layer = tverskyPixelClassificationLayer(name, alpha, beta)
    % layer = tverskyPixelClassificationLayer(name, alpha, beta) creates a Tversky
    % pixel classification layer with the specified name and properties alpha and beta.

    % Set layer name.
    layer.Name = name;

    layer.Alpha = alpha;
    layer.Beta = beta;

    % Set layer description.
    layer.Description = 'Tversky loss';
end

function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the Tversky loss between
    % the predictions Y and the training targets T.

    Pcnot = 1-Y;
    Gcnot = 1-T;
    TP = sum(sum(Y.*T,1),2);
    FP = sum(sum(Y.*Gcnot,1),2);
    FN = sum(sum(Pcnot.*T,1),2);

    numer = TP + layer.Epsilon;
    denom = TP + layer.Alpha*FP + layer.Beta*FN + layer.Epsilon;

    % Compute tversky index
    lossTic = 1 - numer./denom;
    lossTI = sum(lossTic,3);

    % Return average tversky index loss.

```

```

        N = size(Y,4);
        loss = sum(lossTI)/N;
    end
end
end

```

GPU Compatibility

The MATLAB functions used in `forwardLoss` in `tverskyPixelClassificationLayer` all support `gpuArray` inputs, so the layer is GPU compatible.

Check Output Layer Validity

Create an instance of the layer.

```
layer = tverskyPixelClassificationLayer('tversky',0.7,0.3);
```

Check the validity of the layer by using `checkLayer` (Deep Learning Toolbox). Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects a H-by-W-by-K-by-N array inputs, where K is the number of classes, and N is the number of observations in the mini-batch.

```
numClasses = 2;
validInputSize = [4 4 numClasses];
checkLayer(layer,validInputSize, 'ObservationDimension',4)
```

```
Skipping GPU tests. No compatible GPU device found.
```

```
Skipping code generation compatibility tests. To check validity of the layer for code generation
```

```
Running nnet.checklayer.TestOutputLayerWithoutBackward
```

```
.....
```

```
Done nnet.checklayer.TestOutputLayerWithoutBackward
```

```
-----
Test Summary:
```

```
  8 Passed, 0 Failed, 0 Incomplete, 2 Skipped.
  Time elapsed: 3.8848 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

Use Custom Layer in Semantic Segmentation Network

Create a semantic segmentation network that uses the `tverskyPixelClassificationLayer`.

```
layers = [
    imageInputLayer([32 32 1])
    convolution2dLayer(3,64,'Padding',1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,64,'Padding',1)
    reluLayer
    transposedConv2dLayer(4,64,'Stride',2,'Cropping',1)
    convolution2dLayer(1,2)
    softmaxLayer
    tverskyPixelClassificationLayer('tversky',0.3,0.7)]
```

```

layers =
  1x1 Layer array with layers:

   1  ''      Image Input          32x32x1 images with 'zerocenter' normalization
   2  ''      Convolution          64 3x3 convolutions with stride [1 1] and padding
   3  ''      Batch Normalization  Batch normalization
   4  ''      ReLU                 ReLU
   5  ''      Max Pooling          2x2 max pooling with stride [2 2] and padding [0
   6  ''      Convolution          64 3x3 convolutions with stride [1 1] and padding
   7  ''      ReLU                 ReLU
   8  ''      Transposed Convolution 64 4x4 transposed convolutions with stride [2 2] a
   9  ''      Convolution          2 1x1 convolutions with stride [1 1] and padding
  10  ''      Softmax              softmax
  11  'tversky' Classification Output Tversky loss

```

Load training data for semantic segmentation using `imageDatastore` and `pixelLabelDatastore`.

```

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

```

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

```

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

```

Associate the image and pixel label data by using `pixelLabelImageDatastore`.

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

Set the training options and train the network.

```

options = trainingOptions('adam', ...
    'InitialLearnRate',1e-3, ...
    'MaxEpochs',100, ...
    'LearnRateDropFactor',5e-1, ...
    'LearnRateDropPeriod',20, ...
    'LearnRateSchedule','piecewise', ...
    'MiniBatchSize',50);

```

```
net = trainNetwork(ds,layers,options);
```

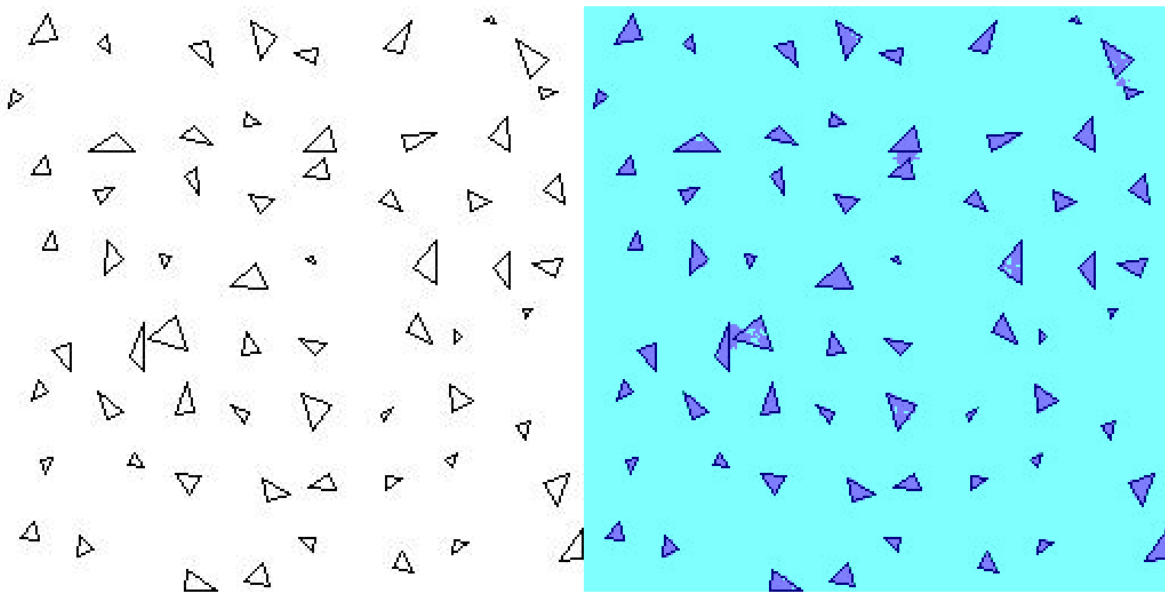
Training on single CPU.

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:02	50.32%	1.2933	0.0010
13	50	00:00:58	98.83%	0.0986	0.0010
25	100	00:01:57	99.32%	0.0545	0.0005
38	150	00:02:55	99.38%	0.0471	0.0005
50	200	00:03:51	99.48%	0.0401	0.0003
63	250	00:04:49	99.48%	0.0378	0.0001
75	300	00:05:45	99.54%	0.0349	0.0001
88	350	00:06:39	99.52%	0.0352	6.2500e-05
100	400	00:07:33	99.56%	0.0330	6.2500e-05

Evaluate the trained network by segmenting a test image and displaying the segmentation result.

```
I = imread('triangleTest.jpg');  
[C,scores] = semanticseg(I,net);  
  
B = labeloverlay(I,C);  
montage({I,B})
```



References

[1] Salehi, Seyed Sadegh Mohseni, Deniz Erdogmus, and Ali Gholipour. "Tversky loss function for image segmentation using 3D fully convolutional deep networks." *International Workshop on Machine Learning in Medical Imaging*. Springer, Cham, 2017.

Track a Face in Scene

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

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

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

```
objectFrame = readFrame(videoReader);  
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.

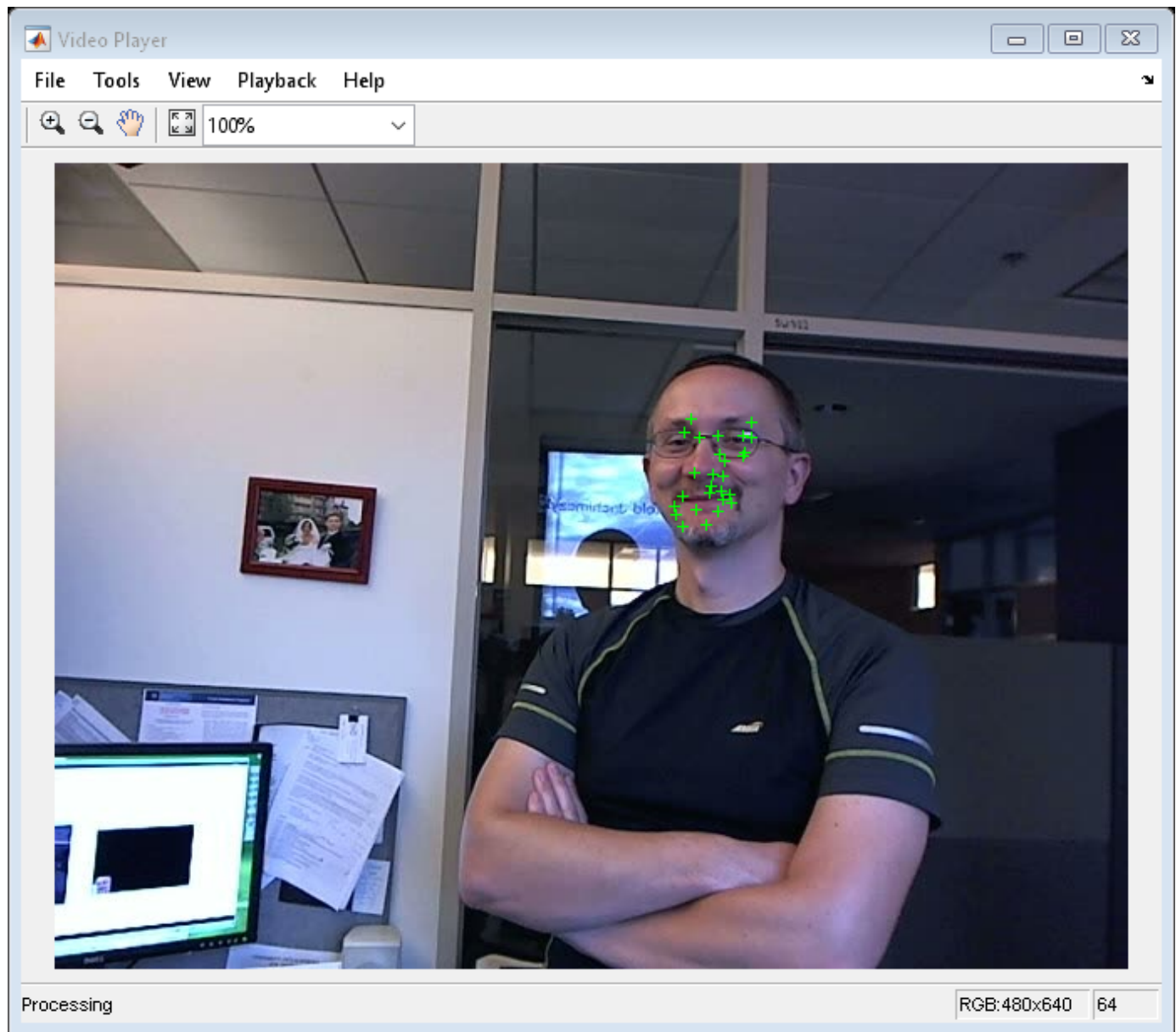
```
tracker = vision.PointTracker('MaxBidirectionalError',1);
```

Initialize the tracker.

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

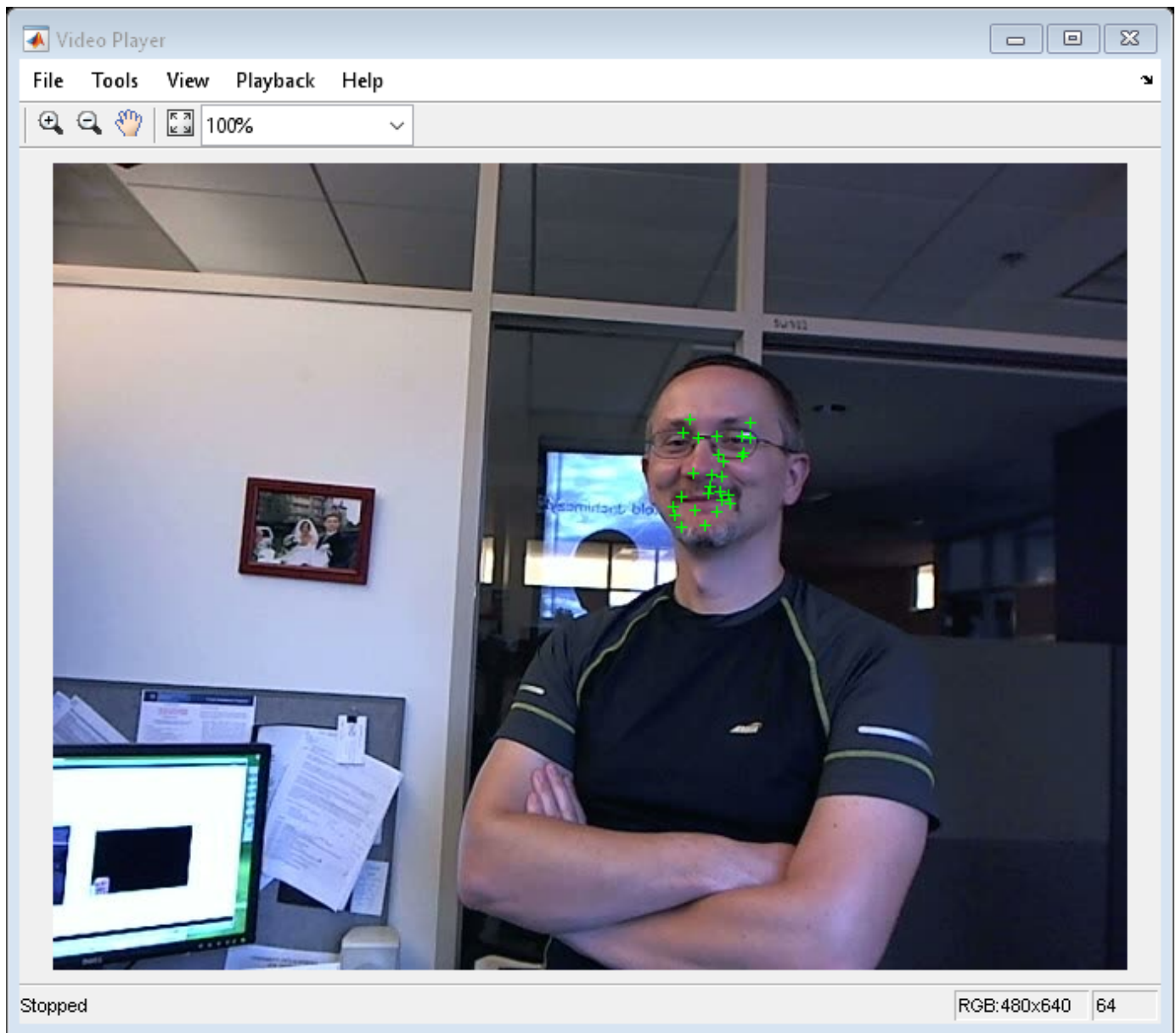
Read, track, display points, and results in each video frame.

```
while hasFrame(videoReader)  
    frame = readFrame(videoReader);  
    [points,validity] = tracker(frame);  
    out = insertMarker(frame,points(validity, :),'+');  
    videoPlayer(out);  
end
```



Release the video player.

```
release(videoPlayer);
```

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 = disparitySGM(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.

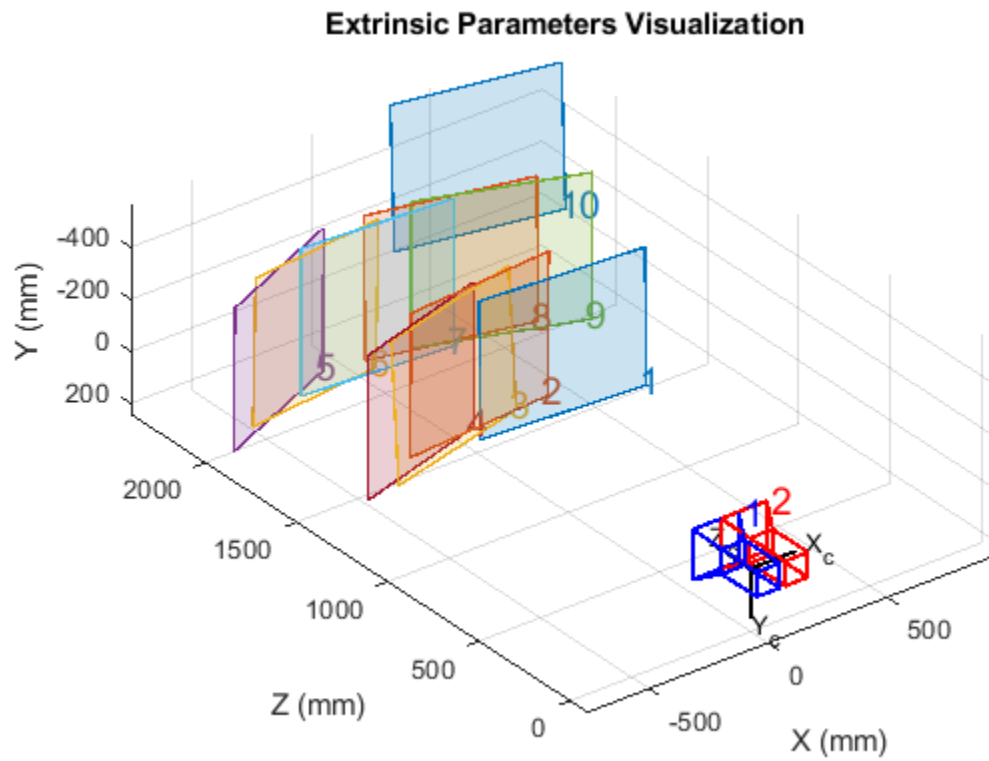
```
squareSize = 108;  
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

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);
```

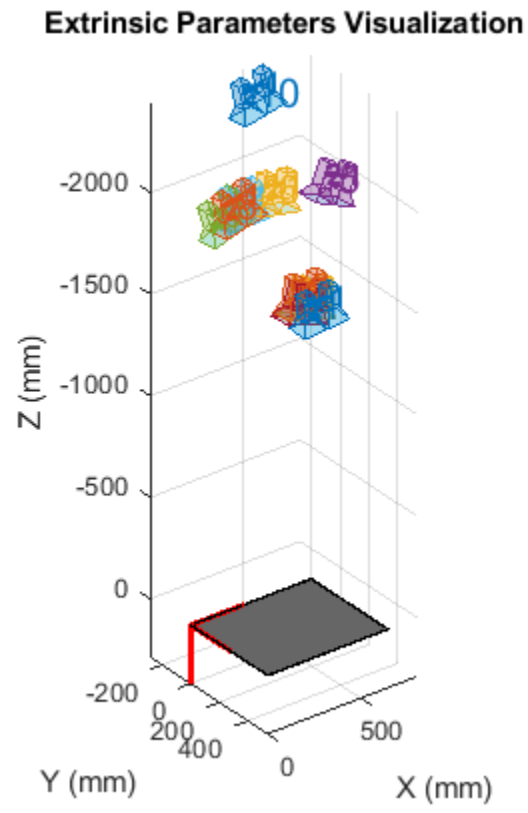
Visualize pattern locations.

```
figure;  
showExtrinsics(cameraParams);
```



Visualize camera locations.

```
figure;  
showExtrinsics(cameraParams, 'patternCentric');
```

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 Camera Calibrator 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',radialDistortion);
```

Remove distortion from the images.

```
I = imread(fullfile(matlabroot,'toolbox','vision','visiondata','calibration','mono','image01.jpg'));  
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)

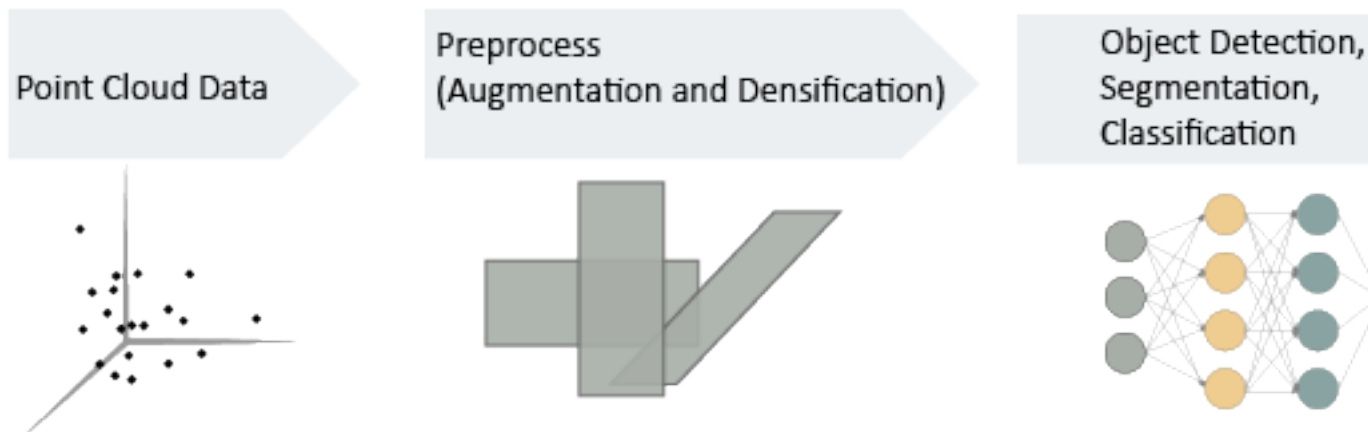


Point Cloud Processing

- “Getting Started with Point Clouds Using Deep Learning” on page 9-2
- “Point Cloud Registration and Mapping Overview” on page 9-4
- “The PLY Format” on page 9-10

Getting Started with Point Clouds Using Deep Learning

Deep learning can automatically process point clouds for a wide range of 3-D imaging applications. Point clouds typically come from 3-D scanners, such as a lidar or Kinect® devices. They have applications in robot navigation and perception, depth estimation, stereo vision, surveillance, scene classification, and in advanced driver assistance systems (ADAS).



In general, the first steps for using point cloud data in a deep learning workflow are:

- 1 Import point cloud data. Use a datastore to hold the large amount of data.
- 2 Optionally augment the data.
- 3 Encode the point cloud to an image-like format consistent with MATLAB®-based deep learning workflows.

You can apply the same deep learning approaches to classification, object detection, and semantic segmentation tasks using point cloud data as you would using regular gridded image data. However, you must first encode the unordered, irregularly gridded structure of point cloud and lidar data into a regular gridded form. For certain tasks, such as semantic segmentation, some postprocessing on the output of image-based networks is required in order to restore a point cloud structure.

Import Point Cloud Data

In order to work with point cloud data in deep learning workflows, first, read the raw data. Consider using a datastore for working with and representing collections of data that are too large to fit in memory at one time. Because deep learning often requires large amounts of data, datastores are an important part of the deep learning workflow in MATLAB. For more details about datastores, see “Datastores for Deep Learning” (Deep Learning Toolbox).

The “Import Point Cloud Data For Deep Learning” on page 5-7 example imports a large point cloud data set, and then configures and loads a datastore.

Augment Data

The accuracy and success of a deep learning model depends on large annotated datasets. Using augmentation to produce larger datasets helps reduce overfitting. Overfitting occurs when a

classification system mistakes noise in the data for a signal. By adding additional noise, augmentation helps the model balance the data points and minimize the errors. Augmentation can also add robustness to data transformations which may not be well represented in the original training data, (for example rotation, reflection, translations). And by reducing overfitting, augmentation can often lead to better results in the inference stage, which makes predictions based on what the deep learning neural network has been trained to detect.

The “Augment Point Cloud Data For Deep Learning” on page 5-2 example setups a basic randomized data augmentation pipeline that works with point cloud data.

Encode Point Cloud Data to Image-like Format

To use point clouds for training with MATLAB-based deep learning workflows, the data must be encoded into a dense, image-like format. Densification or voxelization is the process of transforming an irregular, ungridded form of point cloud data to a dense, image-like form.

The “Encode Point Cloud Data For Deep Learning” on page 5-11 example transforms point cloud data into a dense, gridded structure.

Train a Deep Learning Classification Network with Encoded Point Cloud Data

Once you have encoded point cloud data into a dense form, you can use the data for an image-based classification, object detection, or semantic segmentation task using standard deep learning approaches.

The “Train Classification Network to Classify Object in 3-D Point Cloud” on page 3-207 example preprocesses point cloud data into a voxelized encoding and then uses the image-like data with a simple 3-D convolutional neural network to perform object classification.

See Also

[pcregistercpd](#) | [pcregistericp](#) | [pcregisterndt](#)

Related Examples

- “3-D Point Cloud Registration and Stitching” on page 5-54
- “Build a Map from Lidar Data” (Automated Driving Toolbox)
- “Object Detection Using YOLO v2 Deep Learning” on page 3-170
- “Object Detection Using SSD Deep Learning” on page 3-23
- “Object Detection Using Faster R-CNN Deep Learning” on page 3-197

More About

- “Getting Started with Object Detection Using Deep Learning” on page 14-13
- “Getting Started with YOLO v2” on page 14-26
- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 14-30
- “Getting Started with SSD Multibox Detection” on page 14-9

Point Cloud Registration and Mapping Overview

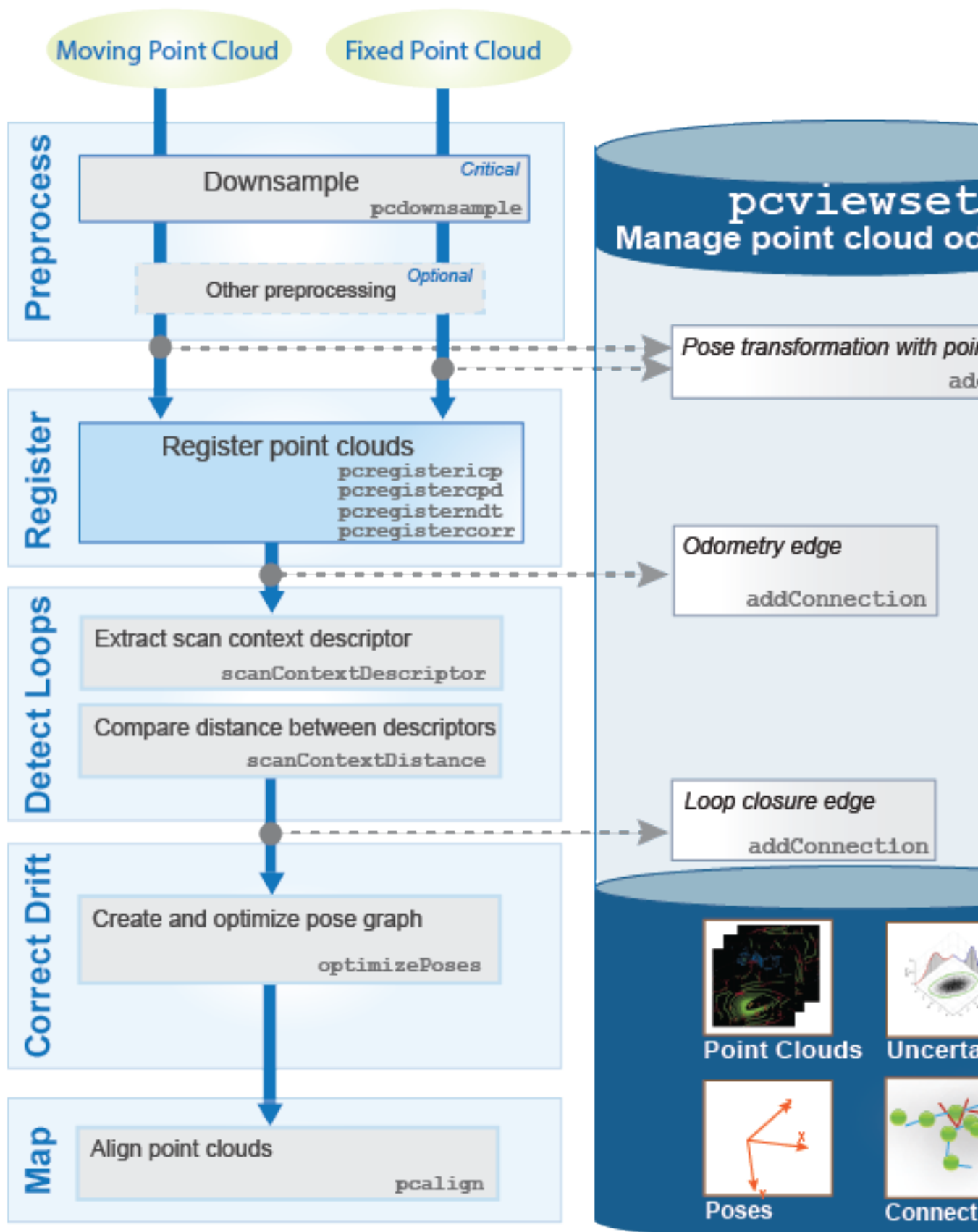
A point cloud is a set of points in 3-D space. Point clouds are typically obtained from 3-D scanners, such as a lidar or Kinect device. They have applications in robot navigation and perception, depth estimation, stereo vision, visual registration, and advanced driver assistance systems (ADAS).

Point cloud registration is the process of aligning two or more 3-D point clouds of the same scene into a common coordinate system. Mapping is the process of building a map of the environment around a robot or a sensor. Registration and mapping can be used to reconstruct a 3-D scene or build a map of a roadway for localization. While registration is commonly followed by mapping, there are other applications using registration, such as deformable motion tracking, which may not require mapping. Computer Vision Toolbox algorithms provide functions for performing point cloud registration and mapping. The workflow consists of preprocessing, registration, drift correction, and alignment of point clouds.

Registration and Mapping Workflow

Follow these steps to perform point cloud registration and mapping on a sequence of point clouds.

- 1** Preprocess Point Clouds — To prepare the point clouds for registration, downsample them and remove unwanted features and noise.
- 2** Register Point Clouds — Register each point cloud against the one preceding it. These registrations are used in Odometry, which is the process of accumulating a registration estimate over successive frames. Using odometry alone can lead to drift between the measured and ground truth poses.
- 3** Detect Loops — To minimize drift, you must identify the return of the sensor to a previously visited location, forming a loop in the trajectory of the sensor. This is referred to as loop closure detection.
- 4** Correct Drift — Use the detected loops to minimize drift through pose graph optimization, which consists of incrementally building a pose graph by adding nodes and edges, and then optimizing the pose graph once sufficient loops are found. The result of pose graph optimization is a set of optimized absolute poses.
- 5** Assemble Map — Assemble a map by aligning the registered point clouds using their optimized absolute poses.



Manage Point Cloud Registration and Mapping Data

Use these objects to manage data associated with the point cloud registration and mapping workflow:

- `pointCloud` object — The point cloud object stores a set of points located in 3-D space. It uses efficient indexing strategies to accomplish nearest neighbor searches, which are leveraged by point cloud preprocessing and registration functions.
- `rigid3d` object — The rigid 3-D object stores a 3-D rigid geometric transformation. In this workflow, it represents the relative and absolute poses.
- `pcviewset` object — The point cloud view set object manages the data associated with the odometry and mapping process. It organizes data as a set of views and pairwise connections between views. It also builds and updates a pose graph.
 - Each view consists of a point cloud and the associated absolute pose transformation. Each view has a unique identifier within the view set and forms a node of the pose graph.
 - Each connection stores information that links one view to another view. This includes the relative transformation between the connected views and the uncertainty involved in computing the measurement. Each connection forms an edge in the pose graph.

Preprocess Point Clouds

Preprocessing includes removing unwanted features and noise from the point clouds, as well as segmenting or downsampling them. Preprocessing can include these functions:

- `pcsegdist` or `segmentLidarData` — Segment the point cloud data into clusters, then use the `select` function to select the desired points.
- `pcfitplane` or `segmentGroundFromLidarData` — Segment the ground plane, then use the `select` function to select the desired points.
- `pcdenoise` — Remove unwanted noise from the point cloud.

Register Point Clouds

You can use the `pcregistericp`, `pcregistercpd`, `pcregisterndt`, or `pcregistercorr` function to register a moving point cloud to a fixed point cloud. The registration algorithms used by these functions are based on the iterative closest point (ICP) algorithm, the coherent point drift (CPD) algorithm, the normal-distributions transform (NDT) algorithm, and a phase correlation algorithm, respectively. For more information on these algorithms, see “References” on page 9-9.

When registering a point clouds, choose the type of transformation that represents how objects in the scene change between them.

Transformation	Description
Rigid	The rigid transformation preserves the shape and size of objects in the scene. Objects in the scene can undergo translations, rotations, or both. The same transformation applies to all points.
Affine	The affine transformation allows the objects to shear and change scale in addition to translations and rotations.

Transformation	Description
Nonrigid	The nonrigid transformation allows the shape of objects in the scene to change. Points undergo distinct transformations. A displacement field represents the transformation.

This table compares the point cloud registration function options, their transformation types, and their performance characteristics. Use this table to help you select the appropriate registration function for your use case.

Registration Method (function)	Transformation Type	Description	Performance Characteristics
<code>pcregisterndt</code>	Rigid	<ul style="list-style-type: none"> Local registration method that relies on an initial transform estimate Robust to outliers Better with point clouds of differing resolutions and densities 	Fast registration method, but generally slower than ICP
<code>pcregistericp</code>	Rigid	Local registration method that relies on an initial transform estimate	Fastest registration method
<code>pcregistercpd</code>	Rigid, affine, and nonrigid	Global method that does not rely on an initial transformation estimate	Slowest registration method
<code>pcregistercorr</code>	Rigid	Registration method that relies on an occupancy grid, assigning probability values to the grid based on the Z-coordinate values of points within each grid cell.	Best suited for ground vehicle navigation Increasing the size of the occupancy grids increases the computational requirements of the function.

Registering the current (moving) point cloud against the previous (fixed) point cloud returns a `rigid3d` transformation that represents the estimated relative pose of the moving point cloud in the frame of the fixed point cloud. Composing this relative pose transformation with all previously accumulated relative pose transformations gives an estimate of the absolute pose transformation.

Add the view formed by the moving point cloud and its absolute pose transformation. You can add the view to the `pcviewset` object using the `addView` function.

Add the odometry edge, an edge defined by the connection between successive views, formed by the relative pose transformation between the fixed and moving point clouds to the `pcviewset` object using the `addConnection` function.

Detect Loops

Using odometry alone leads to drift due to an accumulation of errors. These errors can result in severe inaccuracies over long distances. Using graph-based simultaneous localization and mapping (SLAM) corrects the drift. To do this, detect loop closures by finding a location visited in a previous point cloud using descriptor matching. Close the loop to correct the drift. Follow these steps for loop detection and closure:

- 1 Use the `scanContextDescriptor` function to extract scan context descriptors, which capture the distinctiveness of a view, from two point clouds in the view set.
- 2 Use the `scanContextDistance` function to compute the descriptor distance between the two scan context descriptors. If the distance between two descriptors is below a specified threshold, then it is a potential loop closure.
- 3 Register the point clouds to determine the relative pose transformation between the views and the root mean square error (RMSE) of the Euclidean distance between the aligned point clouds. Use the RMSE to filter invalid loop closures. The relative pose transformation represents a connection between the two views. An edge formed by a connection between nonsuccessive views is called a loop closure edge. You can add the connection to the `pcviewset` object using the `addConnection` function.

Correct Drift

The `pcviewset` object internally updates the pose graph as views and connections are added. To minimize drift, perform pose graph optimization by using the `optimizePoses` function, once sufficient loop closure. The `optimizePoses` function returns a `pcviewset` object with the optimized absolute pose transformations for each view.

You can use the `createPoseGraph` function to return the pose graph as a MATLAB `digraph` object. You can use graph algorithms in MATLAB to inspect, view, or modify the pose graph. Use the `optimizePoseGraph` function from the Navigation Toolbox™ to optimize the modified pose graph, and then use the `updateView` function to update the poses in the view set.

Assemble Map

Use the `pcalign` function to build a point cloud map using the point clouds from the view set and their optimized absolute pose transformations.

Tips

- Local registration methods, such as those that use NDT or ICP (`pcregisterndt` or `pcregistericp`, respectively), require initial estimates. To obtain an initial estimate, use another sensor such as an inertial measurement unit (IMU) or other forms of odometry. Improving the initial estimate helps the registration algorithm converge faster.
- Specifying the higher values for the 'MaxIterations' argument or lower values for the 'Tolerance' property for more accurate registration results, but slower registration speeds.

References

- [1] Myronenko, Andriy and Xubo Song. "Point Set Registration: Coherent Point Drift." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32, no. 12 (December 2010): 2262–75.
- [2] Chen, Yang, and Gérard Medioni. "Object Modelling by Registration of Multiple Range Images." *Image and Vision Computing* 10, no. 3 (April 1992): 145–55.
- [3] Besl, P.J., and Neil D. McKay. "A Method for Registration of 3-D Shapes." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14, no. 2 (February 1992): 239–56.
- [4] Biber, P., and W. Strasser. "The Normal Distributions Transform: A New Approach to Laser Scan Matching." In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, 3:2743–48. Las Vegas, Nevada, USA: IEEE, 2003.
- [5] Magnusson, Martin. "The Three-Dimensional Normal-Distributions Transform: An Efficient Representation for Registration, Surface Analysis, and Loop Detection." Örebro University, 2009.
- [6] Dimitrievski, Martin, David Van Hamme, Peter Veelaert, and Wilfried Philips. "Robust Matching of Occupancy Maps for Odometry in Autonomous Vehicles." In *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 626–33. Rome, Italy: SCITEPRESS - Science and and Technology Publications, 2016.

See Also

Functions

`pcalign` | `pcregistercorr` | `pcregistercpd` | `pcregistericp` | `pcregisterndt`

Objects

`pcviewset` | `pointCloud`

Related Examples

- "Build a Map from Lidar Data Using SLAM" on page 5-38
- "Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment" (Automated Driving Toolbox)
- "3-D Point Cloud Registration and Stitching" on page 5-54
- "Build a Map from Lidar Data" (Automated Driving Toolbox)

The PLY Format

In this section...

"File Header" on page 9-10

"Data" on page 9-11

"Common Elements and Properties" on page 9-12

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 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
		refract_coeff	float	amount of light refracted
		refract_index	float	index of refraction
		extinct_coeff	float	extinction coefficient

See Also


pcread | pcwrite

Using the Installer for Computer Vision System Toolbox Product

- “Install Computer Vision Toolbox Add-on Support Files” on page 10-2
- “Install OCR Language Data Files” on page 10-3
- “Install and Use Computer Vision Toolbox Interface for OpenCV in MATLAB” on page 10-6
- “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 10-11
- “Smile Detection by Using OpenCV Code in Simulink” on page 10-19
- “Convert RGB Image to Grayscale Image by Using OpenCV Importer” on page 10-29
- “Draw Different Shapes by Using OpenCV Code in Simulink” on page 10-36

Install Computer Vision Toolbox Add-on Support Files

After you install third-party support files, you can use the data with the Computer Vision Toolbox product. To install the Add-on support files, use one of the following methods:

-  **Get Support Package Now**
- 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 10-3

“Pretrained Language Data and the ocr function” on page 10-3

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 Toolbox product. To install the Add-on support files, use one of the following methods:

-  Get Support Package Now
- 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)” on page 4-43

Install and Use Computer Vision Toolbox Interface for OpenCV in MATLAB

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 10-6

“Support Package Contents” on page 10-6

“Create MEX-File from OpenCV C++ file” on page 10-7


“Use the Computer Vision Toolbox Interface for OpenCV in MATLAB C++ API” on page 10-7

“Create Your Own OpenCV MEX-files” on page 10-8

“Run OpenCV Examples” on page 10-8

Installation

After you install third-party support files, you can use the data with the Computer Vision Toolbox product. To install the Add-on support files, use one of the following methods:

-  **Get Support Package Now**
- 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 Computer Vision Toolbox Interface for OpenCV in MATLAB 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 <code>README.txt</code> file with step-by-step instructions.

Files	Contents
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 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 or Visual Studio 2017
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 Computer Vision Toolbox Interface for OpenCV in MATLAB 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 3.0 or higher. See the Parallel Computing Toolbox™ System Requirements, The GPU utility functions require the Parallel Computing Toolbox software.

The Computer Vision Toolbox Interface for OpenCV in MATLAB API supports OpenCV version 4.2.0.

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

Function	Description
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 mexOpenCV 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 Computer Vision Toolbox Interface for OpenCV in MATLAB 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 matchTemplate0CV.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 backgroundSubtractor0CV.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 detectORBFeatures0CV.cpp
```

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

```
mexOpenCV extractORBFeatures0CV.cpp
```

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

```
testORBFeatures0CV.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 detectORBFeatures0CV_GPU.cpp -lgpu -lmwocvcpumex -largeArrayDims
```

Linux/Mac:

```
mexOpenCV detectORBFeatures0CV_GPU.cpp -lmwgpu -lmwocvcpumex -largeArrayDims
```

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

```
testORBFeatures0CV_GPU.m
```

See Also

“C Matrix API” | `mxArray`

More About

- “Install Computer Vision Toolbox Add-on Support Files” on page 10-2
- Using OpenCV with MATLAB

Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink

In this section...

“Installation” on page 10-11

“Import OpenCV Code into Simulink” on page 10-11

“Limitations” on page 10-18

You can import OpenCV code to a Simulink model by using the **OpenCV Importer** application. The **OpenCV Importer** application is available only after you install the Computer Vision Toolbox Interface for OpenCV in Simulink support package.

Installation

To install the support package, first click the **Add-Ons** drop-down list on the MATLAB **Home** tab, and then select **Get Add-Ons**. In the **Add-Ons Explorer** window, find and click the Computer Vision Toolbox Interface for OpenCV in Simulink support package, and then click **Install**.

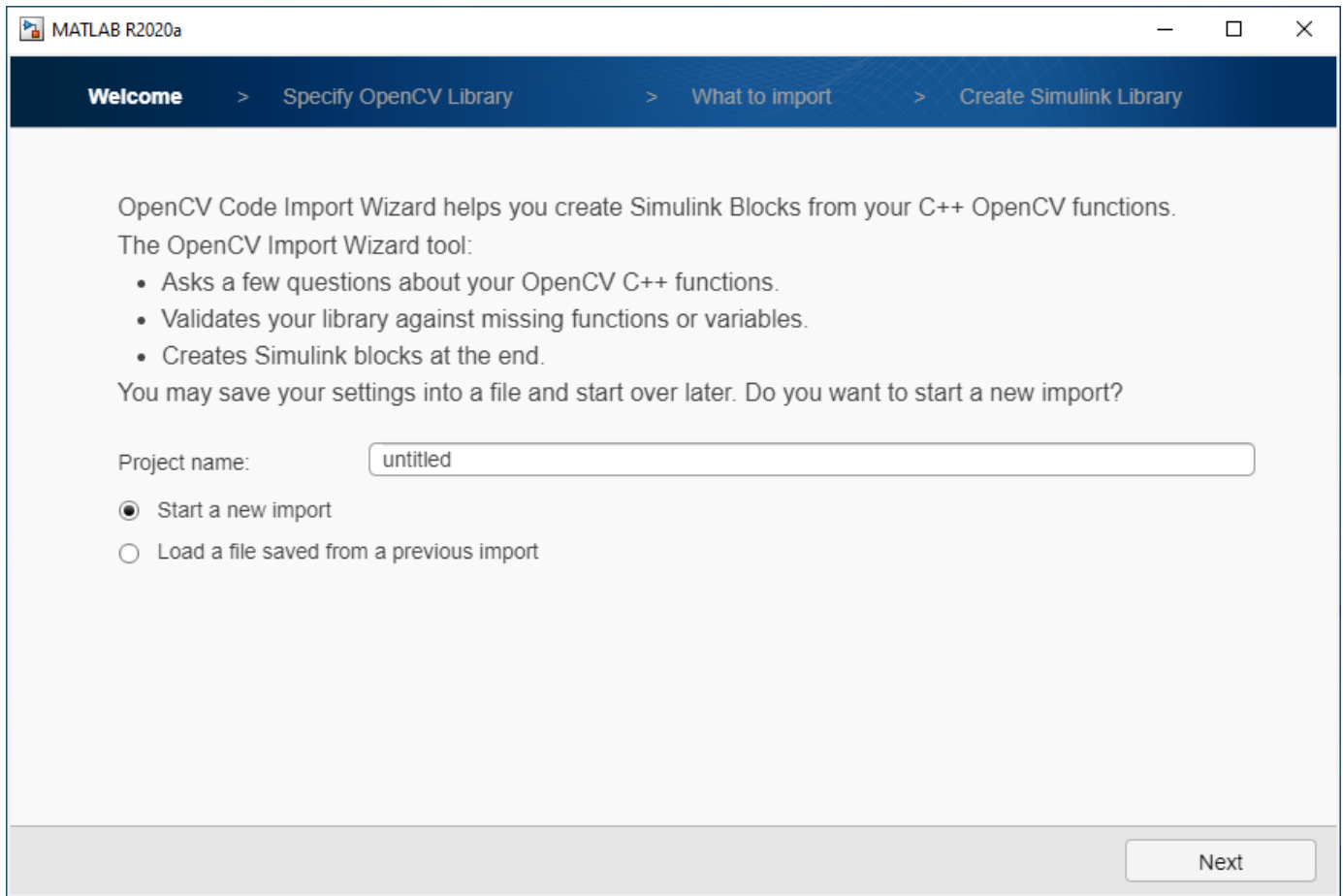
Import OpenCV Code into Simulink

To start the **OpenCV Importer**, click **Apps** on the MATLAB Toolstrip. Click the down arrow to show more options. Under **My Apps**, click the **OpenCV Importer** app icon. Alternatively, you can start the **OpenCV Importer** from the command-line interface. At the MATLAB command line, enter:

```
Simulink.OpenCVImporter
```

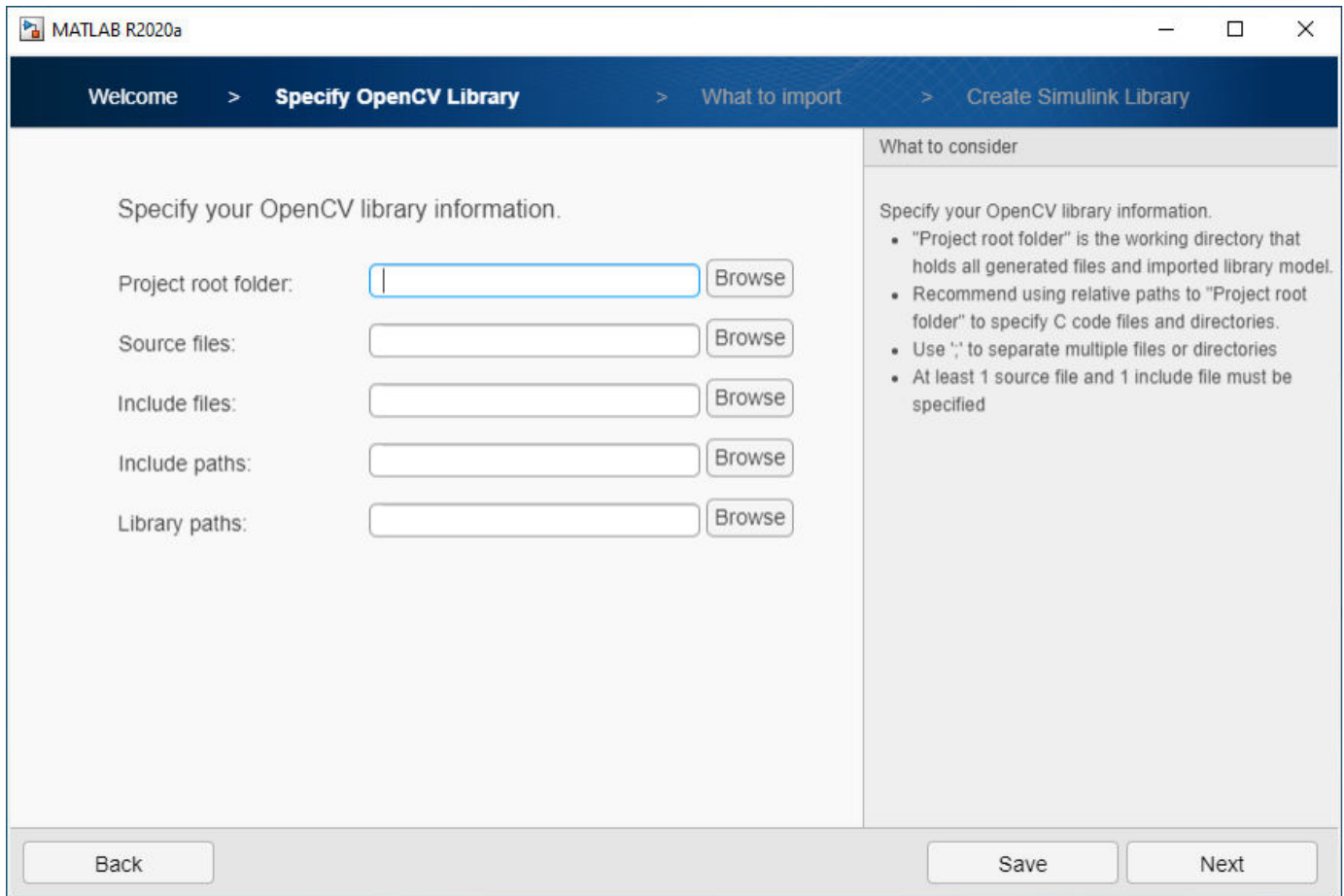
The OpenCV import wizard opens to a Welcome page.

- 1 In the **Project name** field, specify a name for your import. You can either start a new import or load files saved from a previous import. The projects are saved in `.m` file format. To browse a saved file from previous import, select **Load a file saved from a previous import**. Click **Next**.

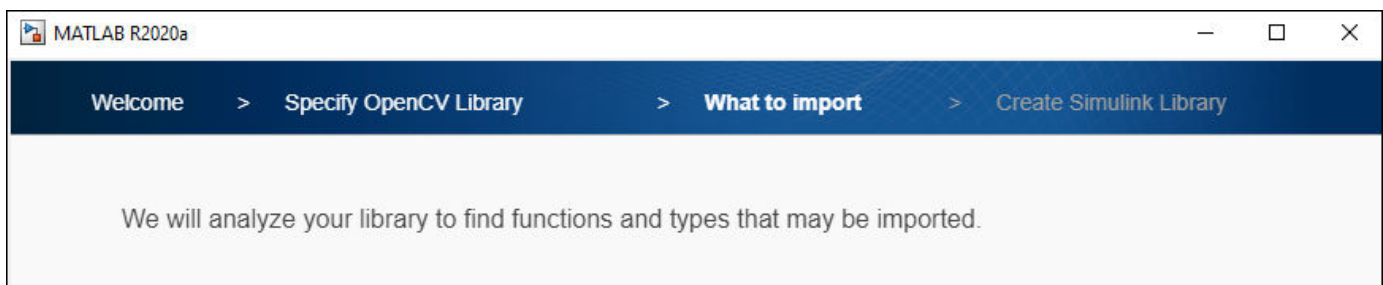


- 2 In the Specify OpenCV Library page, specify your C++ library information. If you import a previously saved project file, all the fields are autopopulated.
- **Project root folder:** A writable folder path where you want to save your output files (wrapper files and Simulink library).
 - **Source files:** OpenCV source file path. Specify the `.cpp` file format. If you provide an absolute path, then the wizard uses the file from the specified location. If you do not provide the absolute path, then the wizard uses the path relative to the project root.
 - **Include files:** Header files path. Specify the `.hpp` file format. If you provide an absolute path, then the wizard uses the file from the specified location. If you do not provide the absolute path, then the wizard uses the path relative to the project root.
 - **Include paths:** Define any additional include folders (Folder information). MATLAB OpenCV include files are included.
 - **Library paths:** Specify the path to external library files.

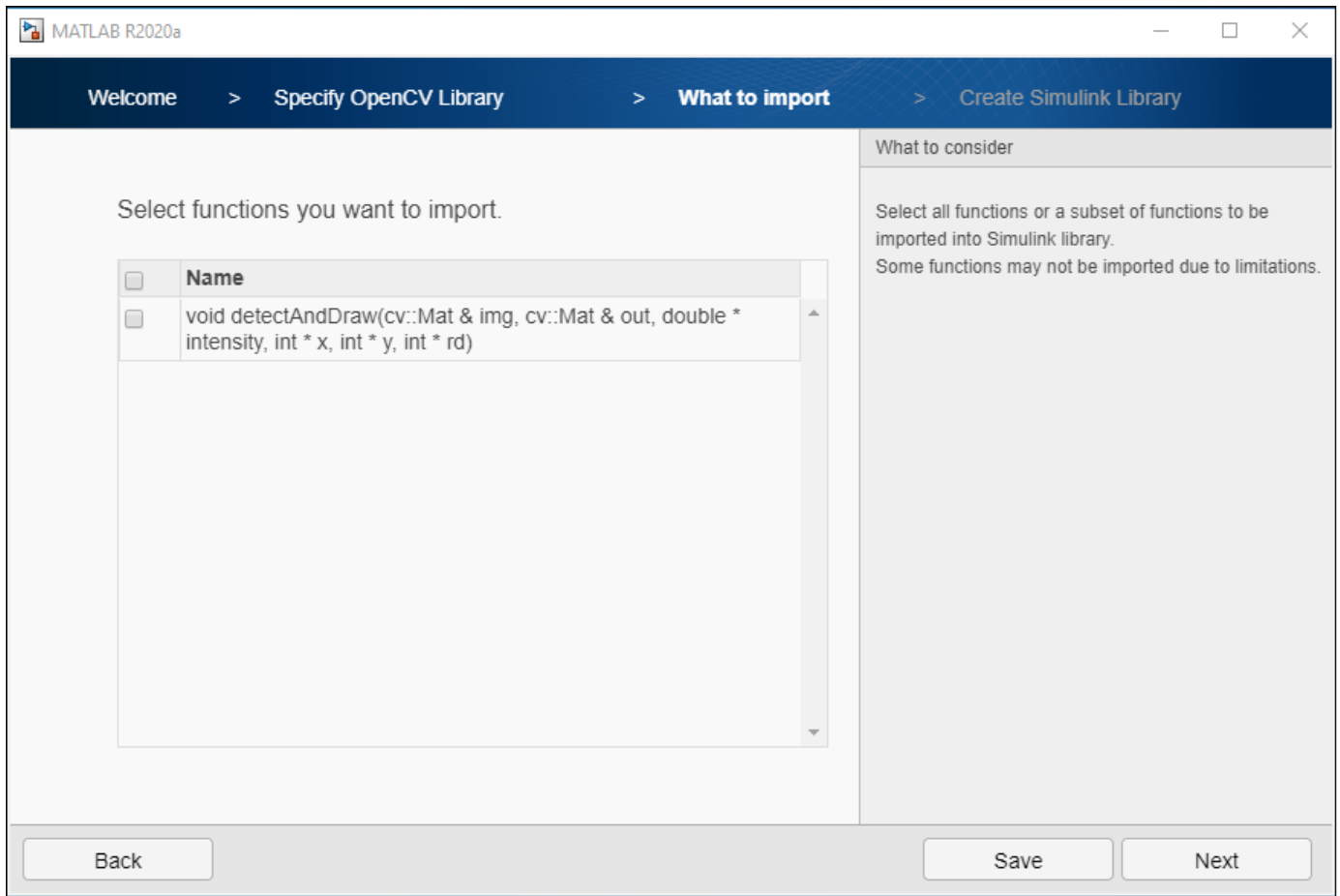
If you have multiple files or folders to specify, use a semicolon-separated list of files or folders. Click **Next**.



- 3 To find functions and types that are supported for import, analyze your library by clicking **Next**. Once the analysis is complete, click **Next**.

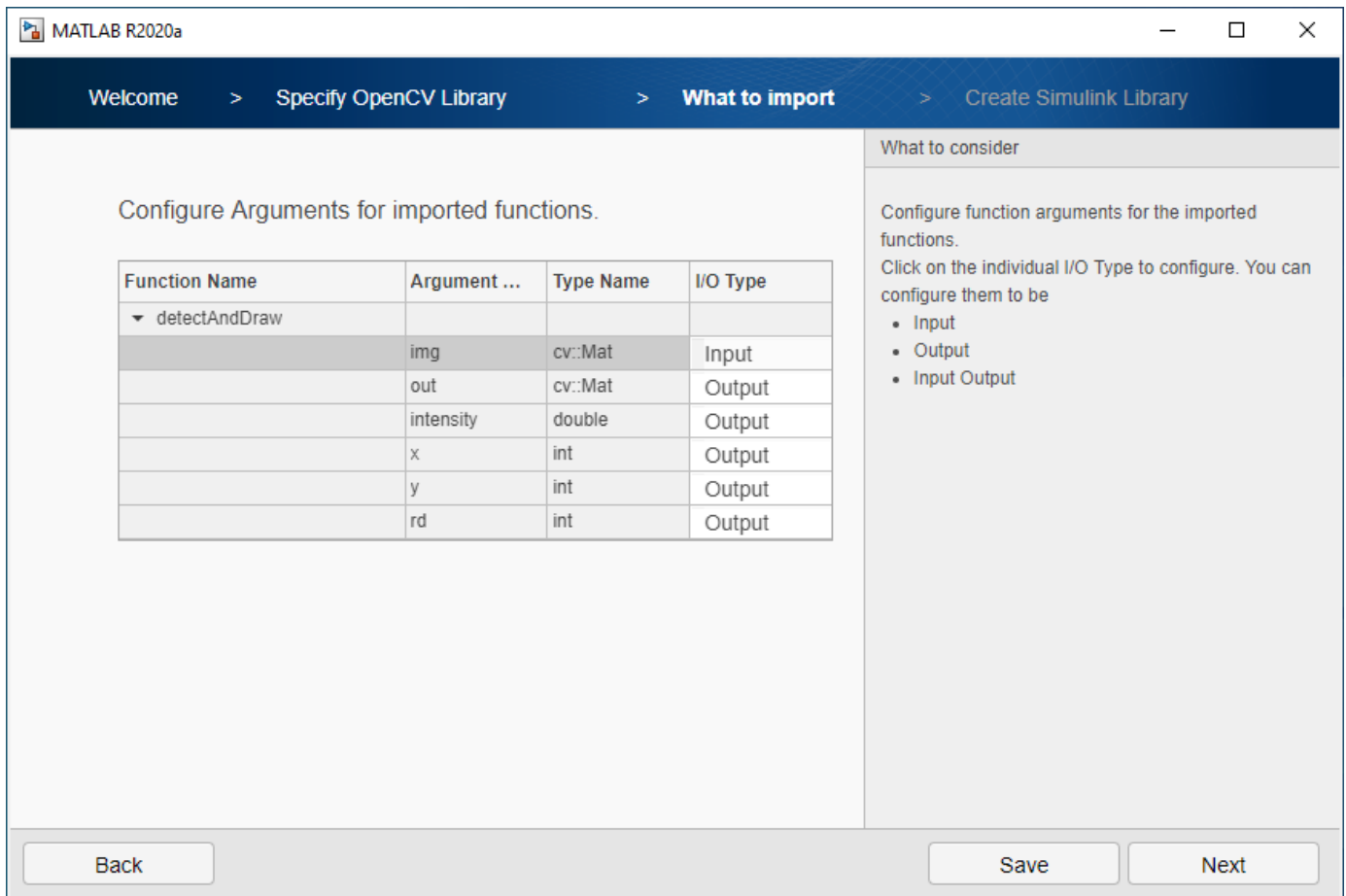


- 4 In the What to Import page, the functions that Computer Vision Toolbox Interface for OpenCV in Simulink supports are listed. Select the functions that you want to import into Simulink library and click **Next**.



- 5 Each **I/O Type** corresponds to the OpenCV function argument to map into the Simulink model. These different **I/O Type** are supported:
- Input- for input arguments
 - Output- for output arguments
 - InputOutput- for input output arguments

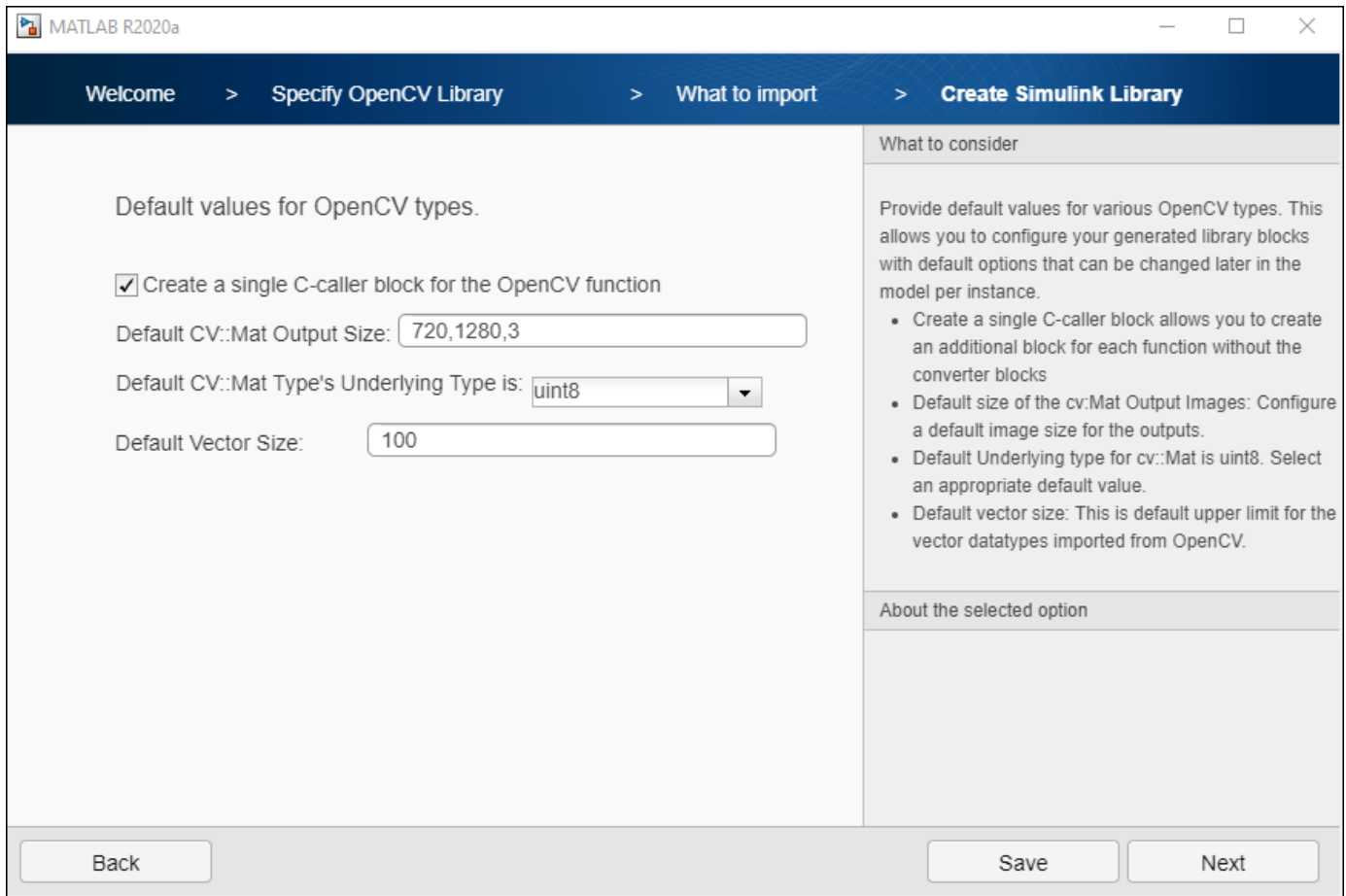
To select the input/output types, double-click the **Output** option in the **I/O Type** column drop-down list, and then click **Next**.



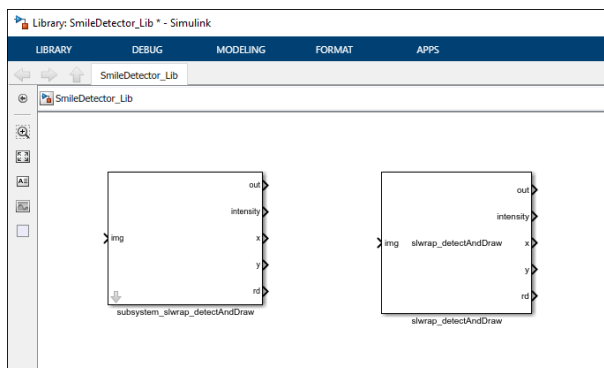
- 6 In the Create Simulink Library page, you can generate either just a subsystem block or a subsystem block and a C Caller block of the selected function. A C Caller block integrates your OpenCV data into Simulink. The generated subsystem block contains C Caller blocks configured by using data conversion blocks.

To generate a subsystem block and a C Caller block, select **Create a single C-caller block for the OpenCV function**, and then click **Next**.

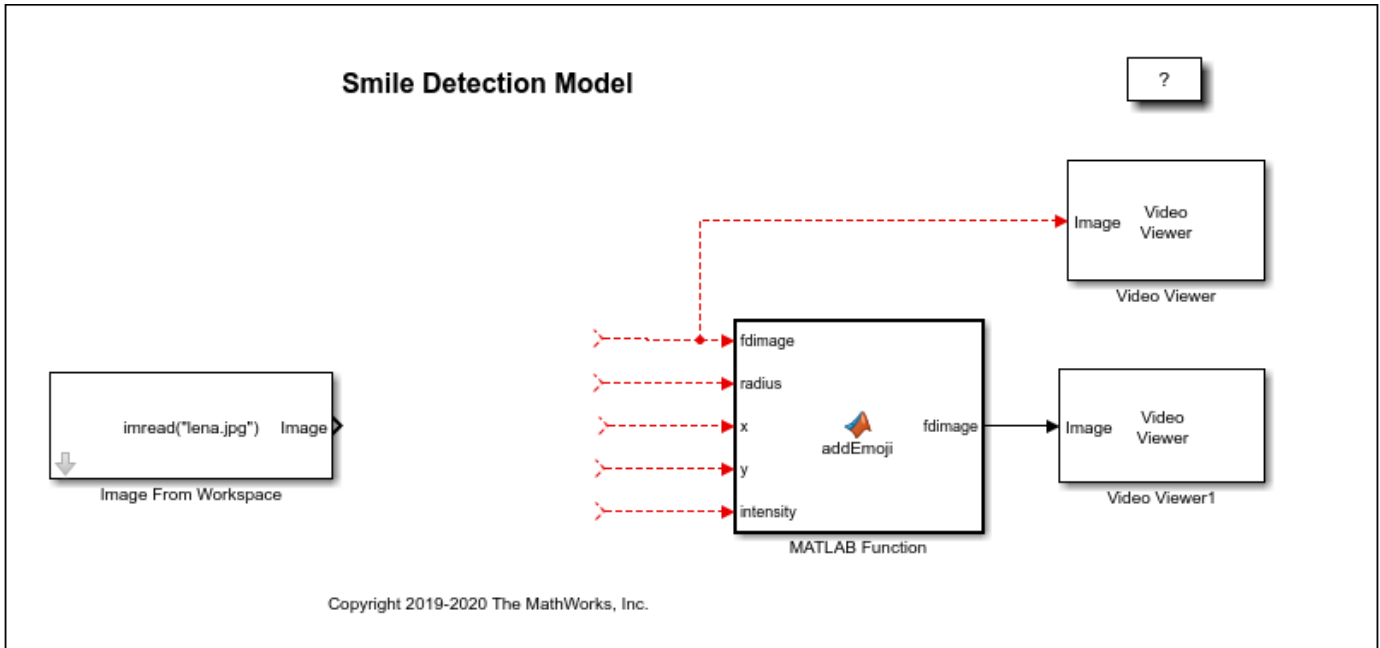
If the OpenCV code contains a `Mat` data type, the default output size is $(720, 1280, 3)$ and the default underlying type is `uint8`. For vectors, the default size is 100. You can change the default size based on your model requirements.



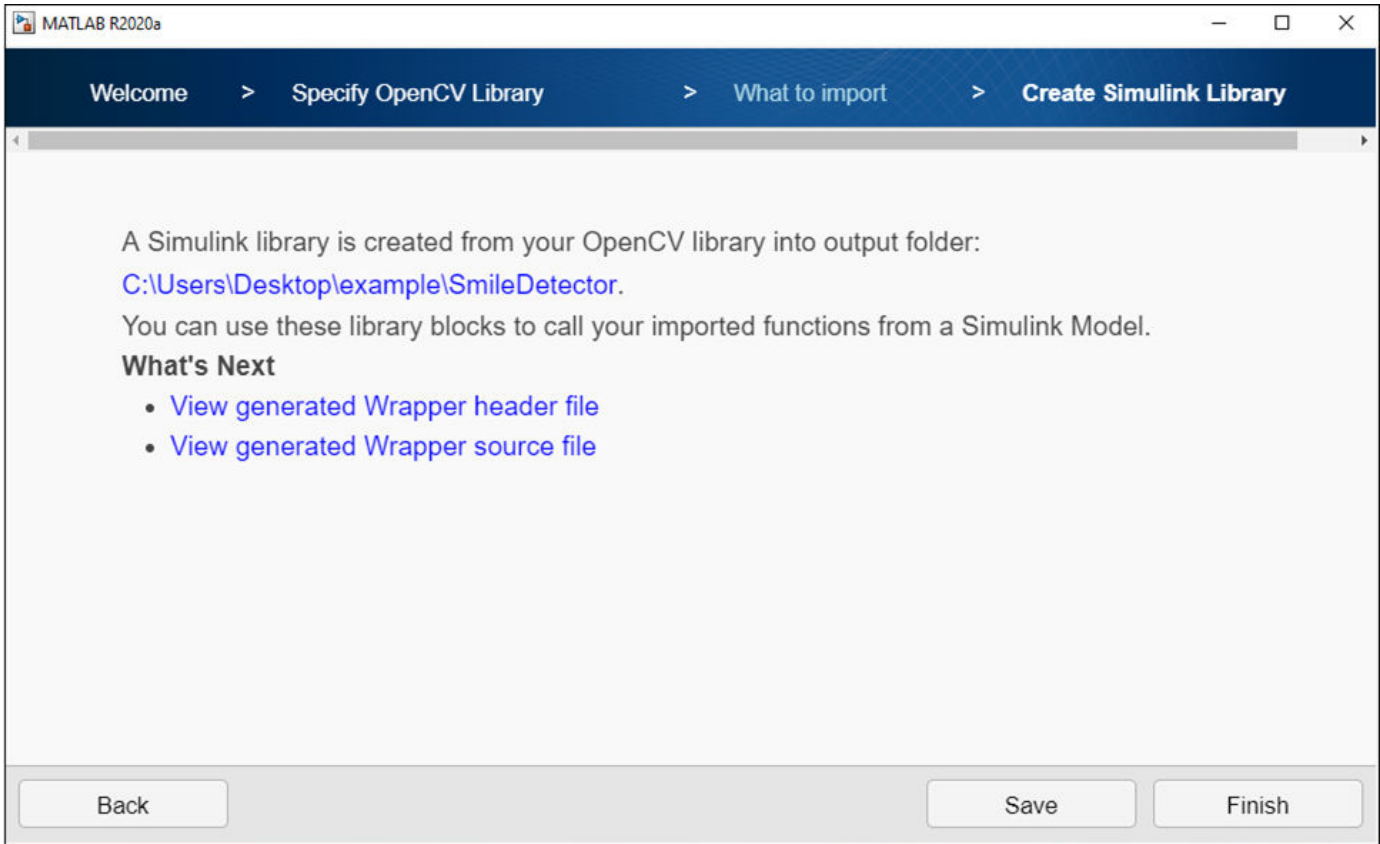
- 7 A Simulink library `Projectname_Lib.slx` is created from your OpenCV code into a project root folder. The library contains a subsystem block and a C Caller block.



You can drag any of these blocks to your model, connect them to the existing blocks in the model, and run the model simulation.



The wizard also creates wrapper files for source and header files.



Limitations

The Computer Vision Toolbox Interface for OpenCV in Simulink support package:

- Uses OpenCV as part of MATLAB third-party support. You can get the OpenCV additional capabilities in **Add-Ons** (Computer Vision Toolbox).
- Does not support external OpenCV libraries (for instance, `opencv_contrib`).
- Does not support `InputArray`, `OutputArray`, and `InputOutputArray` data types.
- Requires Microsoft Visual Studio 2015 or 2017 Professional and Community editions for Windows 64 operating system. For more information on compilers, see [Compilers Used to Build OpenCV Libraries](#).
- Supports C++ code generation that uses row-major array layout.

See Also

[FromOpenCV](#) | [ToOpenCV](#)

More About

- “Smile Detection by Using OpenCV Code in Simulink” on page 10-19
- “Convert RGB Image to Grayscale Image by Using OpenCV Importer” on page 10-29
- “Draw Different Shapes by Using OpenCV Code in Simulink” on page 10-36

Smile Detection by Using OpenCV Code in Simulink

In this section...

“Required Products” on page 10-19

“Set Up Your C++ Compiler” on page 10-19

“Model Description” on page 10-19

“Copy Example Folder to a Writable Location” on page 10-20

“Step 1: Import OpenCV Function to Create a Simulink Library” on page 10-21

“Step 2: Use Generated Subsystem in Simulink Model” on page 10-25

“Step 3: Simulate the Smile Detector” on page 10-26

“Step 4: Generate C++ Code from the Smile Detector Model” on page 10-27

“Deploy the Smile Detector on the Raspberry Pi Hardware” on page 10-28

This example shows how to build a smile detector by using the **OpenCV Importer**. The detector estimates the intensity of the smile on a face image. Based on the estimated intensity, the detector identifies an appropriate emoji from its database, and then places the emoji on the smiling face.

First import an OpenCV function into Simulink by using the OpenCV Code Import Wizard on page 10-11. The wizard creates a Simulink library that contains a subsystem and a C Caller block for the specified OpenCV function. The subsystem is then used in a preconfigured Simulink model to accept the face image for smile detection. You can generate C++ code from the model, and then deploy the code on your target hardware.

You learn how to:

- Import an OpenCV function into a Simulink library.
- Use blocks from a generated library in a Simulink model.
- Generate C++ code from a Simulink model.
- Deploy the model on the Raspberry Pi hardware.

Required Products

- Computer Vision Toolbox Interface for OpenCV in Simulink
- Computer Vision Toolbox
- Embedded Coder® (for deployment)

Set Up Your C++ Compiler

To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in *Compiler Used to Build OpenCV Libraries*. Configure the identified compiler by using the `mex -setup c++` command. For more information, see *Choose a C++ Compiler*.

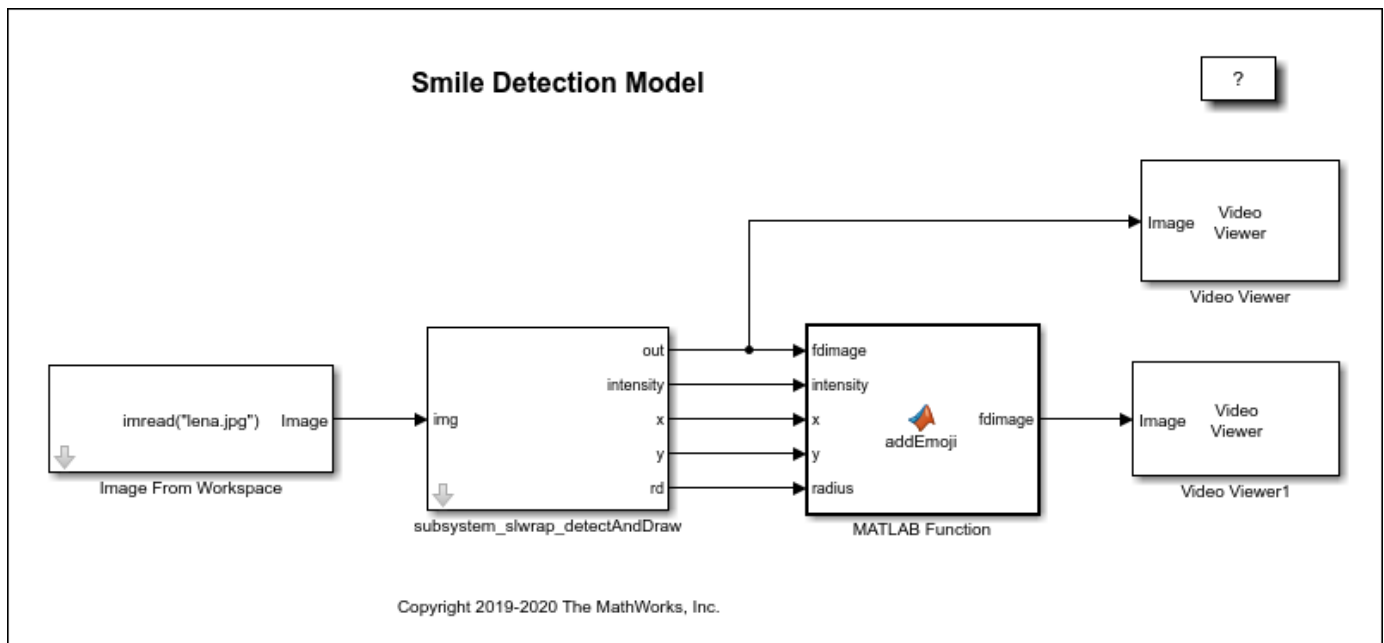
Model Description

In this example, a smile detector is implemented by using the Simulink model `smileDetect.slx`.

In this model, the subsystem_slwrap_detectAndDraw subsystem resides in the Smile_Detect_Lib library. You create the subsystem_slwrap_detectAndDraw subsystem by using the **OpenCV Importer**. The subsystem accepts a face image from the Image From Workspace block and provides these output values.

Output Port	Description
out	Face image with a circle
intensity	Intensity of the smile
x	x coordinate of center of the circle
y	y coordinate of center of the circle
rd	Radius of the circle

The MATLAB Function block accepts input from the subsystem_slwrap_detectAndDraw subsystem block. The MATLAB Function block has a set of emoji images. The smile intensity of the emoji in these images ranges from low to high. From the emoji images, the block identifies the most appropriate emoji for the estimated intensity and places it on the face image. The output is then provided to the Video Viewer blocks.



Copy Example Folder to a Writable Location

To access the path to the example folder, at the MATLAB command line, enter:

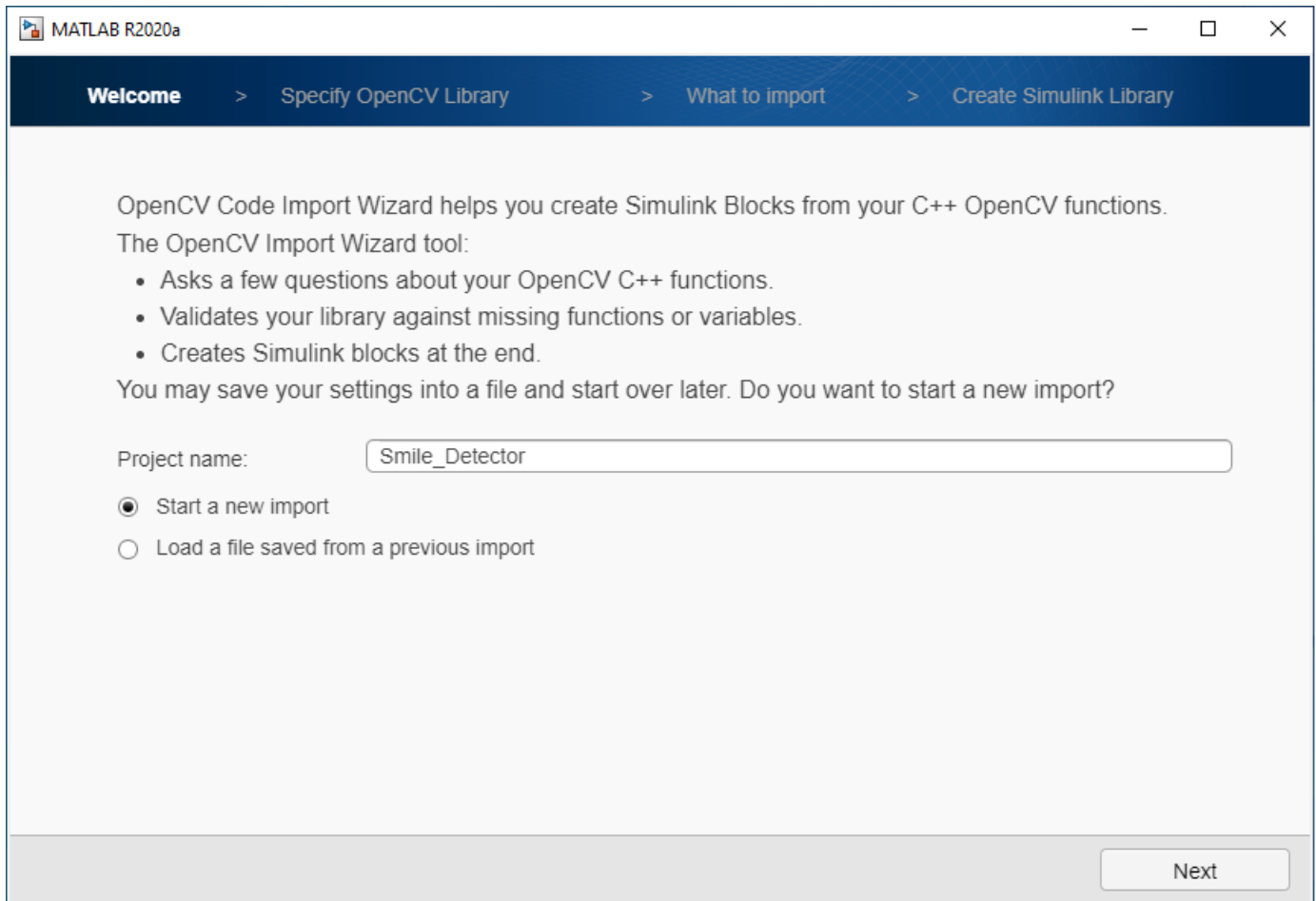
```
OpenCVSimulinkExamples;
```

Each subfolder contains all the supporting files required to run the example.

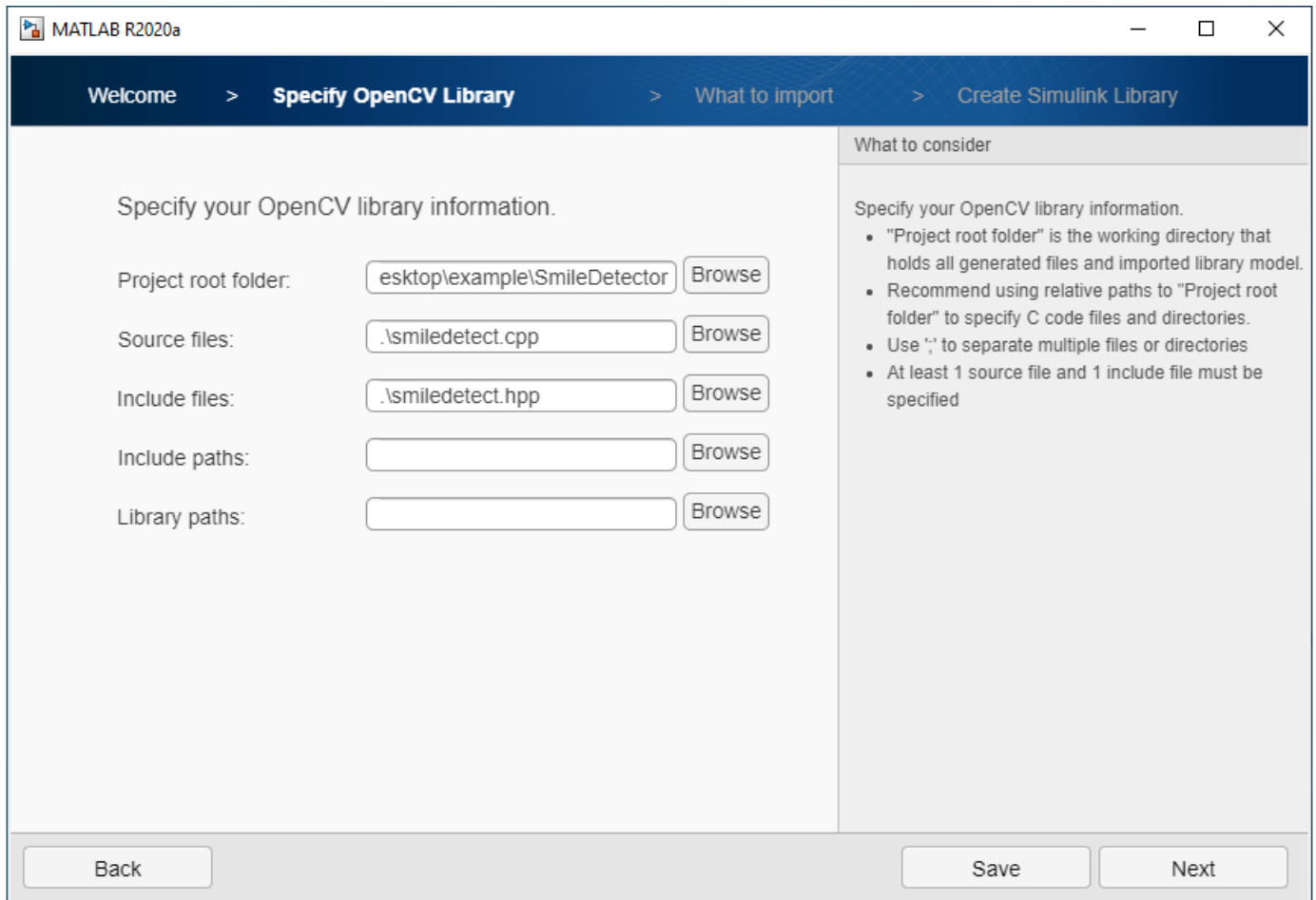
Before proceeding with these steps, ensure that you copy the example folder to a writable folder location and change your current working folder to `...example\SmileDetector`. All your output files are saved to this folder.

Step 1: Import OpenCV Function to Create a Simulink Library

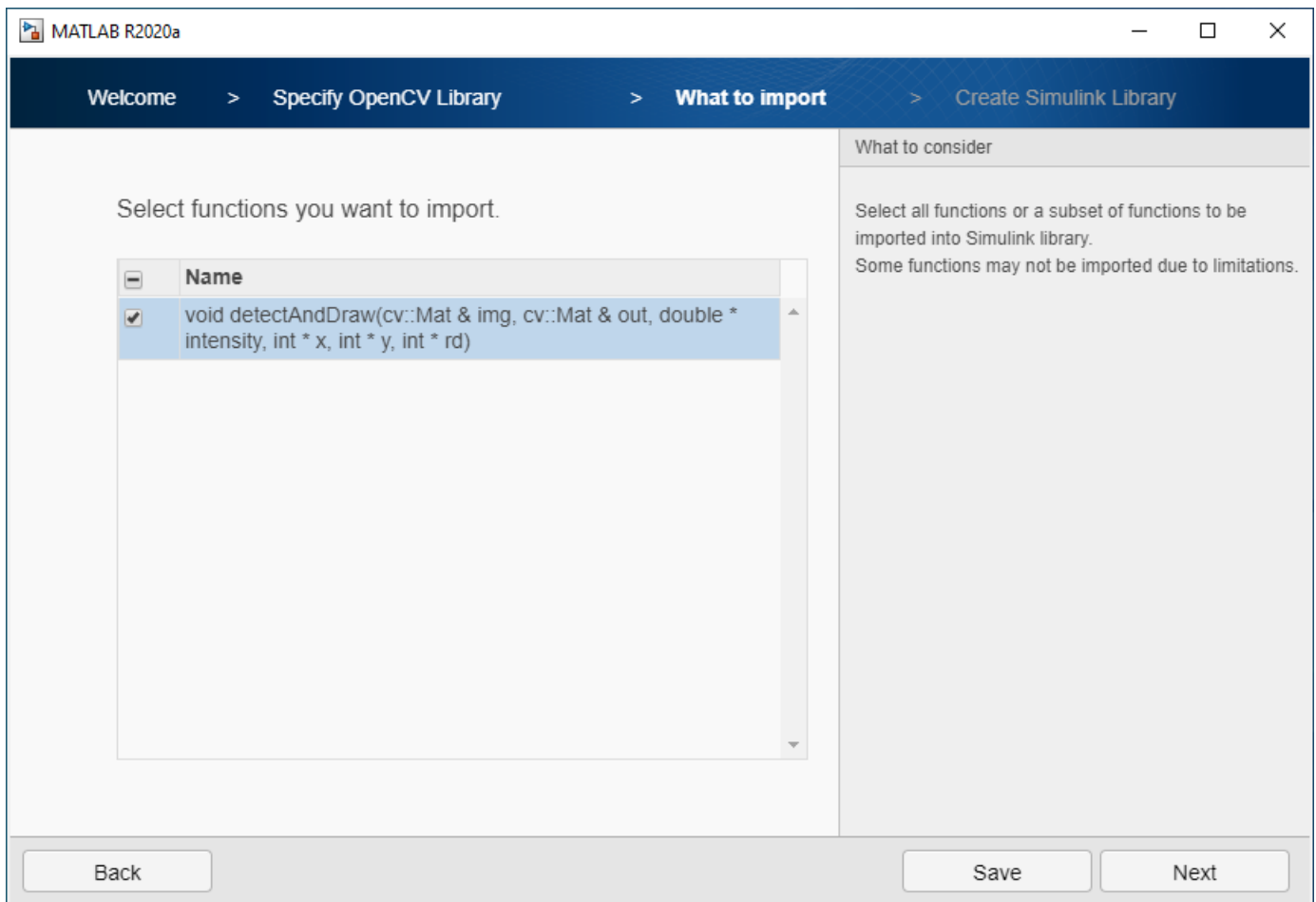
- 1 To start the **OpenCV Importer** app, click **Apps** on the MATLAB Toolstrip. In the Welcome page, specify the **Project name** as `Smile_Detector`. Make sure that the project name does not contain any spaces. Click **Next**.



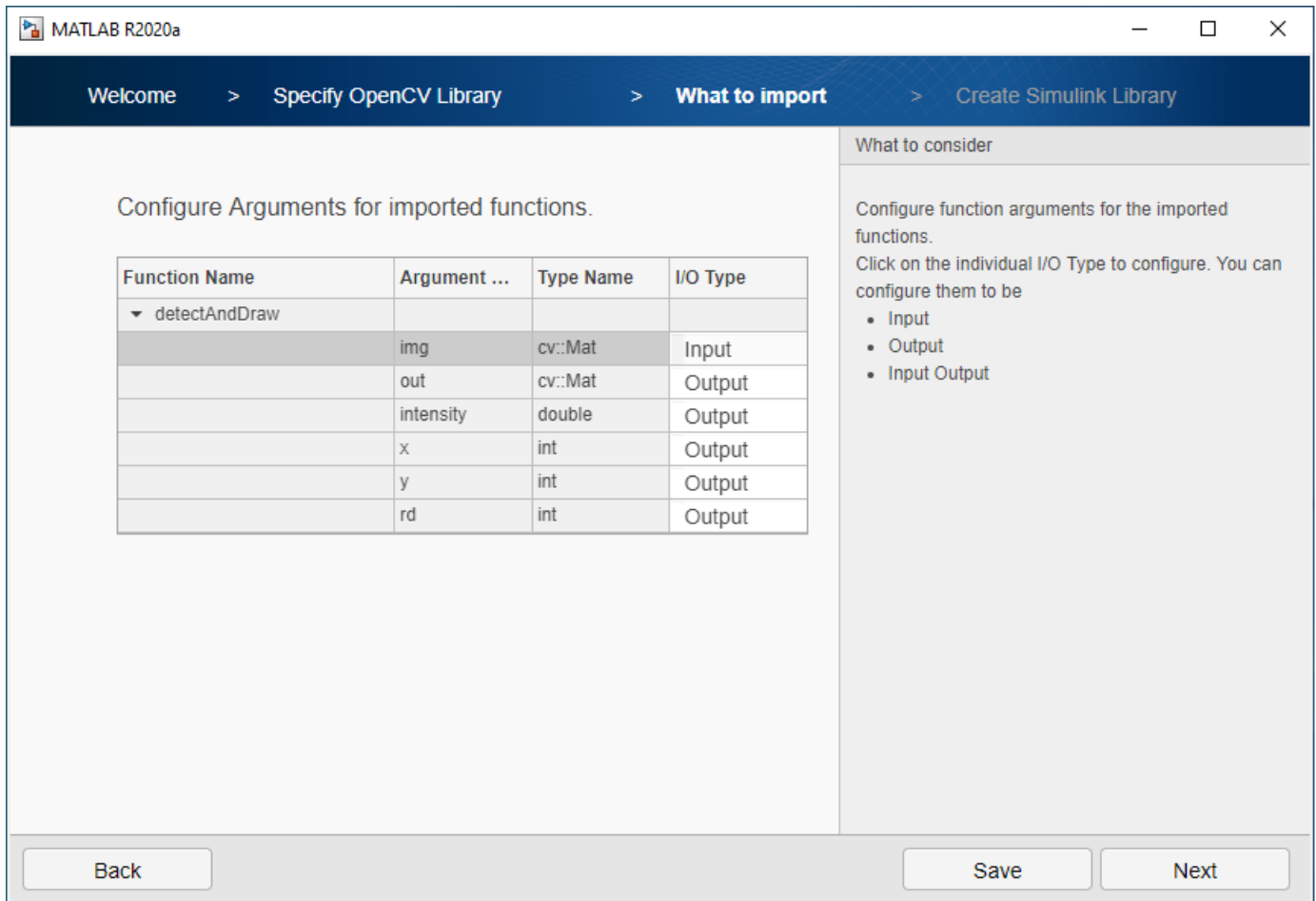
- 2 In Specify OpenCV Library, specify these file locations, and then click **Next**.
 - **Project root folder:** Specify the path of your example folder. This path is the path to the writable project folder where you have saved your example files. All your output files are saved to this folder.
 - **Source files:** Specify the path of the `.cpp` file located inside your project folder as `smiledetect.cpp`.
 - **Include files:** Specify the path of the `.hpp` header file located inside your project folder as `smiledetect.hpp`.



- 3 Analyze your library to find the functions and types for import. Once the analysis is complete, click **Next**. Select the detectAndDraw function and click **Next**.

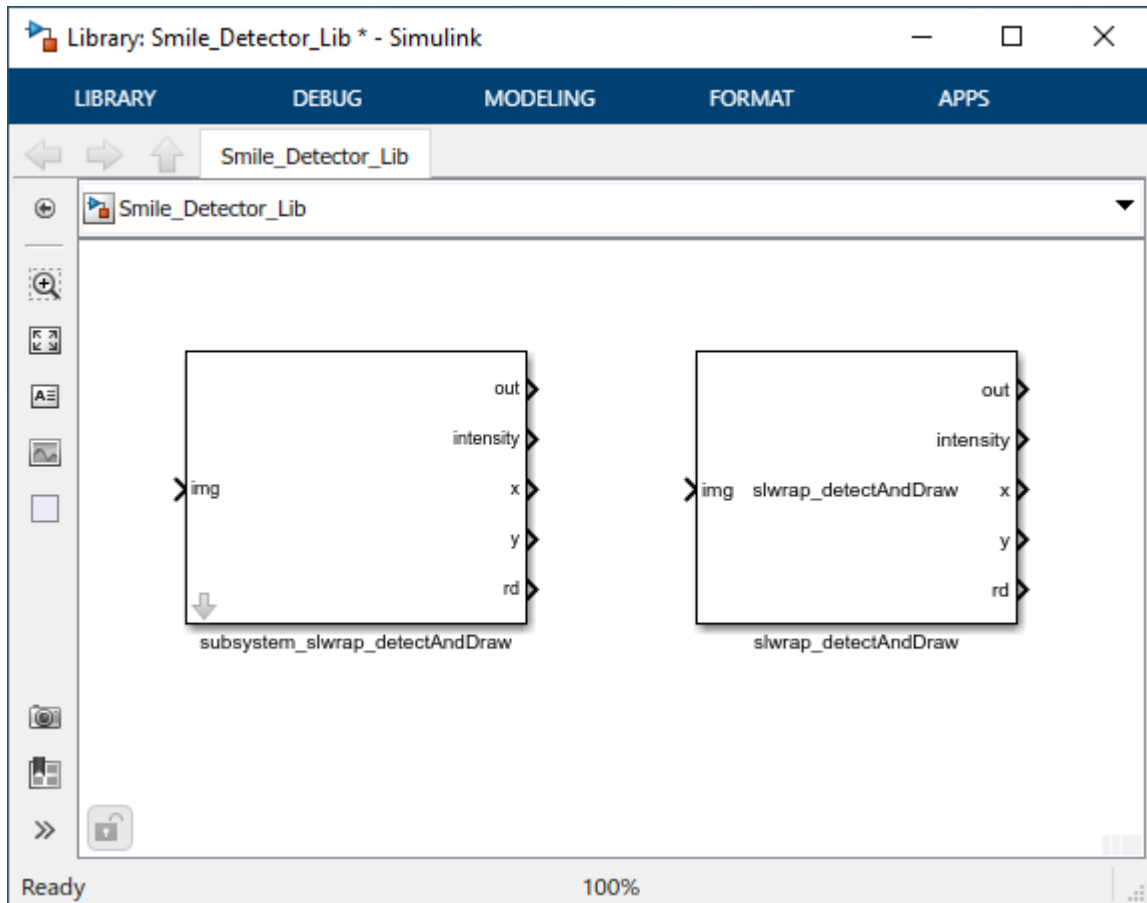


- 4 From What to import, select the I/O Type for img as Input, and then click **Next**.



5 In Create Simulink Library, verify the default values and click **Next**.

A Simulink library `Smile_Detector_Lib` is created from your OpenCV code into the project root folder. The library contains a subsystem and a C Caller block. You can use any of these blocks for model simulation. In this example, the subsystem `subsystem_slwrap_detectAndDraw` is used.



Step 2: Use Generated Subsystem in Simulink Model

To use the generated subsystem `subsystem_slwrap_detectAndDraw` with the Simulink model `smileDetect.slx`:

- 1 In your MATLAB **Current Folder**, right-click the model `smileDetect.slx` and click **Open** from the context menu. In the model, delete the existing `subsystem_slwrap_detectAndDraw` subsystem and drag the generated subsystem `subsystem_slwrap_detectAndDraw` from the `Smile_Detector_Lib` library to the model. Connect the subsystem to the MATLAB Function block.
- 2 Double-click the subsystem and specify these parameter values.


Parameter	Value	Description
Rows	512	Number of rows in the output image
Columns	512	Number of columns in the output image
Channels	3	Number of channels in the output image

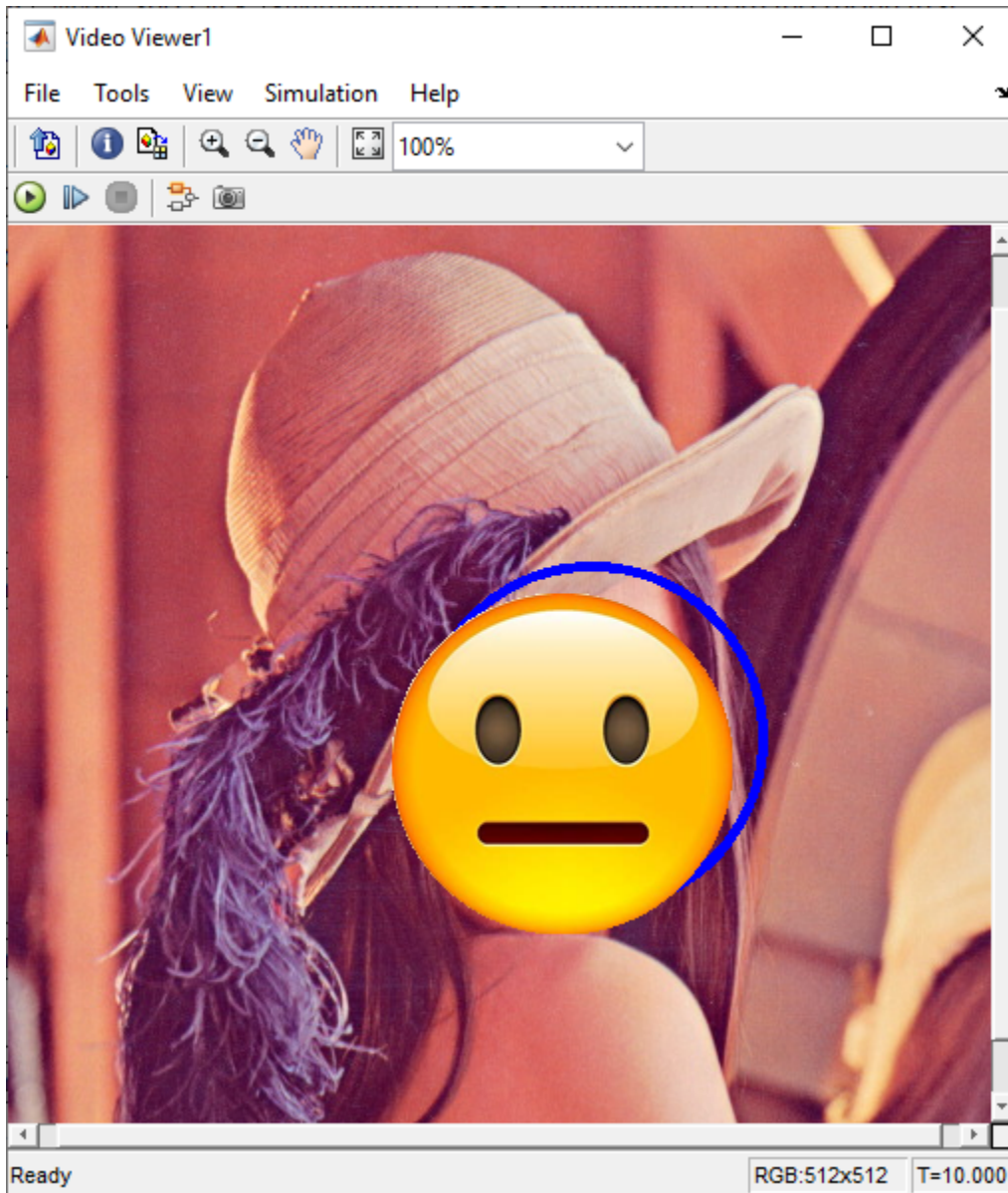
Parameter	Value	Description
Underlying Type	uint8	Underlying data type of OpenCV Mat
is Image	on	Whether input is an image or a matrix

Click **Apply**, and then click **OK**.

Step 3: Simulate the Smile Detector



On the Simulink Toolstrip, in the **Simulation** tab, click  to simulate the model. After the simulation is complete, the Video Viewer block displays an image with an emoji on the face. The emoji represents the intensity of the smile.



Step 4: Generate C++ Code from the Smile Detector Model

Before you generate the code from the model, you must first ensure that you have write permission in your current folder.

To generate C++ code:

- 1 Open the `smileDetect_codegen.slx` model from your MATLAB **Current Folder**.

To review the model settings:

On the **Apps** tab on the Simulink toolstrip, select Embedded Coder.

On the **C++ Code** tab in the **Settings** drop-down list, click **C/C++ Code generation settings** to open the Configuration Parameters dialog box and verify these settings:

- In the **Code Generation** pane, under **Target selection**, **Language** is set to C++.
 - In the **Interface** under **Code Generation**, **Array layout** in the **Data exchange interface** category is set to Row-major.
- 2 Connect the generated subsystem `subsystem_slwrap_detectAndDraw` to the MATLAB Function block.
 - 3 To generate C++ code, under the **C++ Code** tab, click the **Generate Code** drop-down list, and then click **Build**. After the model finishes generating code, the Code Generation Report opens. You can inspect the generated code. The build process creates a zip file called `smileDetect_with_ToOpenCV.zip` in your current MATLAB working folder.

Deploy the Smile Detector on the Raspberry Pi Hardware

Before you deploy the model, connect the Raspberry Pi to your computer. Wait until the PWR LED on the hardware starts blinking.

In the **Settings** drop-down list, click **Hardware Implementation** to open the Configuration Parameters dialog box and verify these settings:

- Set the **Hardware board** to **Raspberry Pi**. The **Device Vendor** is set to **ARM Compatible**.
- In the **Code Generation** pane, under **Target selection**, **Language** is set to C++. Under **Build process**, **Zip file name** is set to `smileDetect_with_ToOpenCV.zip`. Under **Toolchain settings**, the **Toolchain** is specified as **GNU GCC Raspberry Pi**.

To deploy the code to your Raspberry Pi hardware:

- 1 From the generated zip file, copy these files to your Raspberry Pi hardware.
 - `smiledetect.zip`
 - `smileDetect.mk`
 - `main.cpp`
- 2 In Raspberry Pi, go to the location where you saved the files. To generate an `elf` file, enter this command:

```
make -f smileDetect.mk
```
- 3 Run the executable on Raspberry Pi. After successful execution, you see the output on Raspberry Pi with an emoji placed on the face image.

```
smileDetect.elf
```

See Also

[FromOpenCV | ToOpenCV](#)

More About

- “Convert RGB Image to Grayscale Image by Using OpenCV Importer” on page 10-29
- “Draw Different Shapes by Using OpenCV Code in Simulink” on page 10-36

Convert RGB Image to Grayscale Image by Using OpenCV Importer

In this section...

“Required Products” on page 10-29

“Set Up Your C++ Compiler” on page 10-29

“Model Description” on page 10-29

“Copy Example Folder to a Writable Location” on page 10-30

“Step 1: Import OpenCV Function to Create a Simulink Library” on page 10-30

“Step 2: Use Generated Subsystem in Simulink Model” on page 10-33

“Step 3: Simulate the RGB to Gray Converter” on page 10-34

This example shows how to convert an RGB image to a grayscale image by using the **OpenCV Importer**. The converter converts an RGB image to a grayscale image by eliminating the hue and saturation information while retaining the luminance.

First import an OpenCV function into Simulink by using the OpenCV Code Import Wizard on page 10-11. The wizard creates a Simulink library that contains a subsystem and a C Caller block for the specified OpenCV function. The subsystem is then used in a preconfigured Simulink model to accept the RGB image for conversion.

You learn how to:

- Import an OpenCV function into a Simulink library.
- Use blocks from a generated library in a Simulink model.

Required Products

- Computer Vision Toolbox Interface for OpenCV in Simulink
- Computer Vision Toolbox

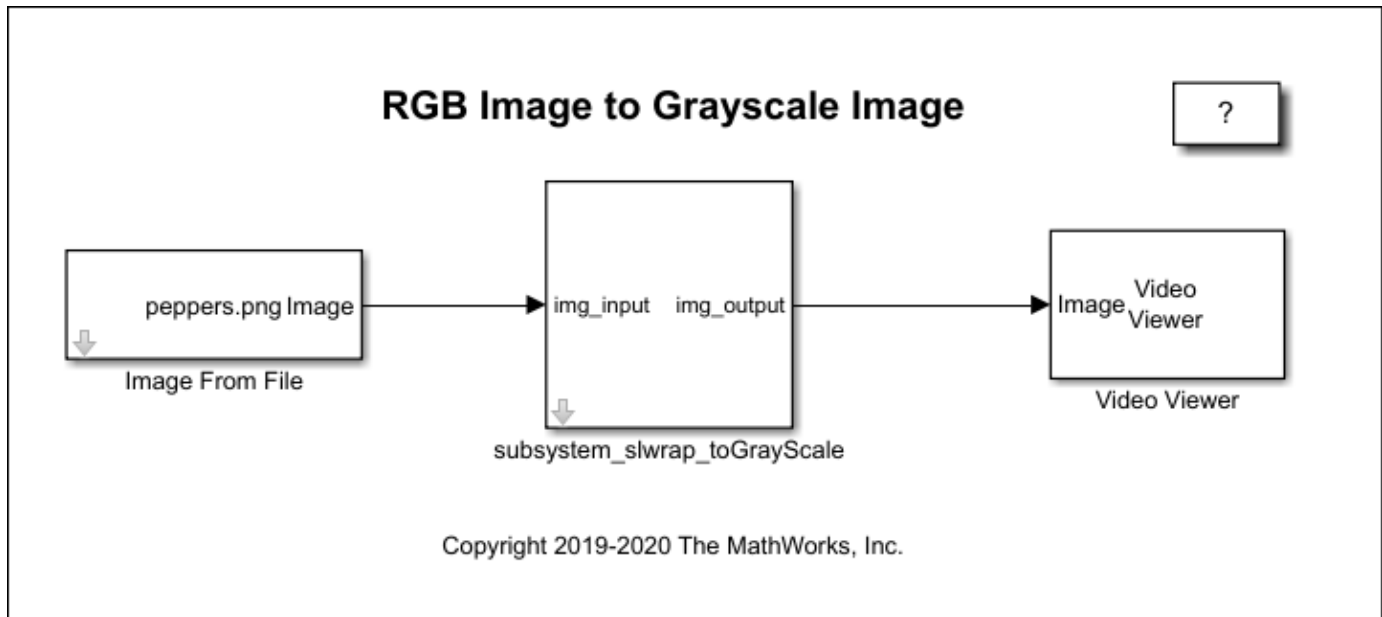
Set Up Your C++ Compiler

To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in *Compiler Used to Build OpenCV Libraries*. Configure the identified compiler by using the `mex -setup c++` command. For more information, see *Choose a C++ Compiler*.

Model Description

This example uses the Simulink model `ToGrayScale.slx`.

In this model, the `subsystem_slwrap_toGrayScale` subsystem resides in the `RGBtoGRAY_Lib` library. You create the `subsystem_slwrap_toGrayScale` subsystem by using the **OpenCV Importer**. The subsystem accepts an RGB image from the Image From File block and converts it to a grayscale output image. The output is then displayed on a Video Viewer block.



Copy Example Folder to a Writable Location

To access the path to the example folder, at the MATLAB command line, enter:

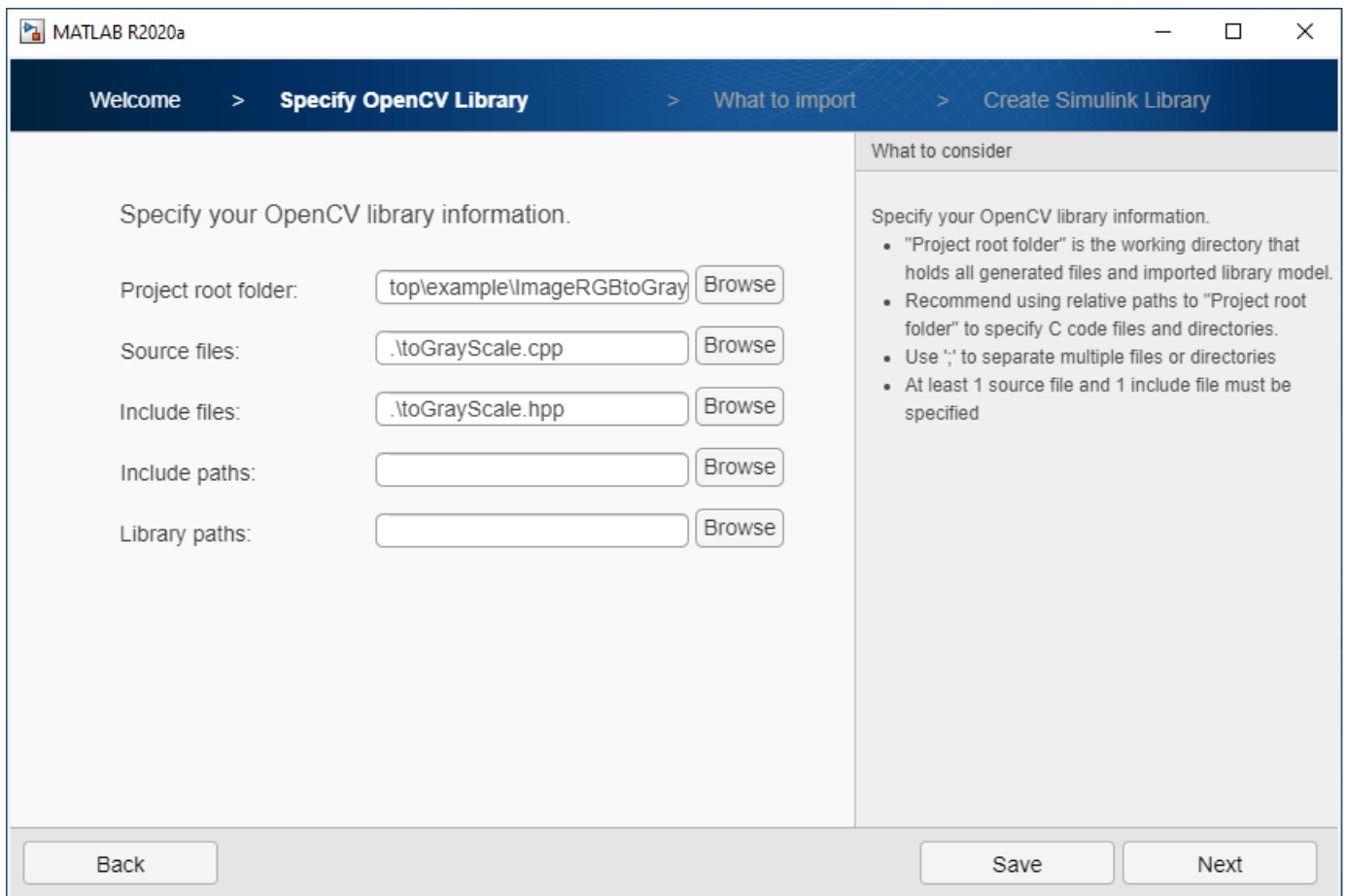
```
OpenCVSimulinkExamples;
```

Each subfolder contains all the supporting files required to run the example.

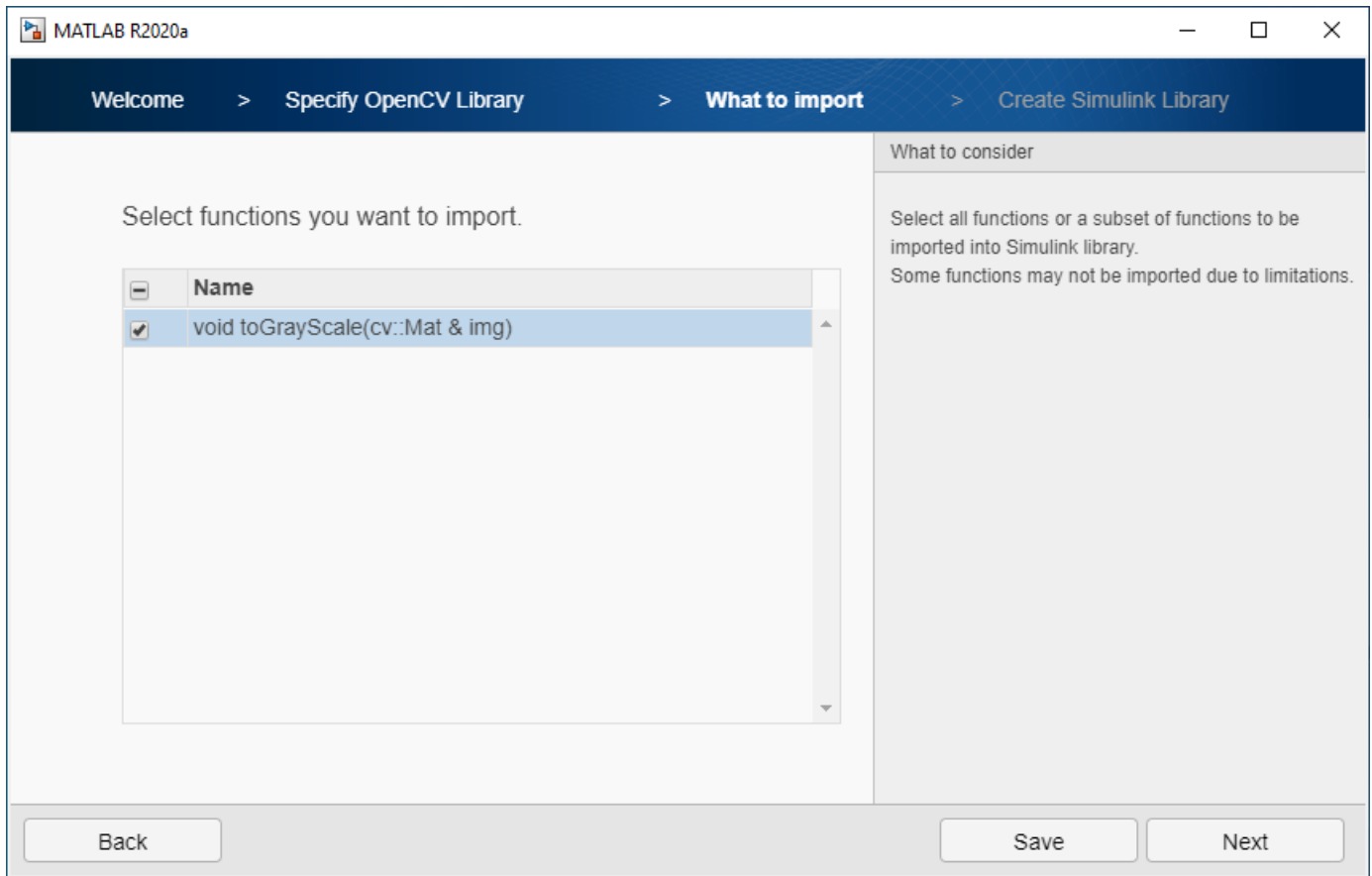
Before proceeding with these steps, ensure that you copy the example folder to a writable folder location and change your current working folder to `...example\ImageRGBtoGray`. All your output files are saved to this folder.

Step 1: Import OpenCV Function to Create a Simulink Library

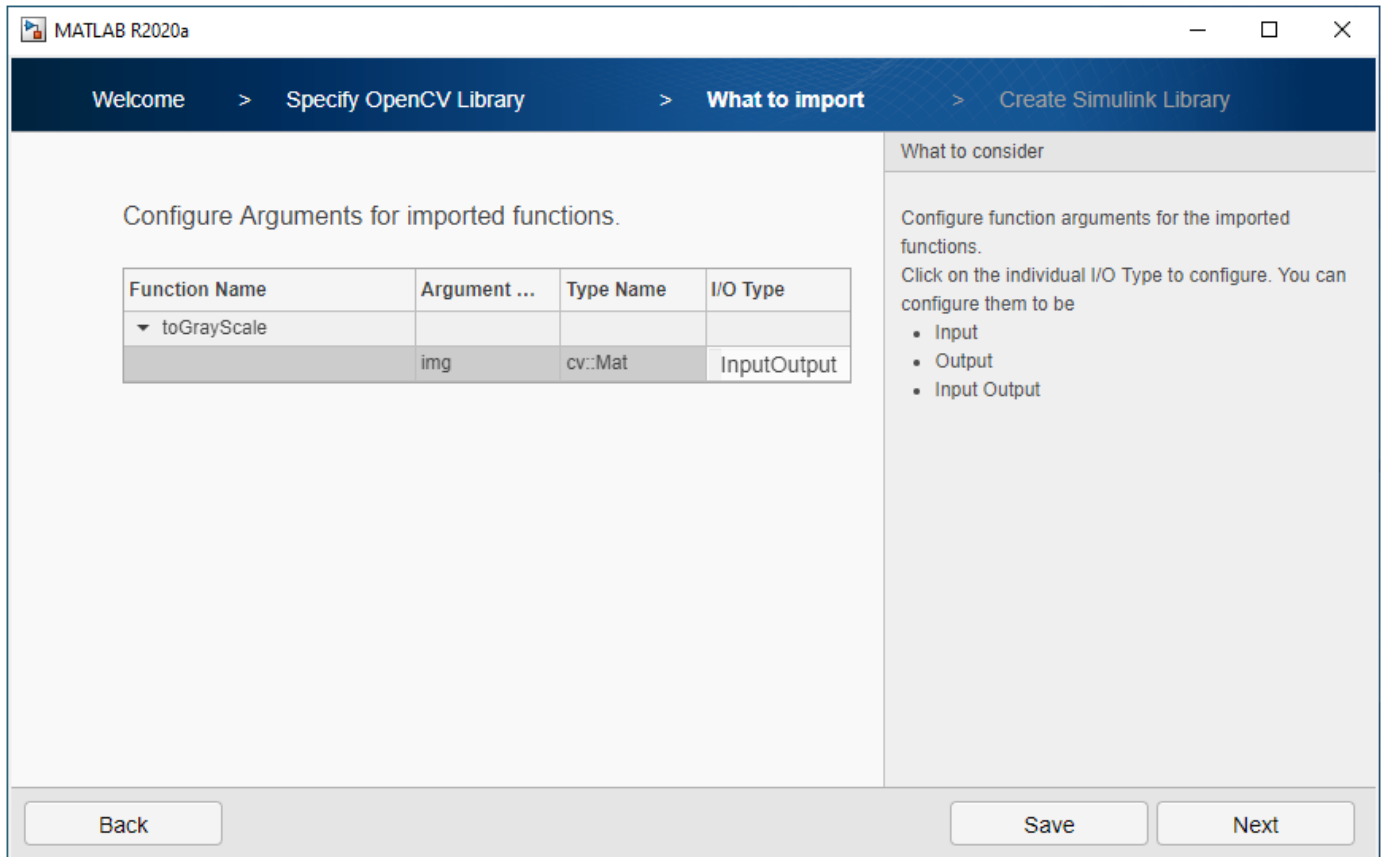
- 1 To start the **OpenCV Importer** app, click **Apps** on the MATLAB Toolstrip. In the Welcome page, specify the **Project name** as `RGBtoGRAY`. Make sure that the project name does not contain any spaces. Click **Next**.
- 2 In Specify OpenCV Library, specify these file locations, and then click **Next**.
 - **Project root folder:** Specify the path of your example folder. This path is the path to the writable project folder where you have saved your example files. All your output files are saved to this folder.
 - **Source files:** Specify the path of the `.cpp` file located inside your project folder as `toGrayScale.cpp`.
 - **Include files:** Specify the path of the `.hpp` header file located inside your project folder as `toGrayScale.hpp`.



- 3** Analyze your library to find the functions and types for import. Once the analysis is complete, click **Next**. Select the `toGrayScale` function and click **Next**.



- 4 From What to import, click the **I/O Type** for img as InputOutput, and then click **Next**.



- 5 In Create Simulink Library, verify the default values of OpenCV types. By default, **Create a single C-caller block for the OpenCV function** is selected to create a C Caller block with the subsystem. To create a Simulink library, click **Next**.

A Simulink library RGBtoGRAY_Lib is created from your OpenCV code. You can use any of these blocks in the library for model simulation. In this example, the subsystem `subsystem_slwrap_toGrayScale` is used.

Step 2: Use Generated Subsystem in Simulink Model

To use the generated subsystem `subsystem_slwrap_toGrayScale` with the Simulink model `toGrayScale.slx`:

- 1 In your MATLAB **Current Folder**, right-click the model `ToGrayScale.slx` and click **Open** from the context menu. Drag the generated subsystem to the model and connect the blocks.
- 2 Double-click the subsystem and specify these parameter values.


Parameter	Value	Description
Rows	384	Number of rows in the output image
Columns	512	Number of columns in the output image

Parameter	Value	Description
Channels	1	Number of channels in the output image
Underlying Type	uint8	Underlying data type of OpenCV Mat
is Image	on	Whether input is an image or a matrix

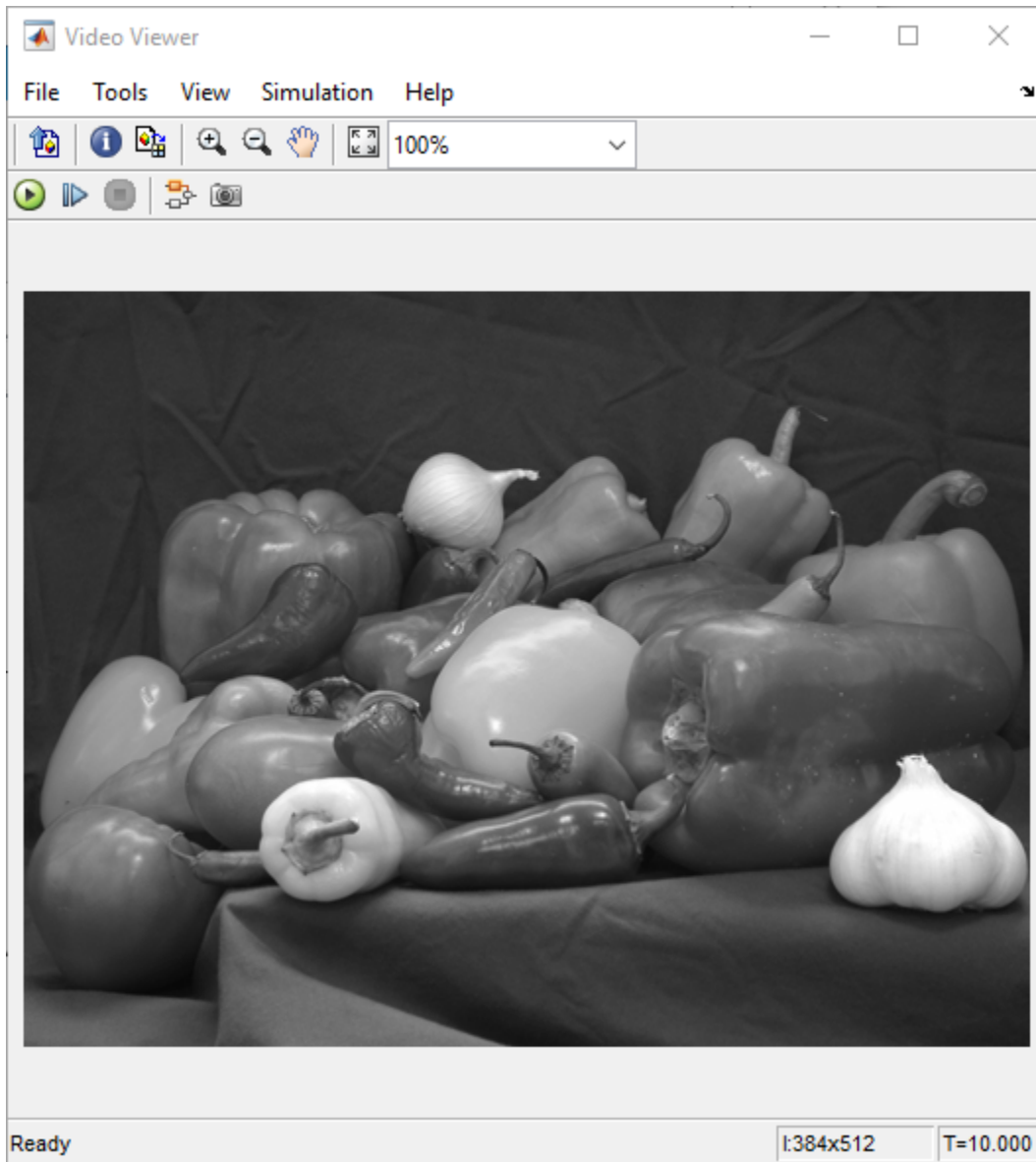
Click **Apply**, and then click **OK**.

Step 3: Simulate the RGB to Gray Convertor



On the Simulink Toolstrip, in the **Simulation** tab, click  to simulate the model. After the simulation is complete, the Video Viewer block displays the grayscale image of the input image peppers.png.





See Also

[FromOpenCV | ToOpenCV](#)

More About

- “Smile Detection by Using OpenCV Code in Simulink” on page 10-19
- “Draw Different Shapes by Using OpenCV Code in Simulink” on page 10-36

Draw Different Shapes by Using OpenCV Code in Simulink

This example shows how to draw different shapes on images.

First import an OpenCV function into Simulink by using the OpenCV Code Import Wizard on page 10-11. The wizard creates a Simulink library that contains a subsystem and a C Caller block for the specified OpenCV function. The subsystem is then used in a preconfigured Simulink model. This subsystem accepts coordinates of a specified shape. A defined shape is then displayed on a Video Viewer.

You learn how to:

- Import an OpenCV function into a Simulink library.
- Use blocks from a generated library in a Simulink model.

Required Products

- Computer Vision Toolbox Interface for OpenCV in Simulink
- Computer Vision Toolbox

Set Up Your C++ Compiler

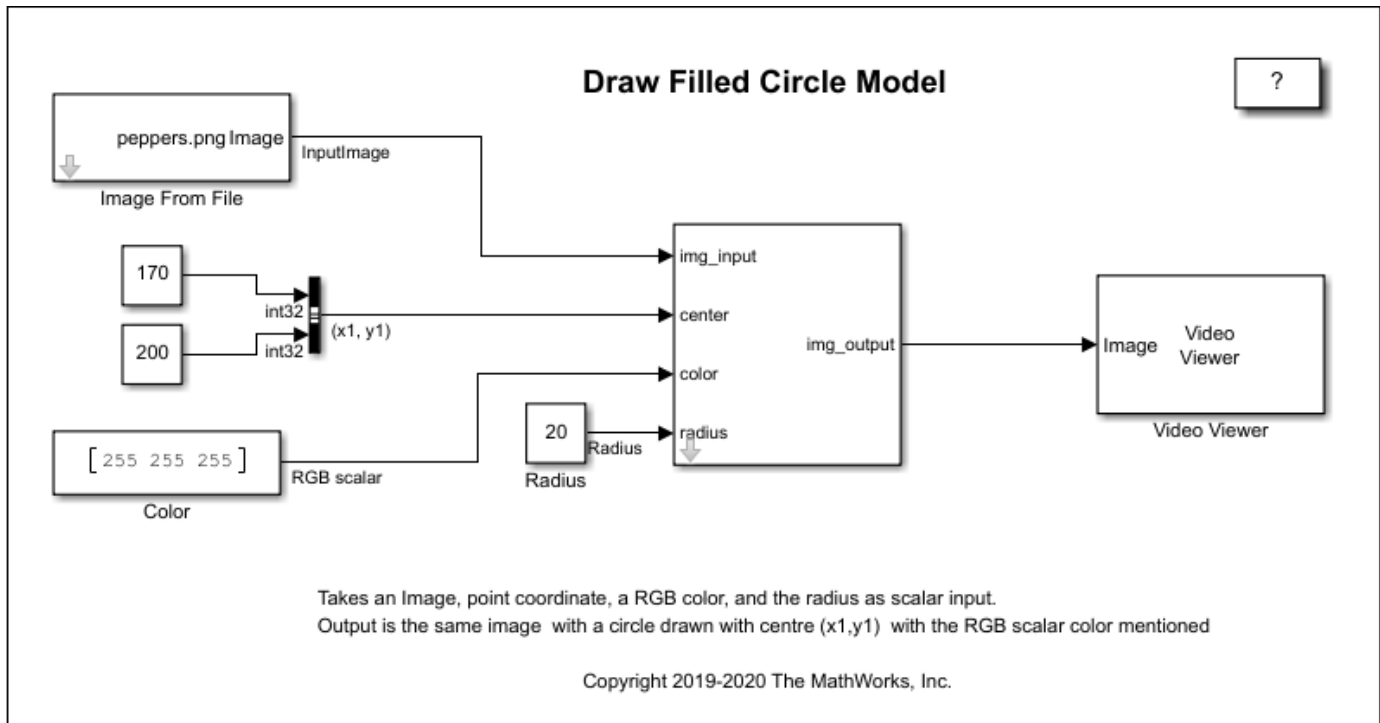
To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in *Compiler Used to Build OpenCV Libraries*. Configure the identified compiler by using the `mex -setup c++` command. For more information, see *Choose a C++ Compiler*.

Model Description

These Simulink models are available in the `DrawShapes` folder:

- `DrawAtom.slx`
- `DrawEllipse.slx`
- `DrawFilledCircle.slx`
- `DrawLine.slx`
- `DrawPolygon.slx`
- `DrawRook.slx`

This example uses the `DrawFilledCircle.slx` model. In this model, the `subsystem_slwrap_drawFilledCircle` subsystem resides in the `DrawCircle_Lib` library. You create the `subsystem_slwrap_drawFilledCircle` subsystem by using the **OpenCV Importer**. The subsystem accepts the x and y coordinates for the center of the circle and radius as input to the subsystem. The subsystem creates a circle on an input image from the Image From File block. The output is then displayed on a Video Viewer block.



Copy Example Folder to a Writable Location

To access the path to the example folder, at the MATLAB command line, enter:

```
OpenCVSimulinkExamples;
```

Each subfolder contains all the supporting files required to run the example.

Before proceeding with these steps, ensure that you copy the example folder to a writable folder location and change your current working folder to `...example\DrawShapes`. All your output files are saved to this folder.

Step 1: Import OpenCV Function to Create a Simulink Library

- 1 To start the **OpenCV Importer** app, click **Apps** on the MATLAB Toolstrip. The OpenCV import wizard opens to a Welcome page. Specify the **Project name** as `DrawCircle`. Make sure that the project name does not contain any spaces. Click **Next**.
- 2 In Specify OpenCV Library, specify these file locations, and then click **Next**.
 - **Project root folder:** Specify the path of your example folder. This path is the path to the writable project folder where you have saved your example files. All your output files are saved to this folder.
 - **Source files:** Specify the path of the `.cpp` file located inside your project folder as `opencvcode.cpp`.
 - **Include files:** Specify the path of the `.hpp` header file located inside your project folder as `opencvcode.hpp`.
- 3 Analyze your library to find the functions and types for import. Once the analysis is complete, click **Next**. From the listed functions, select the `drawFilledCircle` function and click **Next**.

- 4 From What to import, select the **I/O Type** for `img` as `InputOutput` and other arguments as `Input`. Click **Next**.
- 5 In Create Simulink Library, verify the default values of OpenCV types. By default, **Create a single C-caller block for the OpenCV function** is selected to create a C Caller block with the subsystem. To create a Simulink library, click **Next**.


A Simulink library `DrawCircle_Lib` is created from your OpenCV code. You can use any of these blocks for model simulation. In this example, the subsystem `subsystem_slwrap_drawFilledCircle` is used.

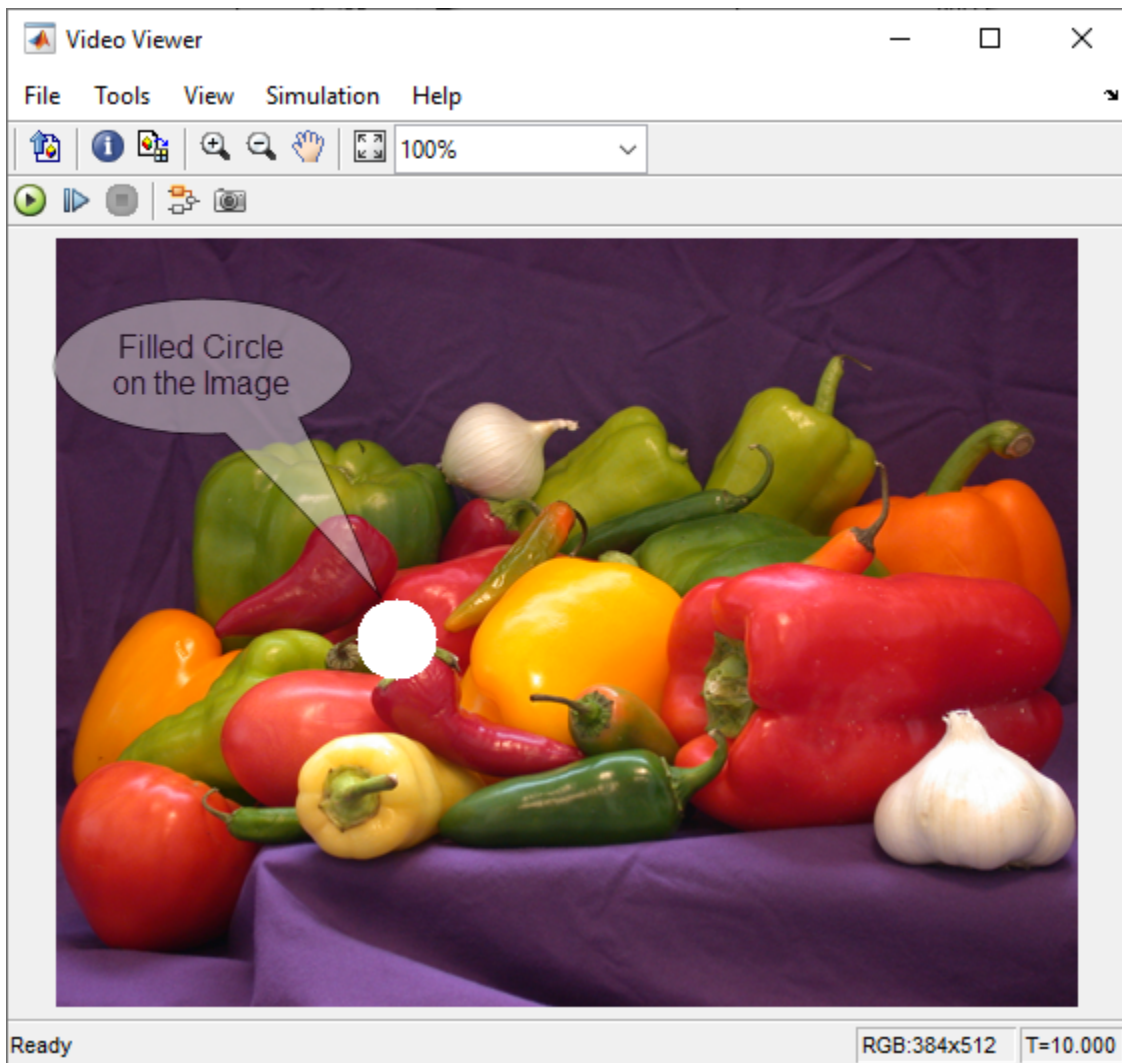
Step 2: Use Generated Subsystem in Simulink Model

To use the generated subsystem `subsystem_slwrap_drawFilledCircle` with the Simulink model `DrawFilledCircle.slx`:

- 1 In your MATLAB **Current Folder**, right-click the model `DrawFilledCircle.slx` and click **Open** from the context menu. Drag the generated subsystem to the model and connect the blocks.
- 2 Double-click the subsystem and verify the parameter values.



On the Simulink Toolstrip, in the **Simulation** tab, click  to simulate the model. After the simulation is complete, the Video Viewer block displays the filled circle on the input image `peppers.png`.



Draw Atom on Image by Using C Caller Block

This example shows how to use a C Caller block in a Simulink model to draw an atom on an image.

- 1 Import `drawEllipse` and `drawFilledCircle` OpenCV functions into Simulink by using the OpenCV Code Import Wizard on page 10-11. During import, select the **I/O Type** for `drawEllipse` and `drawFilledCircle`, as shown in this graphic.

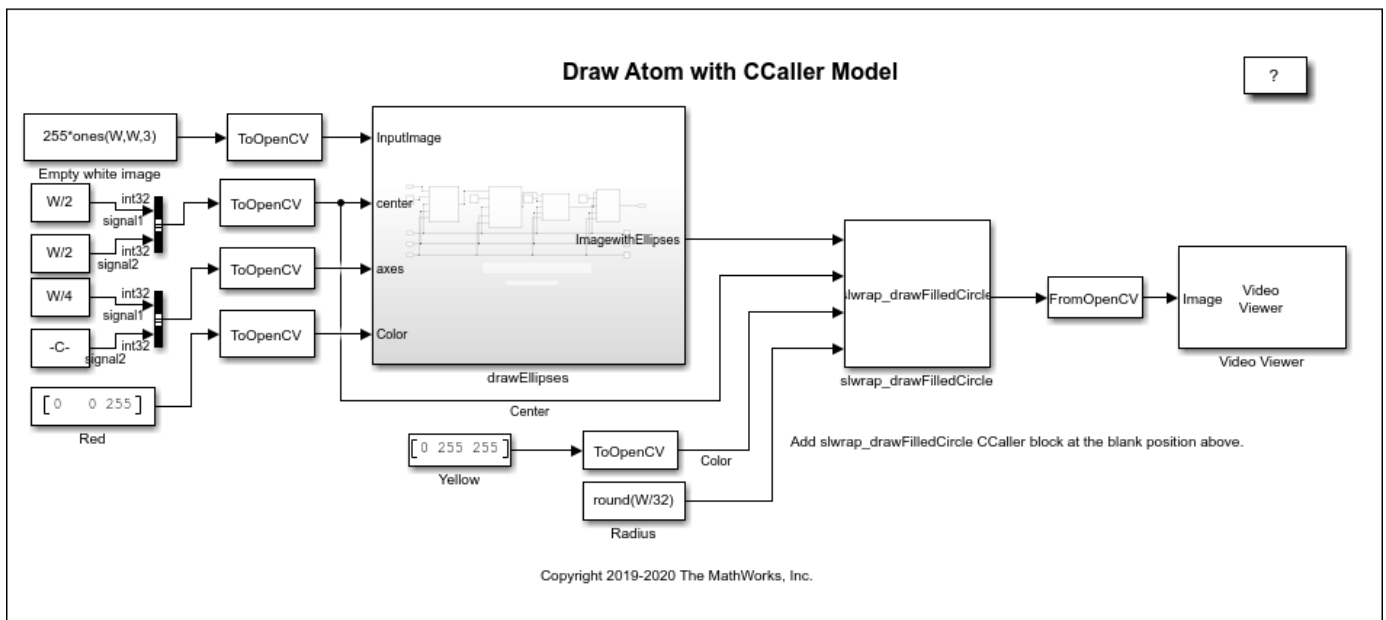
Configure Arguments for imported functions.

Function Name	Argument Name	Type Name	I/O Type
▼ drawEllipse			
	img	cv::Mat	InputOutput
	angle	double	Input
	center	cv::Point2i	Input
	axesSize	cv::Size	Input
	color	cv::Scalar	Input
▼ drawFilledCircle			
	img	cv::Mat	InputOutput
	center	cv::Point2i	Input
	color	cv::Scalar	Input
	radius	int	Input

- Once you import the functions, the DrawCircle_Lib library is created. This Simulink library contains subsystems and the C Caller blocks required to draw an atom on an image.


Open the model DrawAtomCcaller.slx. Drag the slwrap_drawEllipse C Caller block from the Simulink library DrawCircle_Lib to drawEllipses subsystem in the model. Create three copies of the C Caller block, and then place these blocks at the four blank positions inside the drawEllipses subsystem.

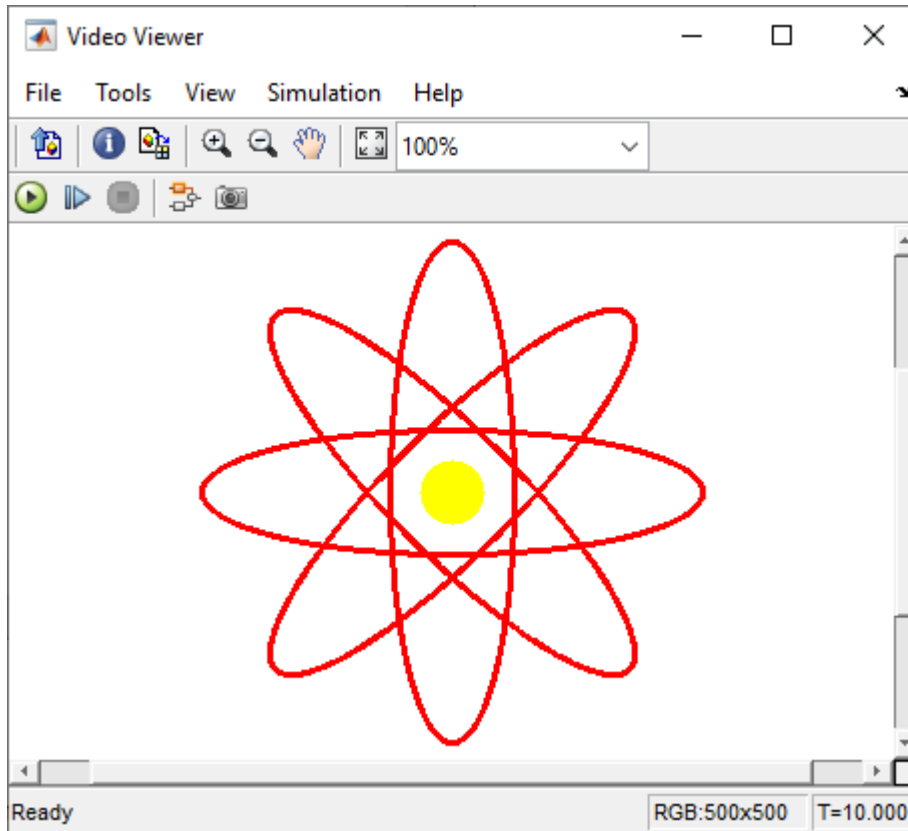
In the model, drag the slwrap_drawFilledCircle C Caller block from the Simulink library DrawCircle_Lib and place the block at the blank position.



3



On the Simulink Toolstrip, in the **Simulation** tab, click  to simulate the model. After the simulation is complete, the Video Viewer block displays the atom on a white input image.



See Also

[FromOpenCV | ToOpenCV](#)

More About

- “Smile Detection by Using OpenCV Code in Simulink” on page 10-19
- “Convert RGB Image to Grayscale Image by Using OpenCV Importer” on page 10-29

Input, Output, and Conversions

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

- “Export to Video Files” on page 11-2
- “Import from Video Files” on page 11-4
- “Batch Process Image Files” on page 11-6
- “Convert R'G'B' to Intensity Images” on page 11-7
- “Process Multidimensional Color Video Signals” on page 11-10
- “Video Formats” on page 11-12
- “Image Formats” on page 11-13

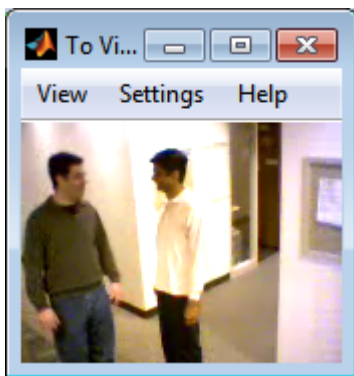
Export to Video Files

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

Configuration Parameters

Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings** > **Model Settings**. Set the **Solver** parameters as follows:

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

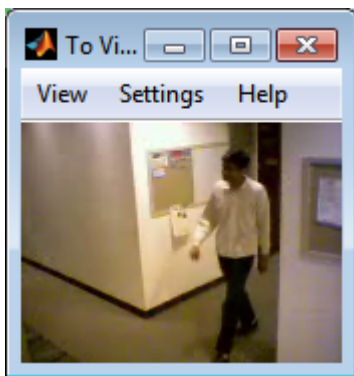
Import from Video Files

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 MATLAB command 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 11-2 example you can manipulate your video stream and export it to a multimedia file.

For more information on the blocks used in this example, see the From Multimedia File and To Video Display block reference pages.

Setting Block Parameters for this Example

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

Block	Parameter
From Multimedia File	<p>Use the From Multimedia File block to import the multimedia file into the model:</p> <ul style="list-style-type: none"> • If you do not have your own multimedia file, use the default <code>vipmen.avi</code> file, for the File name parameter. • If the multimedia file is on your MATLAB path, enter the filename for the File name parameter. • If the file is not on your MATLAB path, use the Browse button to locate the multimedia file. • Set the Image signal parameter to <code>Separate color signals</code>. <p>By default, the Number of times to play file parameter is set to <code>inf</code>. The model continues to play the file until the simulation stops.</p>

Block	Parameter
To Video Display	Use the To Video Display block to view the multimedia file. <ul style="list-style-type: none"><li data-bbox="594 352 1170 384">• Image signal: Separate color signals Set this parameter from the Settings menu of the display viewer.

Configuration Parameters

Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings** > **Model Settings**. Set the **Solver** parameters 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_lm4_*.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” section for the Image Processing Toolbox™.

Configuration Parameters

Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings** > **Model Settings**. Set the **Solver** parameters as follows:

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

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

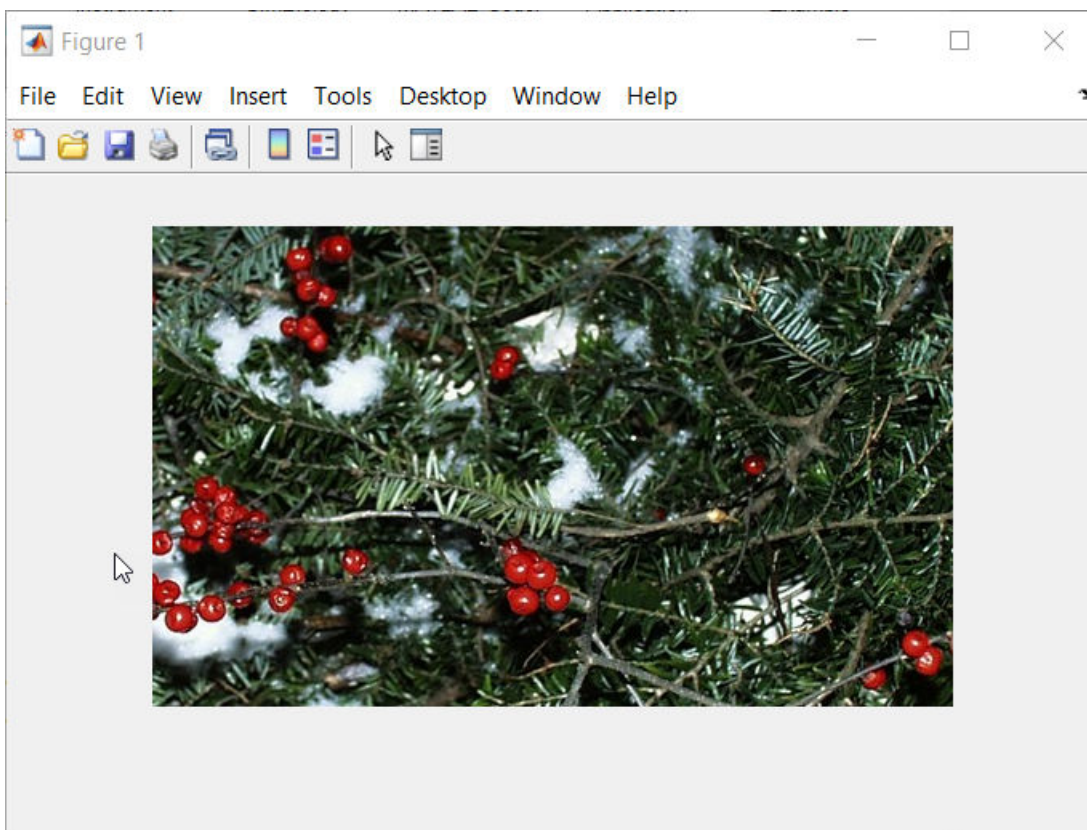
- 1 Define an R'G'B' image in the MATLAB workspace. To read in an R'G'B' image from a JPG file, at the MATLAB command prompt, type

```
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

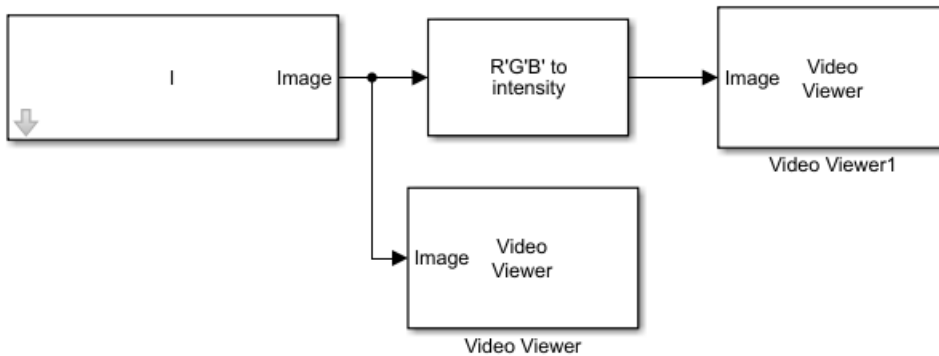
```
imshow(I)
```



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

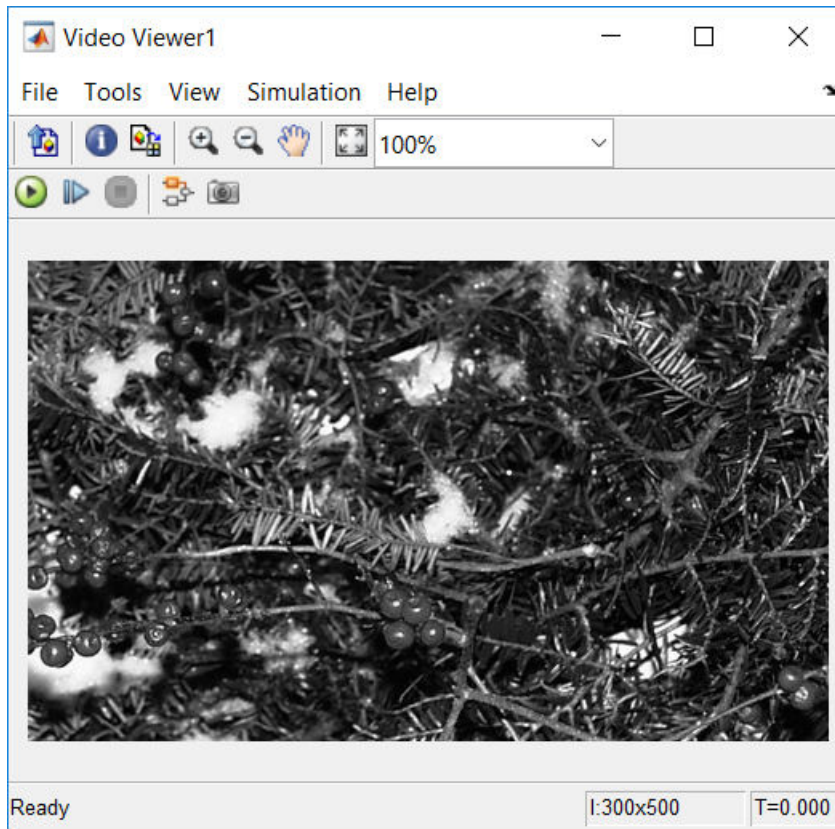
Block	Library	Number of Blocks
Image From Workspace	Computer Vision Toolbox > Sources	1
Color Space Conversion	Computer Vision Toolbox > Conversions	1
Video Viewer	Computer Vision Toolbox > Sinks	2

- 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 Settings** from the **Setup** menu on the **Modeling** tab. 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.



Process Multidimensional Color Video Signals

The Computer Vision 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



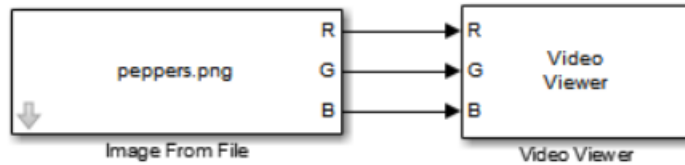
You can choose to process the image as a multidimensional array by setting the **Image signal** parameter to `One multidimensional signal` in the Image From File block dialog box.

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.

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 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 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 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 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”.

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”.

RGB Images

RGB images are also known as a true-color images. With Computer Vision 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 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”.

Display and Graphics



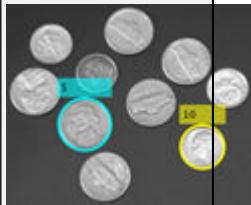
- “Choose Function to Visualize Detected Objects” on page 12-2
- “Display, Stream, and Preview Videos” on page 12-5
- “Draw Shapes and Lines” on page 12-7



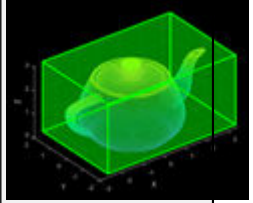
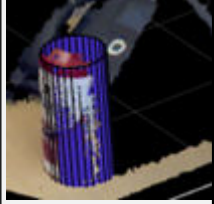
Choose Function to Visualize Detected Objects

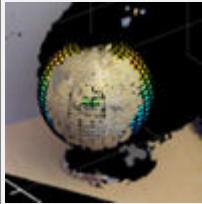

Computer Vision Toolbox offers several functions to visualize detected objects by inserting or overlaying shapes on image, video, and point cloud data.

The `insert`-related functions draw shapes and text by fusing them with image and video data. The `showShape` function uses MATLAB graphics to overlay shapes and text on top of image, video, and point cloud data and is rendered in a MATLAB axes.

This table compares the visualization functions on the basis of their support for image, video, and point cloud data.

Function	Images	Video	Point Clouds	Code Generation	Example
<code>insertShape</code>	Yes	Yes	No	Yes	 Insert Shapes on Image
<code>insertText</code>	Yes	Yes	No	Yes	 Insert Text on Image
<code>insertObject Annotation</code>	Yes	Yes	No	Yes	 Annotate an Image

Function	Images	Video	Point Clouds	Code Generation	Example
insertMarker	Yes	Yes	No	Yes	 <p>Insert Markers on Image</p>
insertObject Mask	Yes	Yes	No	Yes	 <p>Insert Multicolor Masks on Image</p>
showShape	Yes	Yes	Yes	No	 <p>Show Cuboid on Detected Object in Point Cloud</p>
plot object function of the cylinderModel	No	No	Yes	No	 <p>Fit Cylinder Shape on Point Cloud</p>

Function	Images	Video	Point Clouds	Code Generation	Example
plot object function of the sphereModel	No	No	Yes	No	 Fit Sphere Shape on Point Cloud
plot object function of the planeModel	No	No	Yes	No	 Fit Plane Shape on Point Cloud

See Also

Objects

cylinderModel | pcplayer | planeModel | sphereModel | vision.DeployableVideoPlayer | vision.VideoPlayer

Functions

imshow | insertMarker | insertObjectAnnotation | insertObjectMask | insertShape | insertText | pcshow | showShape

Display, Stream, and Preview Videos

In this section...

“View Streaming Video in MATLAB” on page 12-5

“Preview Video in MATLAB” on page 12-5

“View Video in Simulink” on page 12-5

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 20-3.

Example 12.1. 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** .

The video appears in the player window.

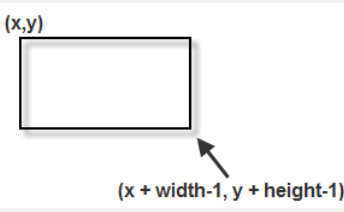
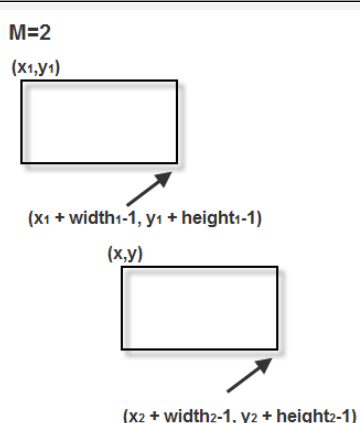
- 6** You can use multiple `implay` players to display different Simulink signals.

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

Draw Shapes and Lines

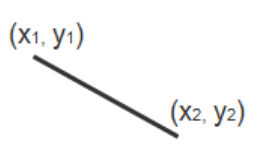
When you specify the type of shape to draw, you must also specify its location on the image. The table shows the format for the points input for the different shapes.

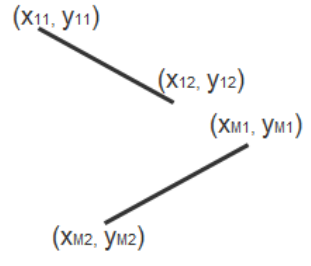
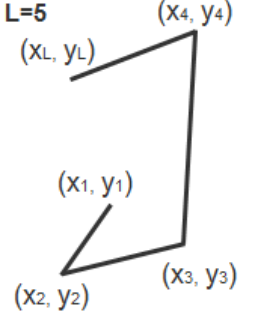
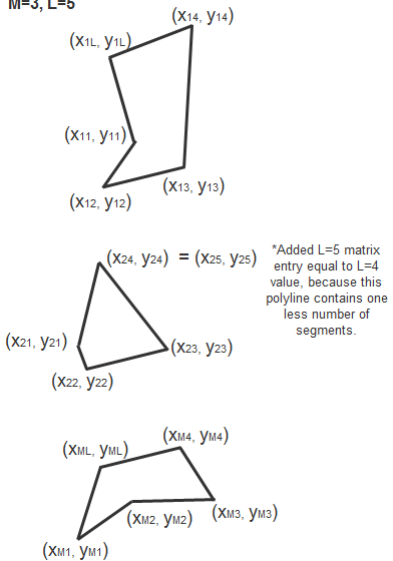
Rectangle

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

Line and Polyline

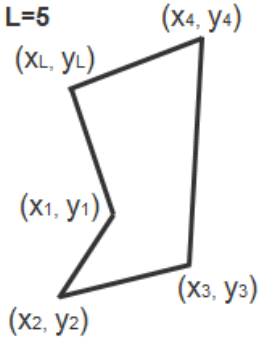
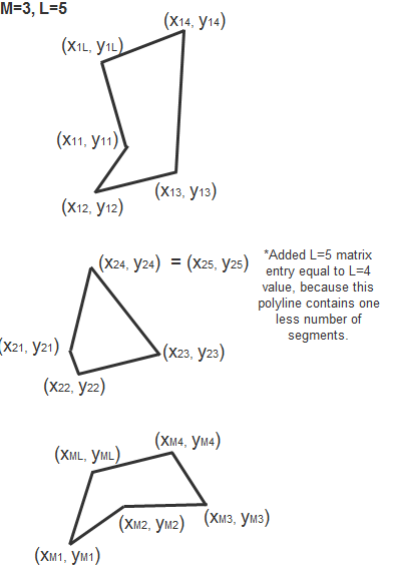
You can draw one or more lines, and one or more polylines. A polyline contains a series of connected line segments.

Shape	PTS input	Drawn Shape
Single Line	<p>Four-element row vector $[x_1 \ y_1 \ x_2 \ y_2]$ where</p> <ul style="list-style-type: none"> x_1 and y_1 are the coordinates of the beginning of the line. x_2 and y_2 are the coordinates of the end of the line. 	

Shape	PTS input	Drawn Shape
<p>M Lines</p>	<p>M-by-4 matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} \\ x_{21} & y_{21} & x_{22} & y_{22} \\ \vdots & \vdots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different line and is of the same form as the vector for a single line.</p>	
<p>Single Polyline with $(L-1)$ Segments</p>	<p>Vector of size $2L$, where L is the number of vertices, with format, $[x_1, y_1, x_2, y_2, \dots, x_L, y_L]$.</p> <ul style="list-style-type: none"> x_1 and y_1 are the coordinates of the beginning of the first line segment. x_2 and y_2 are the coordinates of the end of the first line segment and the beginning of the second line segment. x_L and y_L are the coordinates of the end of the $(L-1)^{th}$ line segment. <p>The polyline always contains $(L-1)$ number of segments because the first and last vertex points do not connect. The block produces an error message when the number of rows is less than two or not a multiple of two.</p>	<p>$L=5$</p> 
<p>M Polylines with $(L-1)$ Segments</p>	<p>$2L$-by-N matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \dots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \dots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \dots & x_{ML} & y_{ML} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different polyline and is of the same form as the vector for a single polyline. When you require one polyline to contain less than $(L-1)$ number of segments, fill the matrix by repeating the coordinates of the last vertex.</p> <p>The block produces an error message if the number of rows is less than two or not a multiple of two.</p>	<p>$M=3, L=5$</p>  <p>*Added $L=5$ matrix entry equal to $L=4$ value, because this polyline contains one less number of segments.</p>

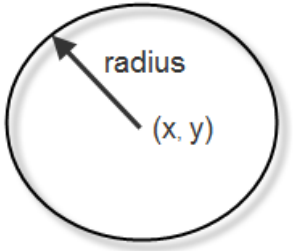
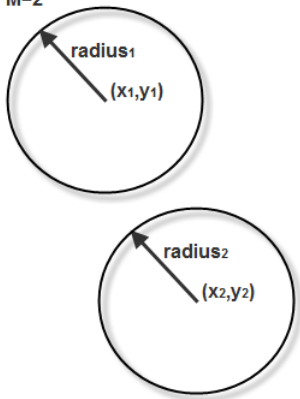
Polygon

You can draw one or more polygons.

Shape	PTS input	Drawn Shape
<p>Single Polygon with L line segments</p>	<p>Row vector of size $2L$, where L is the number of vertices, with format, $[x_1 \ y_1 \ x_2 \ y_2 \ \dots \ x_L \ y_L]$ where</p> <ul style="list-style-type: none"> x_1 and y_1 are the coordinates of the beginning of the first line segment. x_2 and y_2 are the coordinates of the end of the first line segment and the beginning of the second line segment. x_L and y_L are the coordinates of the end of the $(L-1)^{th}$ line segment and the beginning of the L^{th} line segment. <p>The block connects $[x_1 \ y_1]$ to $[x_L \ y_L]$ to complete the polygon. The block produces an error if the number of rows is negative or not a multiple of two.</p>	<p>$L=5$</p> 
<p>M Polygons with the largest number of line segments in any line being L</p>	<p>M-by-$2L$ matrix</p> $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \dots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \dots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \dots & x_{ML} & y_{ML} \end{bmatrix}$ <p>where each row of the matrix corresponds to a different polygon and is of the same form as the vector for a single polygon. If some polygons are shorter than others, repeat the ending coordinates to fill the polygon matrix.</p> <p>The block produces an error message if the number of rows is less than two or is not a multiple of two.</p>	<p>$M=3, L=5$</p>  <p>*Added $L=5$ matrix entry equal to $L=4$ value, because this polyline contains one less number of segments.</p>

Circle

You can draw one or more circles.

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

See Also

`insertMarker` | `insertObjectAnnotation` | `insertShape` | `insertText`

Registration and Stereo Vision

- “Fisheye Calibration Basics” on page 13-2
- “Single Camera Calibrator App” on page 13-8
- “Stereo Camera Calibrator App” on page 13-25
- “What Is Camera Calibration?” on page 13-39
- “Structure from Motion” on page 13-45

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

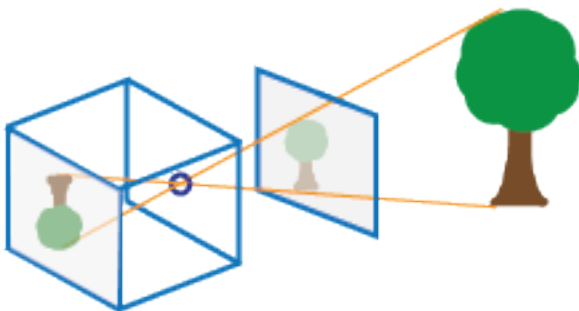


Fisheye image

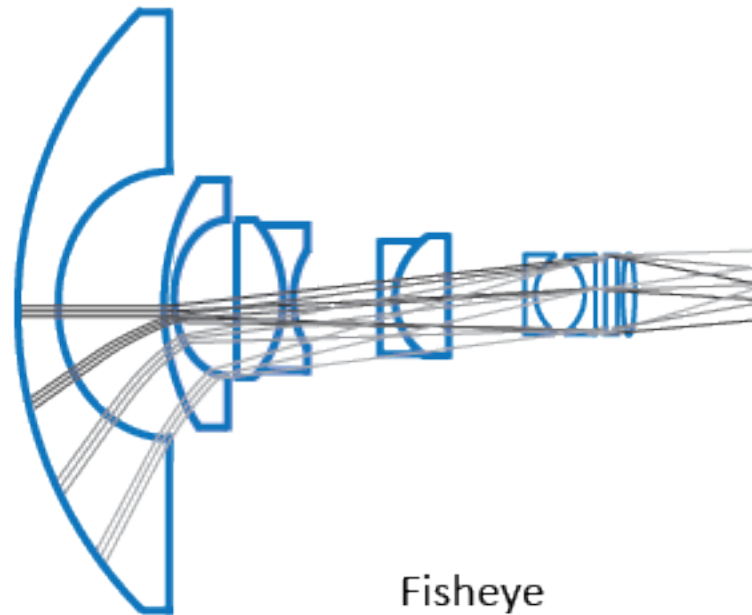


Undistorted fisheye image

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



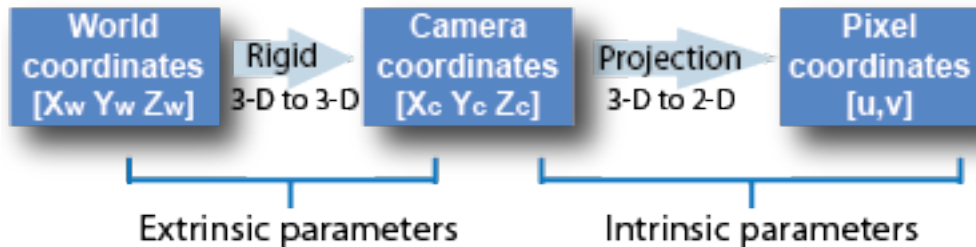
Pinhole



Fisheye

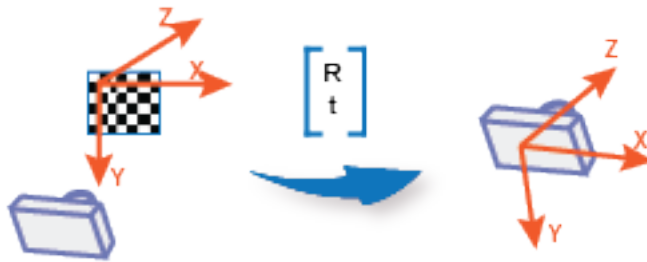
Fisheye Camera Model

The Computer Vision Toolbox calibration algorithm uses the fisheye camera model proposed by Scaramuzza [1] on page 13-6. 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:

$$\underbrace{\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix}}_{\text{Camera points}} = \underbrace{R}_{\text{Rotation}} \underbrace{\begin{pmatrix} X_w \\ Y_w \\ Z_w \end{pmatrix}}_{\text{World points}} + \underbrace{T}_{\text{Translation}}$$

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.

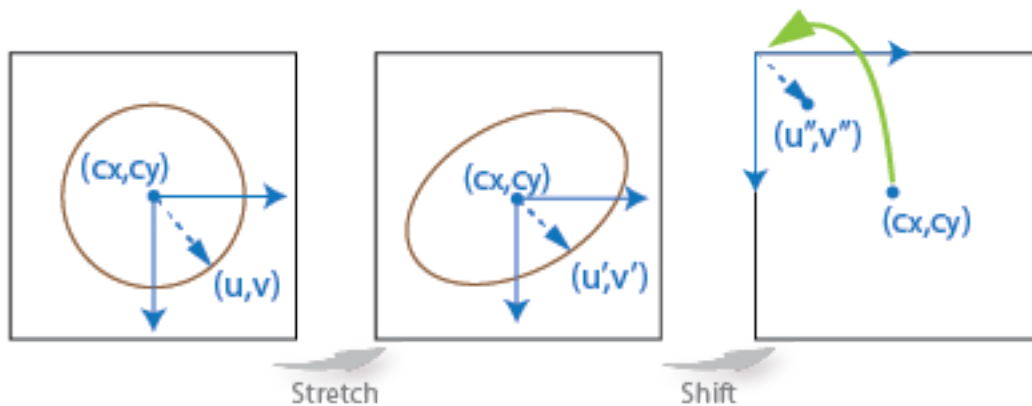
The following equation maps an image point into its corresponding 3-D vector.

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

- (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 coefficients 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 checkerboard 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);
```

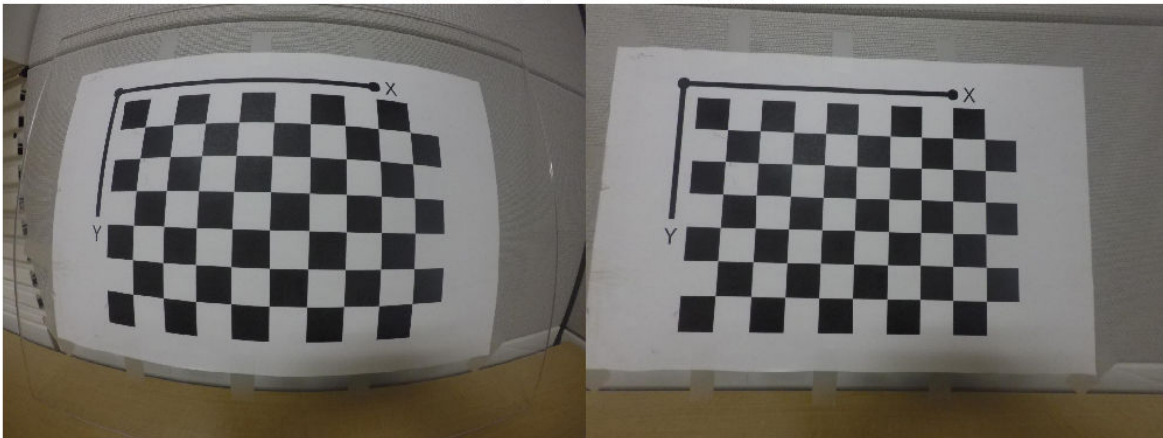
Estimate the fisheye camera calibration parameters based on the image and world points. Use the first image to get the image size.

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

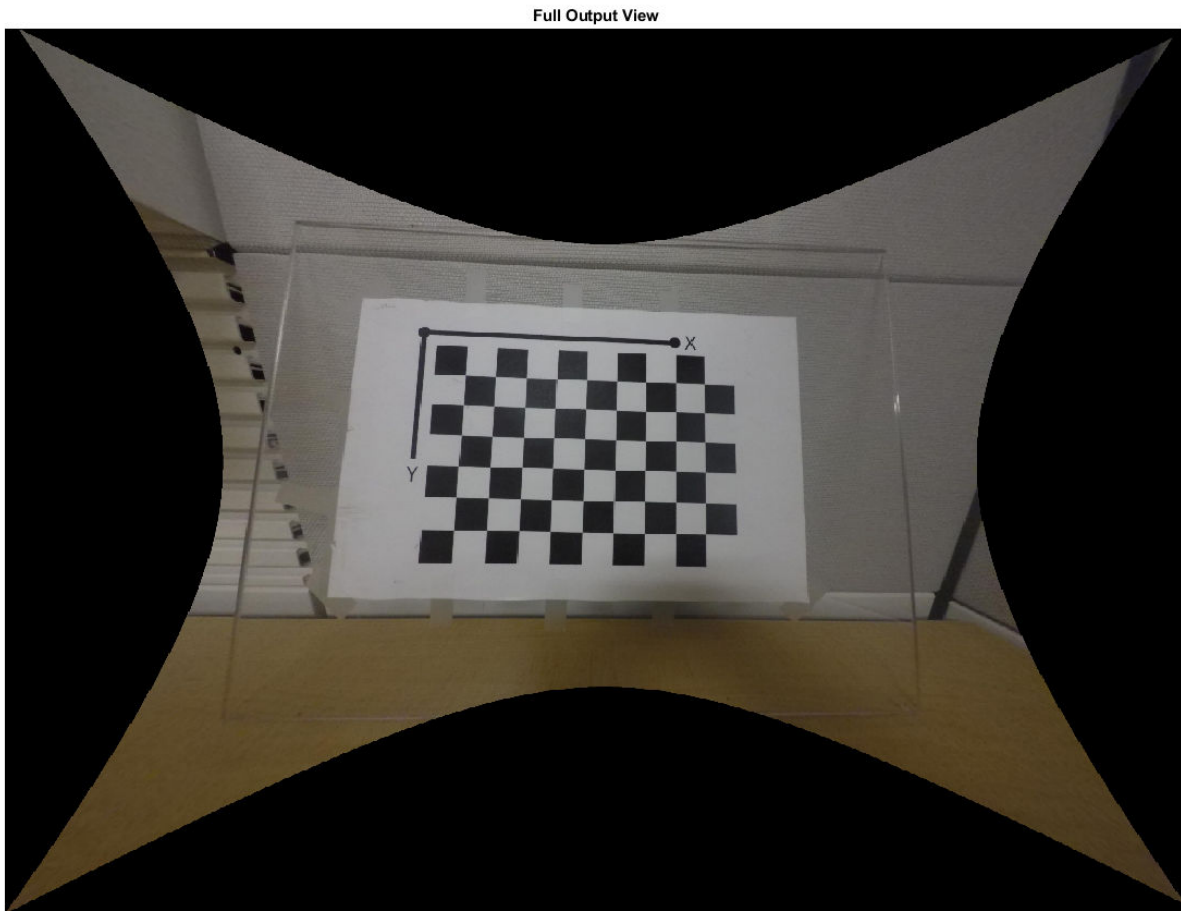
Remove lens distortion from the first image I and display the results.

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

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



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

Functions

[estimateFisheyeParameters](#) | [undistortFisheyeImage](#) | [undistortFisheyePoints](#)

Objects

[fisheyeCalibrationErrors](#) | [fisheyeIntrinsics](#) | [fisheyeIntrinsicsEstimationErrors](#) | [fisheyeParameters](#)

Related Examples

- “Configure Monocular Fisheye Camera” (Automated Driving Toolbox)
- “Calibrate a Monocular Camera” (Automated Driving Toolbox)
- “Structure From Motion From Two Views” on page 1-37
- “Structure From Motion From Multiple Views” on page 1-70
- “Configure Monocular Fisheye Camera” (Automated Driving Toolbox)

Single Camera Calibrator App

In this section...

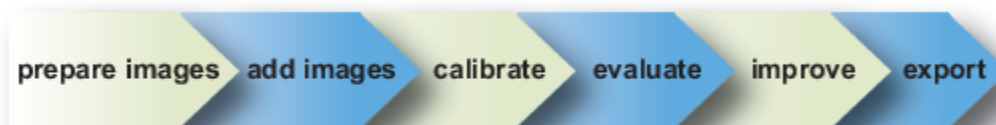
“Camera Calibrator Overview” on page 13-8
 “Single Camera Calibration” on page 13-8
 “Open the Camera Calibrator” on page 13-9
 “Prepare the Pattern, Camera, and Images” on page 13-9
 “Add Images and Select Camera Model” on page 13-12
 “Calibrate” on page 13-15
 “Evaluate Calibration Results” on page 13-17
 “Improve Calibration” on page 13-20
 “Export Camera Parameters” on page 13-23

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 the MATLAB workspace. For a list of functions, see “Single and Stereo 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 and Stereo 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 and Stereo Camera Calibration” for the list of functions.

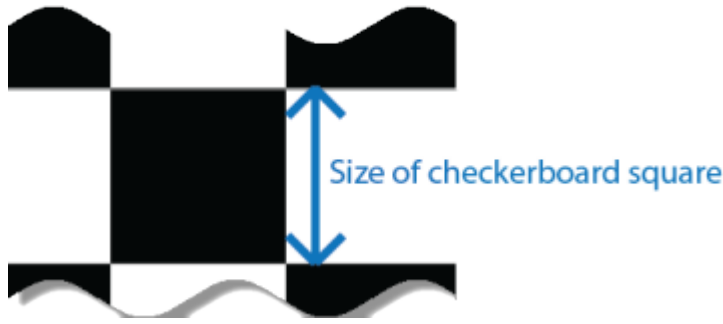
You can print (from MATLAB) and use the checkerboard pattern provided.

Tip Use a checkerboard that contains an even number of squares along one edge and an odd number of squares along the other edge. Using a non-square 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 and the origin. The calibrator assigns the longer side to be the x-direction. A square pattern can produce unexpected results for camera extrinsics.



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.



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

Camera Setup

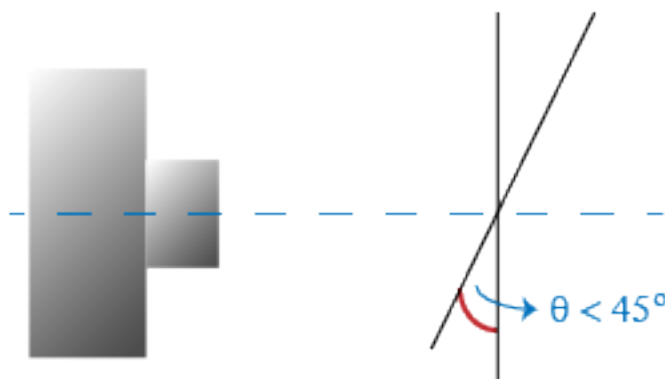
To calibrate your camera, follow these rules:

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

Capture Images

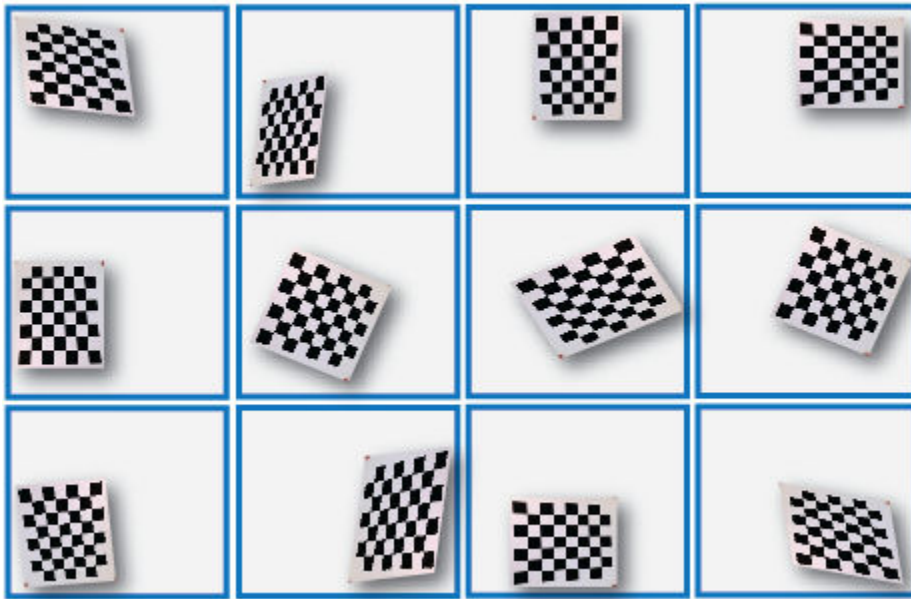
For better results, use at least 10 to 20 images of the calibration pattern. The calibrator requires at least three images. Use uncompressed images or images in lossless compression formats such as PNG. For greater calibration accuracy:

- 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.

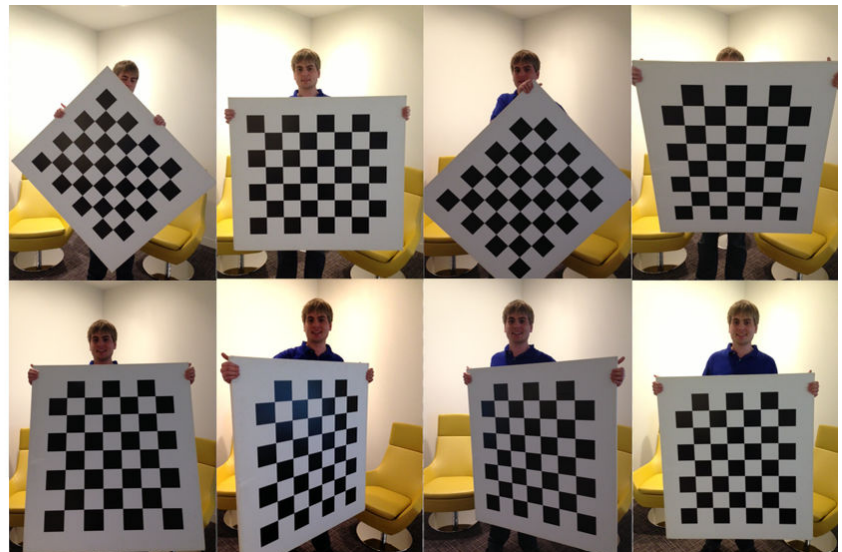
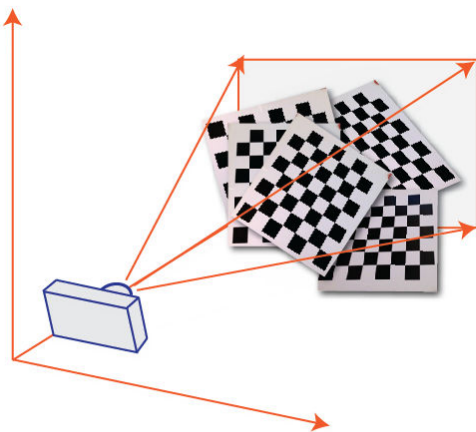


- Do not modify the images, (for example, do not crop them).

- 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. 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. 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.

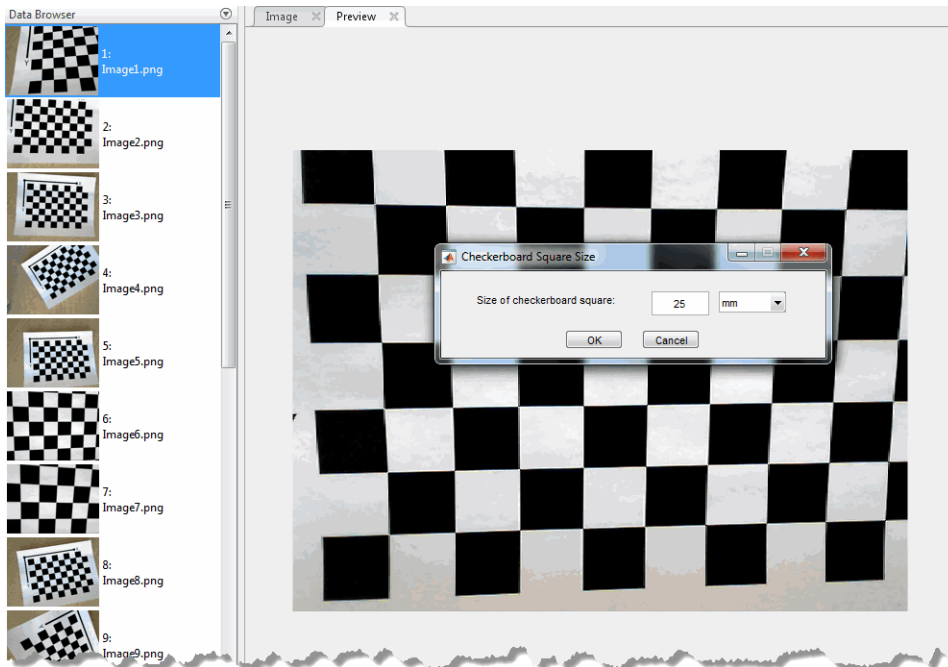
- 3 Enter a location for the acquired image files in the **Save Location** box by typing the path to the folder or using the **Browse** button. You must have permission to write to the folder you select.
- 4 Set the capture parameters.
 - To set the number of seconds between image captures, use the **Capture 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.

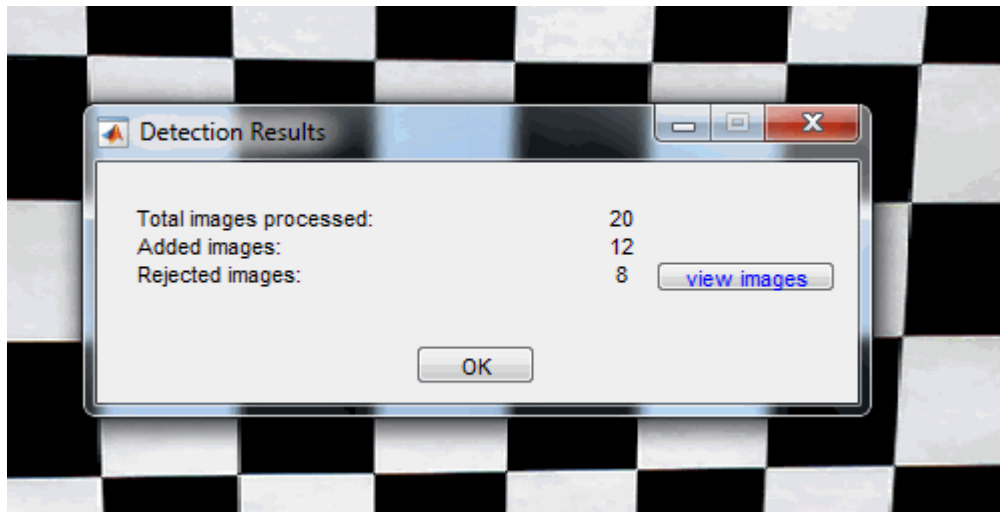
- 5 The Preview window shows the live images streamed as RGB data. After you adjust any device properties and capture settings, use the Preview window as a guide to line up the camera to acquire the checkerboard pattern image you want to capture.
- 6 Click the **Capture** button. The number of images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



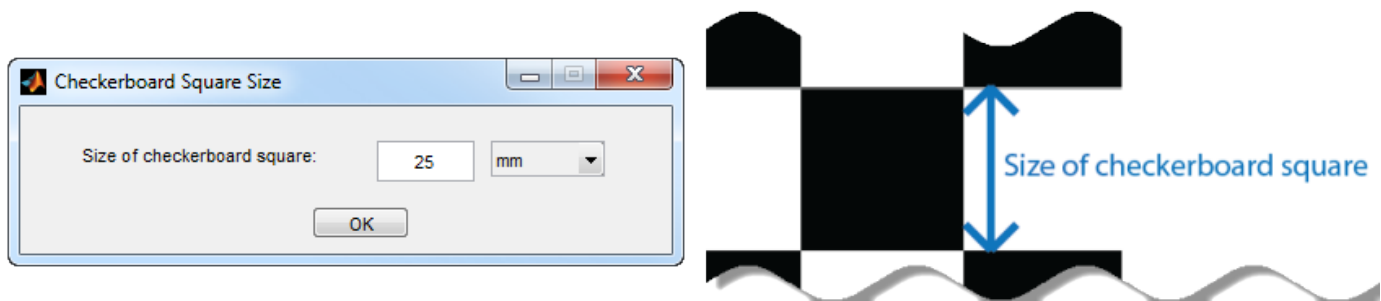
The detection results are then calculated and displayed. For example:



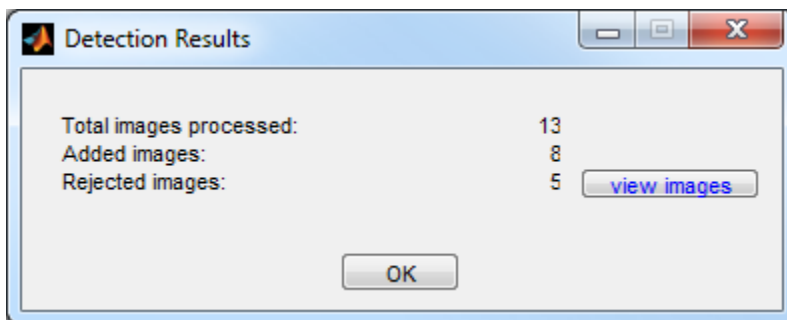
- 7 Click **OK** to dismiss the Detection Results dialog box.
- 8 When you have finished acquiring live images, click **Close Image Capture** to close the **Camera** tab.

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.



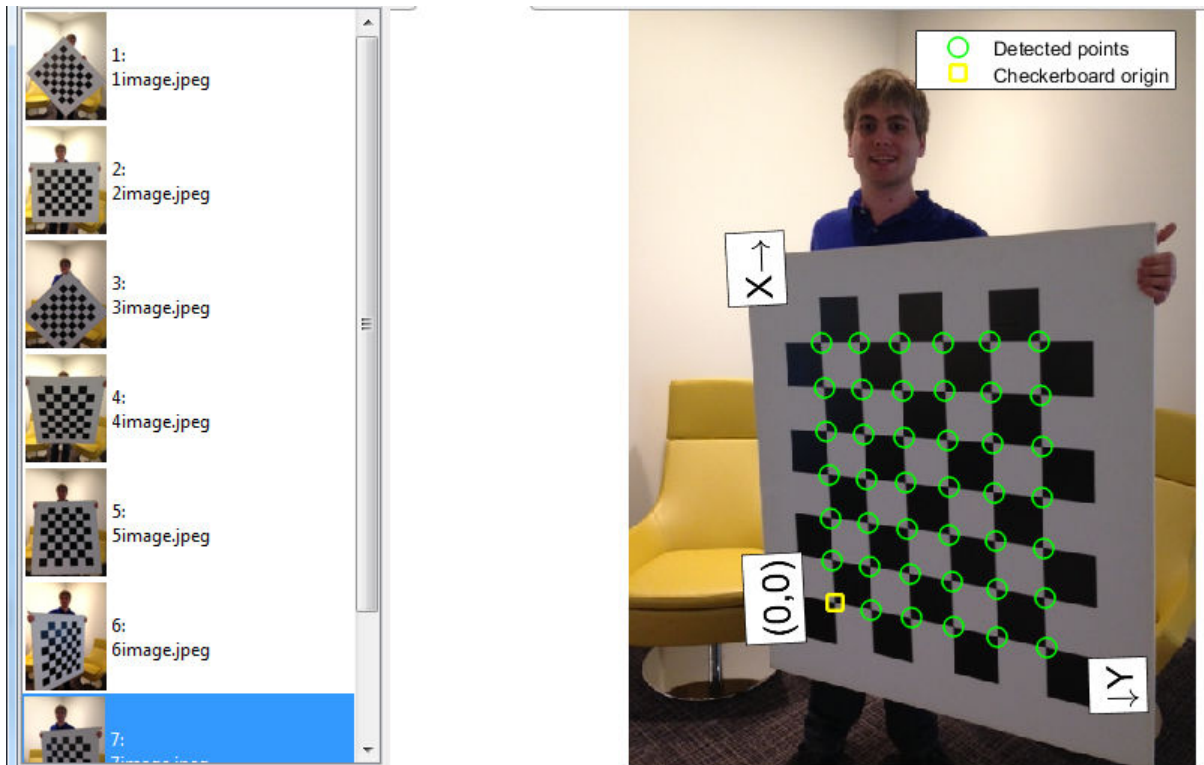
The calibrator attempts to detect a checkerboard in each of the added images, displaying an Analyzing Images progress bar window, indicating detection progress. If any of the images are rejected, the Detection Results dialog box appears, which contains diagnostic information. The results indicate how 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 $\mathbf{0}$ 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.

For details about the fisheye camera model calibration algorithm, see “Fisheye Calibration Basics” on page 13-2.

Calibration Algorithm

For fisheye camera model calibration, see “Fisheye Calibration Basics” on page 13-2.

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

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

- (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 $\mathbf{0}$. 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.

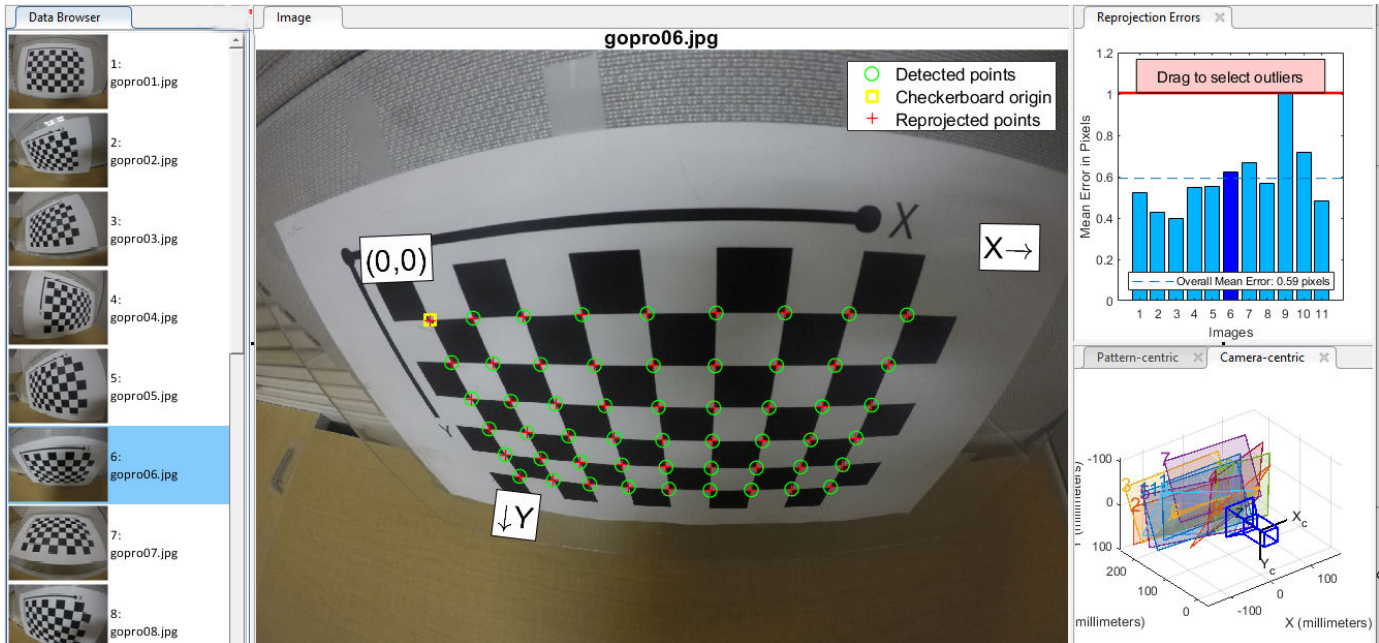
The camera calibration algorithm estimates the values of the intrinsic parameters, the extrinsic parameters, and the distortion coefficients. Camera calibration involves these steps:

- 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]

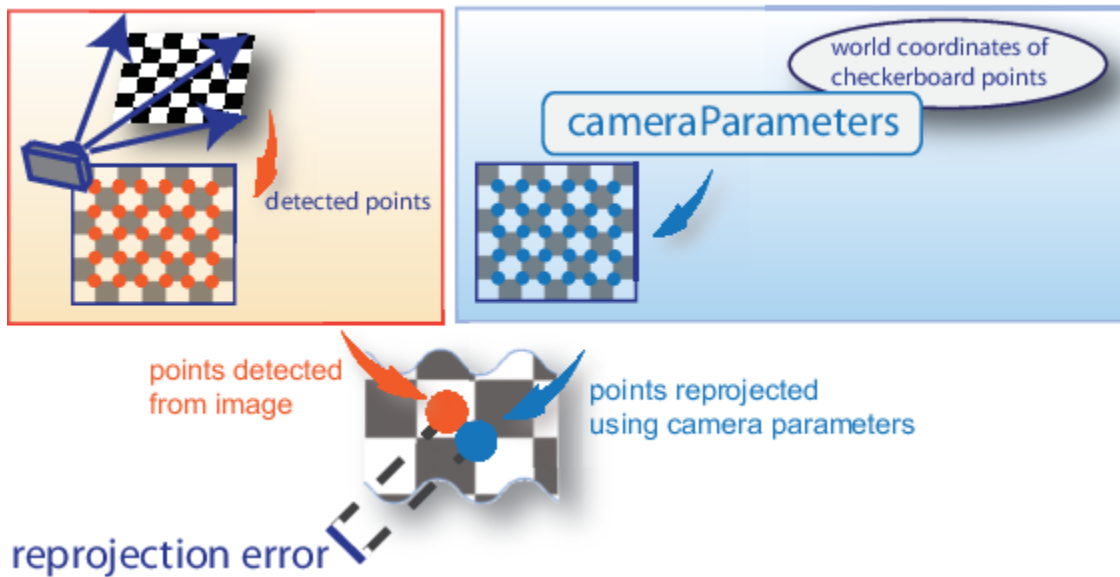
Evaluate Calibration Results

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.



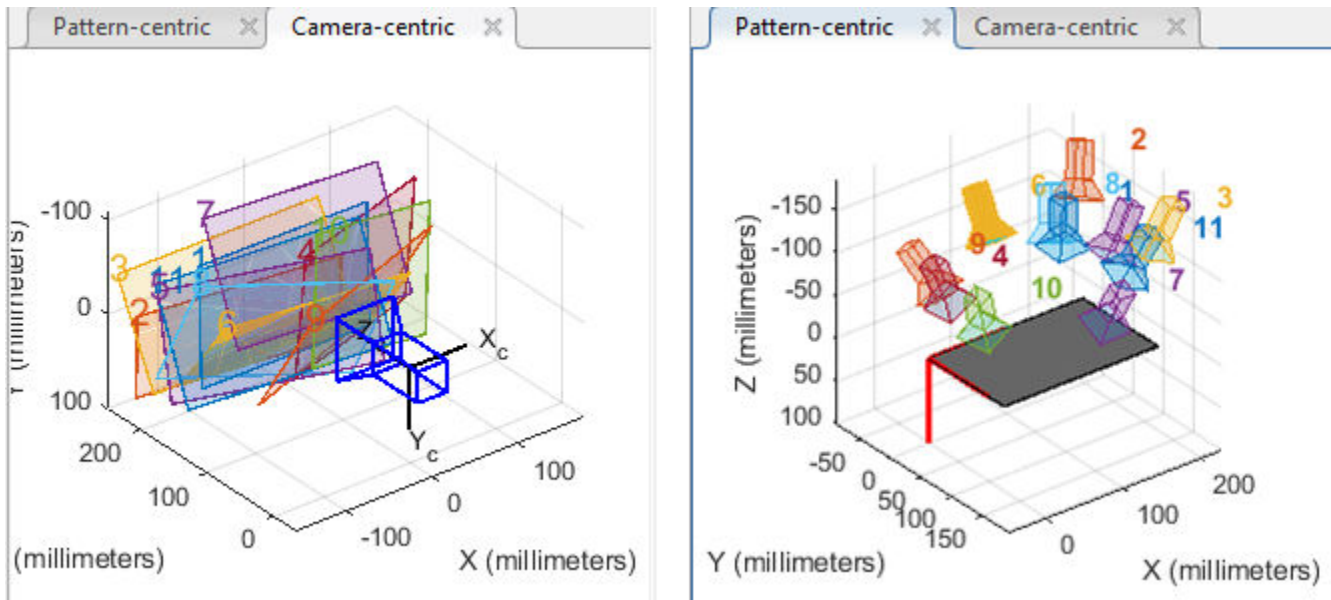
Select an image in one of these ways:

- 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

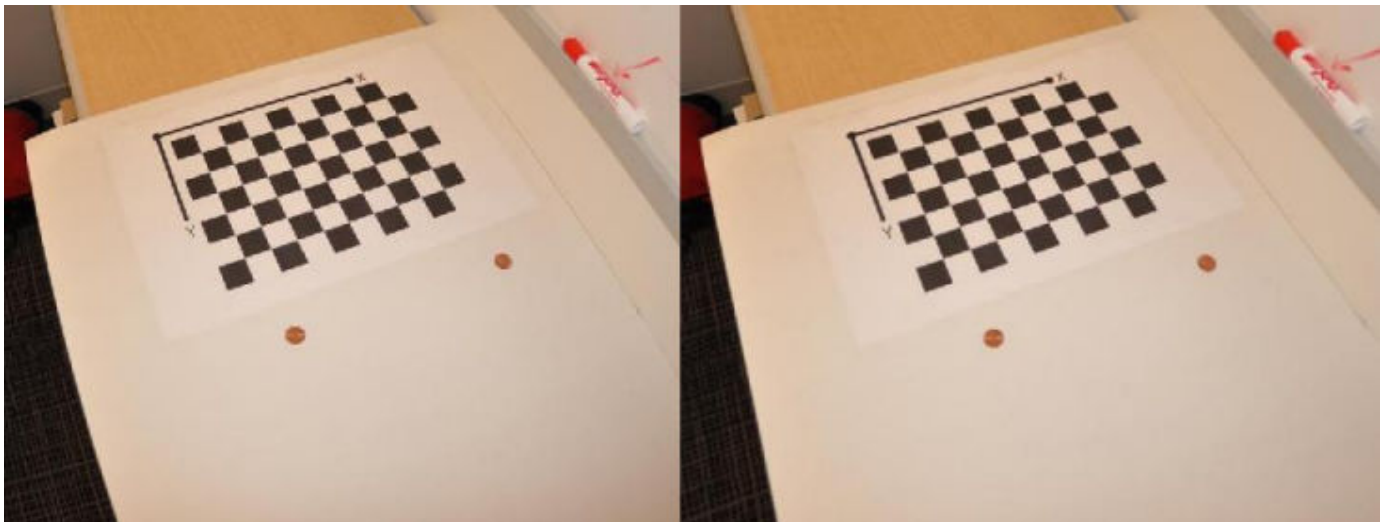
The 3-D extrinsic parameters plot provides a camera-centric view of the patterns and a pattern-centric view of the camera. The camera-centric view is helpful if the camera was stationary when the

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

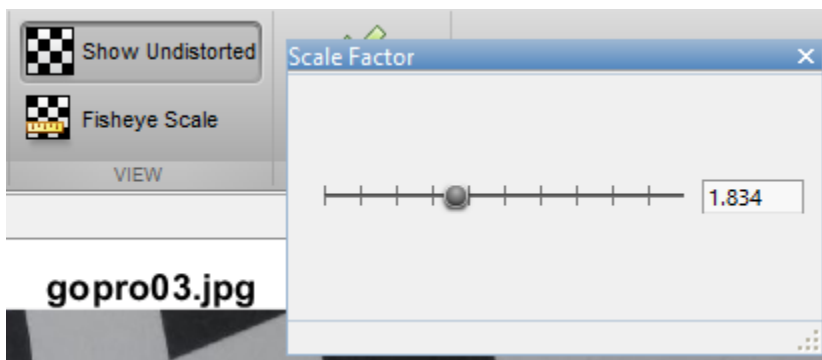
To view the effects of removing lens distortion, click **Show Undistorted** in the **View** section of the **Calibration** tab. If the calibration was accurate, the distorted lines in the image become straight.



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.

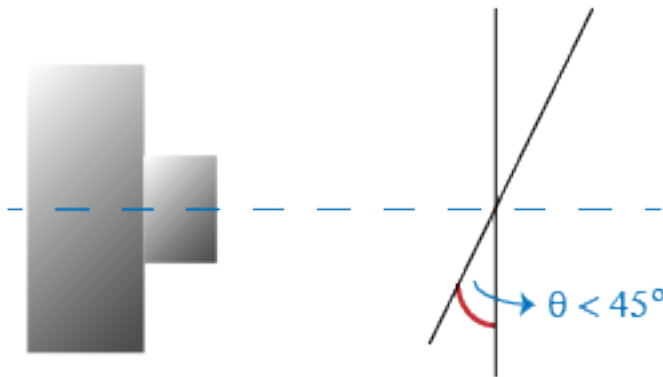
Add or Remove Images

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:

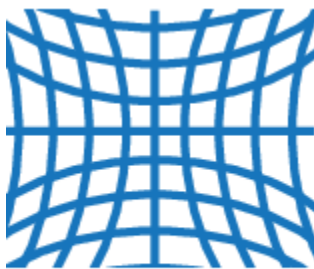
- The images have a high mean reprojection error.
- The images are blurry.
- The images contain a checkerboard at an angle greater than 45 degrees relative to the camera plane.



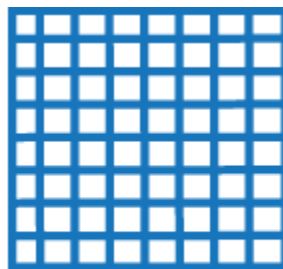
- The images contain incorrectly detected checkerboard points.

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 .

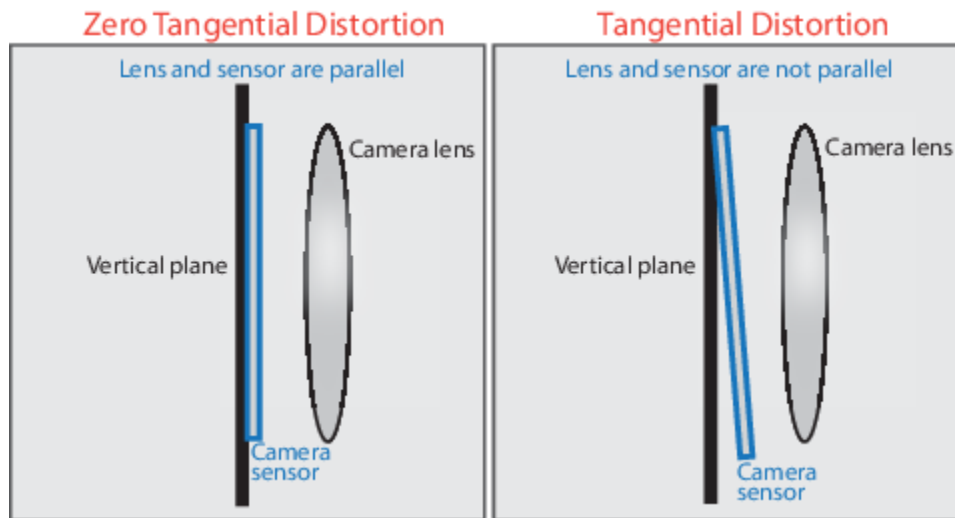
The undistorted pixel locations are in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

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” on page 1-52. 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 Omnidirectional 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 Omnidirectional Camera Calibration.” *ISPRS Journal of Photogrammetry and Remote 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” on page 1-47
- “Measuring Planar Objects with a Calibrated Camera” on page 1-52
- “Structure From Motion From Two Views” on page 1-37
- “Structure From Motion From Multiple Views” on page 1-70
- “Depth Estimation From Stereo Video” on page 1-61

- “3-D Point Cloud Registration and Stitching” on page 5-54
- “Uncalibrated Stereo Image Rectification” on page 1-78
- Checkerboard pattern

More About

- “Stereo Camera Calibrator App” on page 13-25
- “Coordinate Systems”

External Websites

- [Camera Calibration with MATLAB](#)

Stereo Camera Calibrator App

In this section...

“Stereo Camera Calibrator Overview” on page 13-25
 “Stereo Camera Calibration” on page 13-25
 “Open the Stereo Camera Calibrator” on page 13-26
 “Prepare Pattern, Camera, and Images” on page 13-26
 “Add Image Pairs” on page 13-29
 “Calibrate” on page 13-31
 “Evaluate Calibration Results” on page 13-31
 “Improve Calibration” on page 13-35
 “Export Camera Parameters” on page 13-37

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.

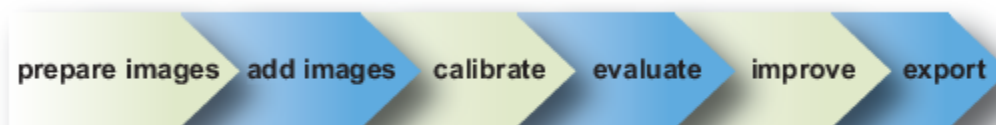
The **Stereo Camera Calibrator** app produces an object containing the stereo camera parameters. You can use this object to

- 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 use these functions directly in the MATLAB workspace. For a list of calibration functions, see “Single and Stereo 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 and Stereo 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 improve 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

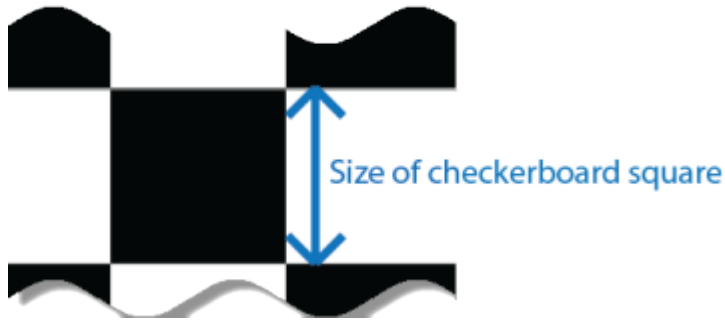
The **Camera Calibrator** app uses a checkerboard pattern. A checkerboard pattern is a convenient calibration target. If you want to use a different pattern to extract key points, you can use the camera calibration MATLAB functions directly. See “Single and Stereo Camera Calibration” for the list of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



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

Camera Setup

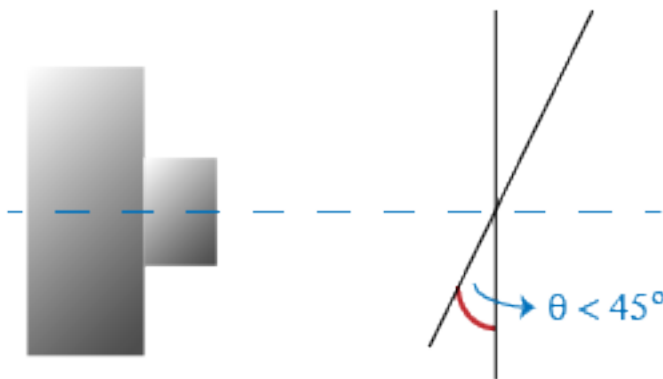
To calibrate your camera, follow these rules:

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

Capture Images

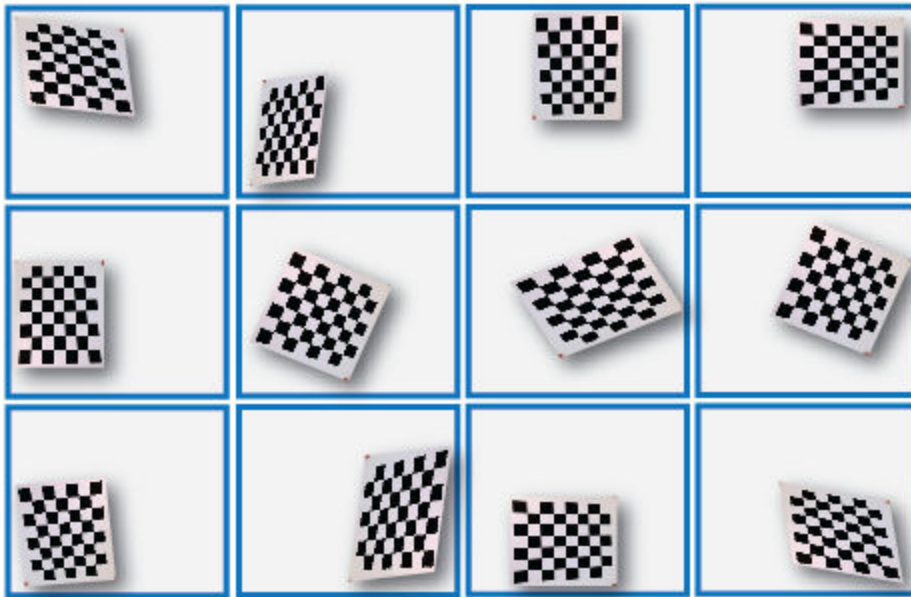
For best results, use at least 10 to 20 images of the calibration pattern. The calibrator requires at least three images. Use uncompressed images or images in lossless compression formats such as PNG. For greater calibration accuracy:

- 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.

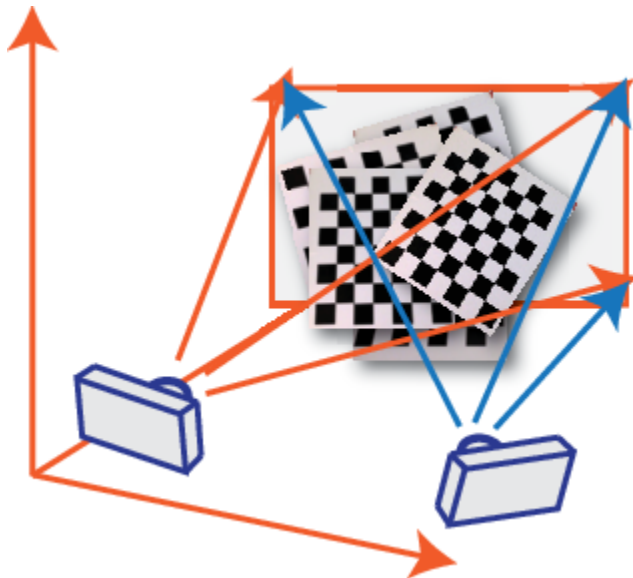


- Do not modify the images, (for example, do not crop them).

- 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.

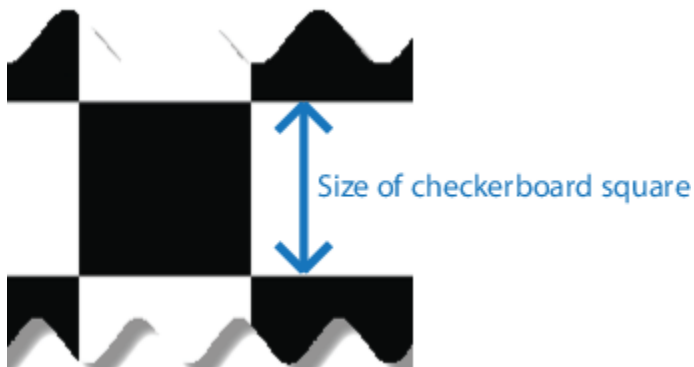
- For greater reconstruction accuracy at longer distances, position your cameras farther apart.

Add Image Pairs

To begin calibration, click [Add Image Pairs](#), specifically two sets of stereo images of the checkerboard, one set from each camera.

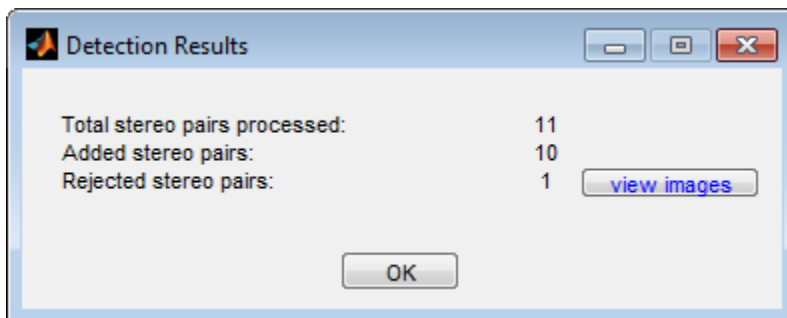
Load Images

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.



Analyze Images

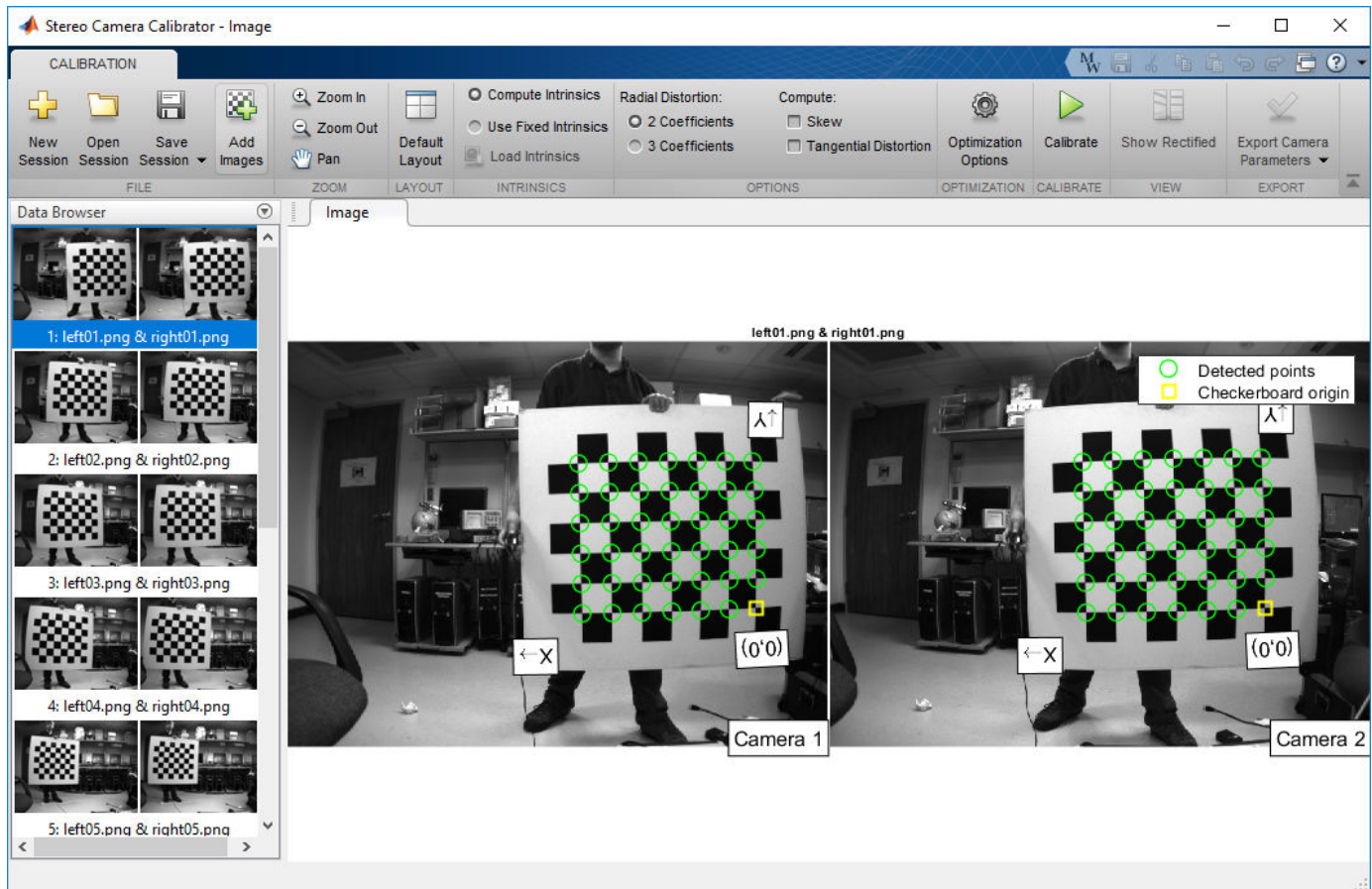
The calibrator attempts to detect a checkerboard in each of the added images, displaying an Analyzing Images progress bar window, indicating detection progress. If any of the images are rejected, the Detection Results dialog box appears, which contains diagnostic information. The results indicate how 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.

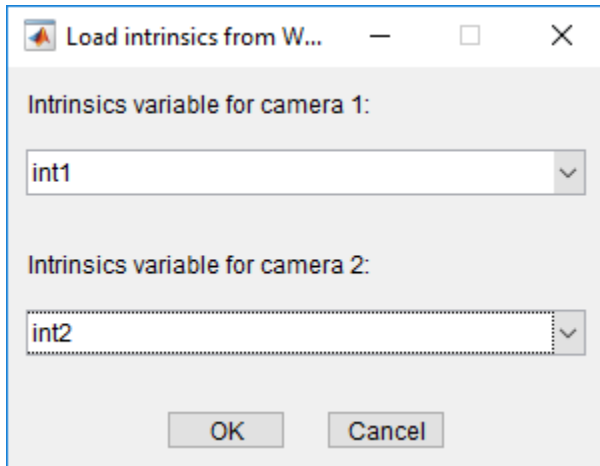
Intrinsics

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 Intrinsic** in the Intrinsic 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 Intrinsic**. For example, if the `wideBaselineStereo` struct contains the intrinsics for both cameras.

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


Then, click **Load Intrinsic**s 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

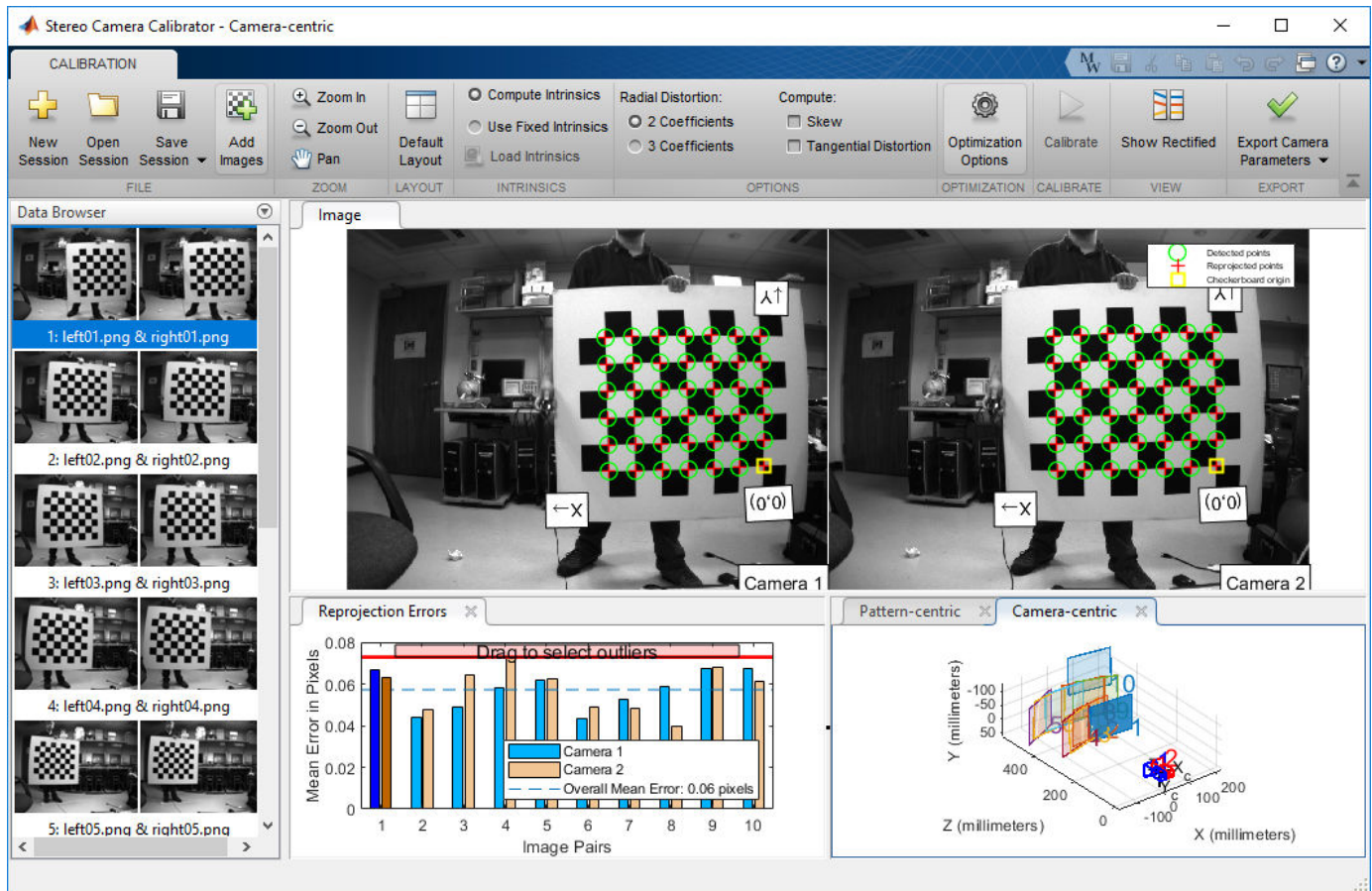
When the camera has severe lens distortion, the app can fail to compute the initial values for the camera intrinsics. If you have the manufacturer's specifications for your camera and know the pixel size, focal length, or lens characteristics, you can manually set initial guesses for camera intrinsics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

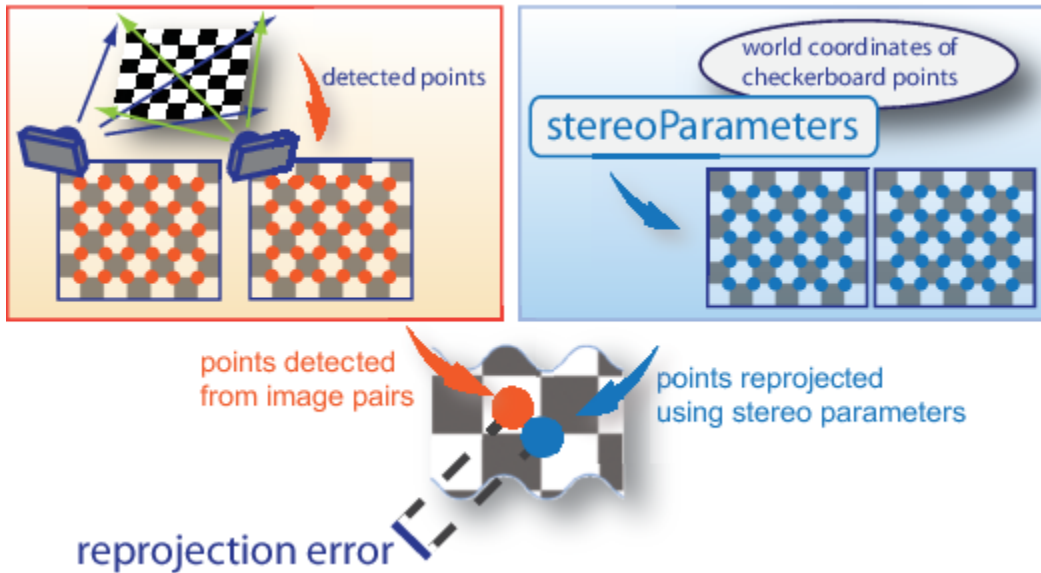
Evaluate Calibration Results

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

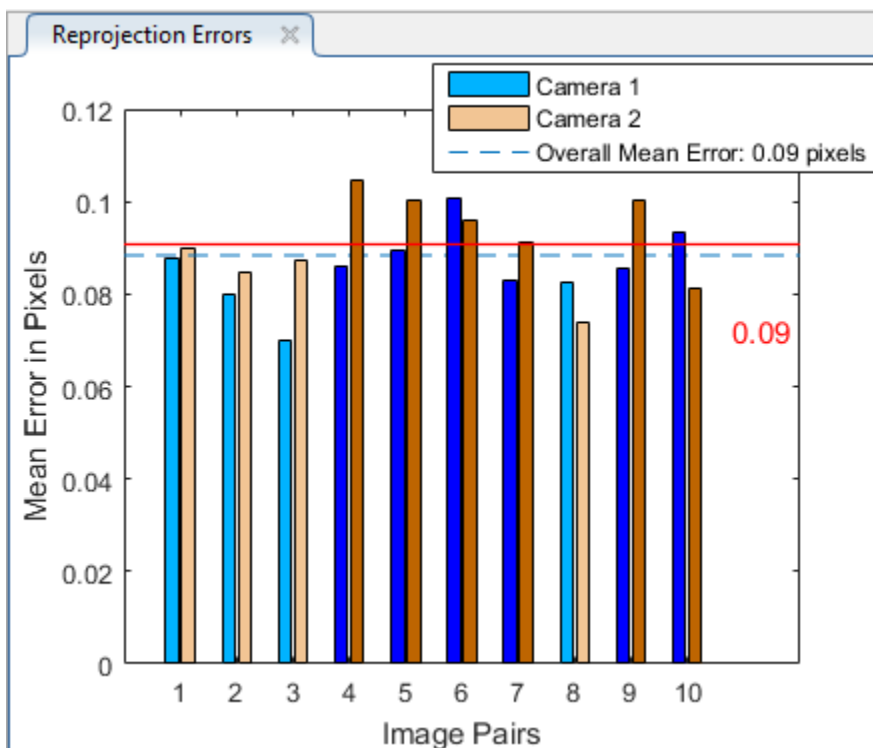
The reprojection errors are the distances, in pixels, between the detected and the reprojected points. The **Stereo Camera Calibrator** app calculates reprojection errors by projecting the checkerboard points from world coordinates, defined by the checkerboard, into image coordinates. The app then compares 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** 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.

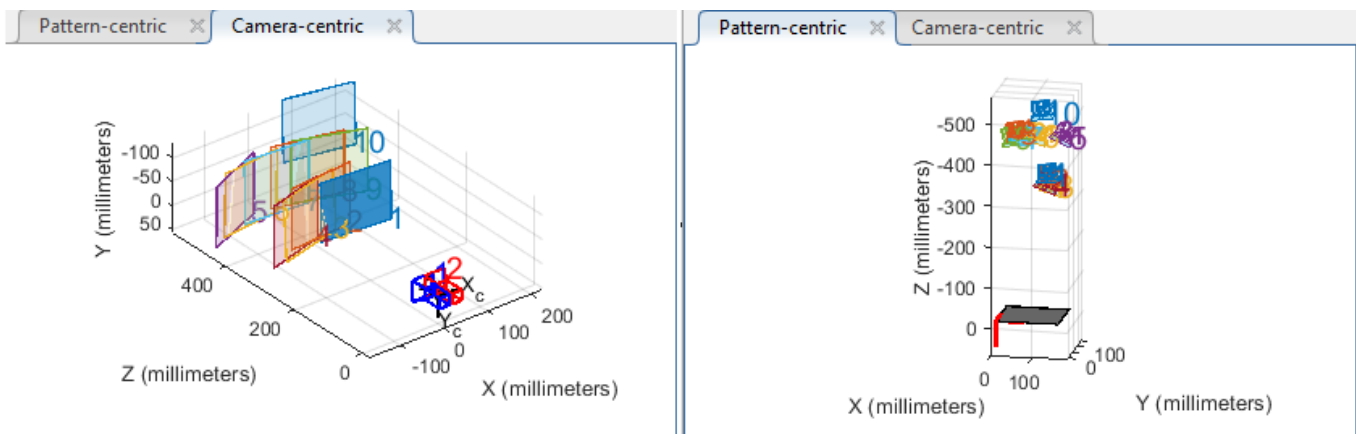


Select an image pair in one of these ways:

- 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.

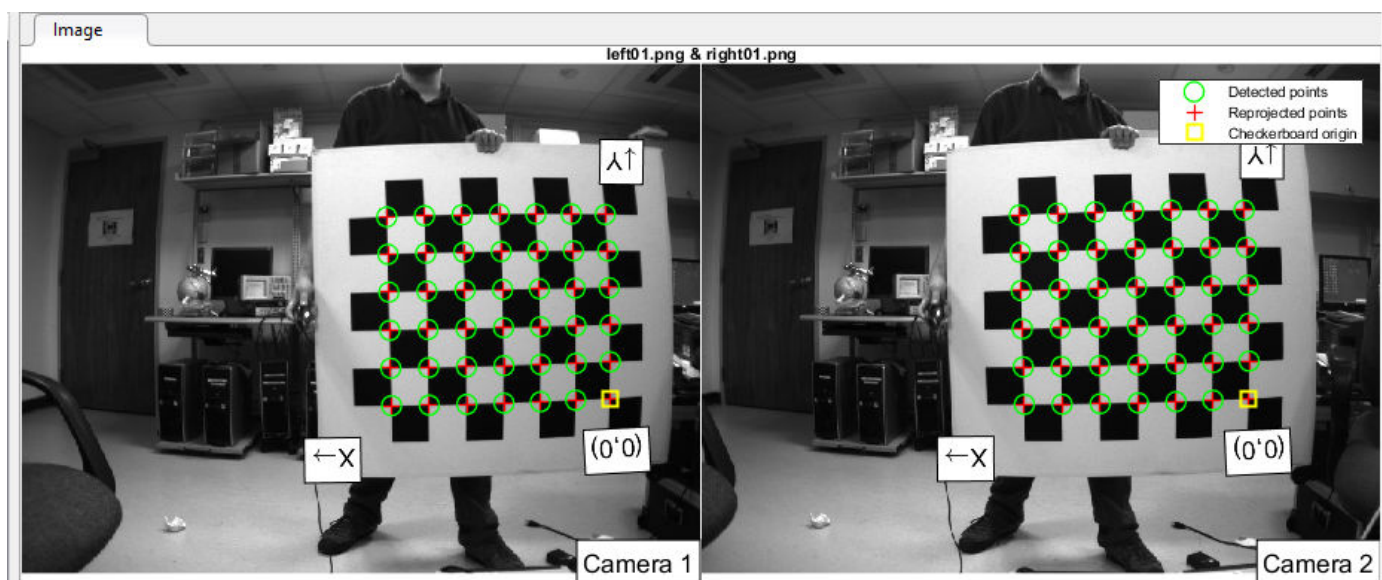
Examine Extrinsic Parameter Visualization

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 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.



Checking 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.

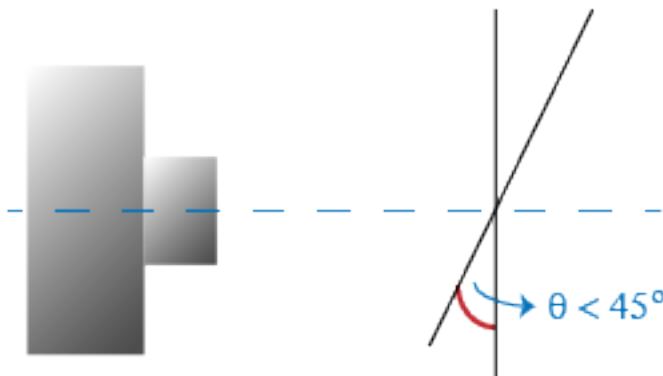
Add or Remove Images

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:

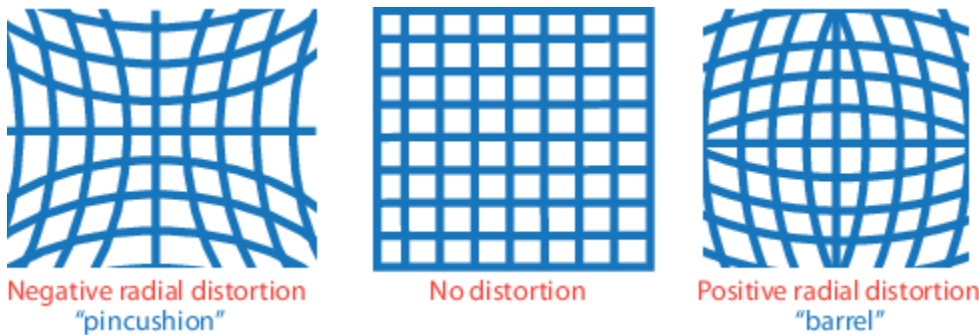
- The images have a high mean reprojection error.
- The images are blurry.
- The images contain a checkerboard at an angle greater than 45 degrees relative to the camera plane.



- 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.



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.
- $r^2: x^2 + y^2$

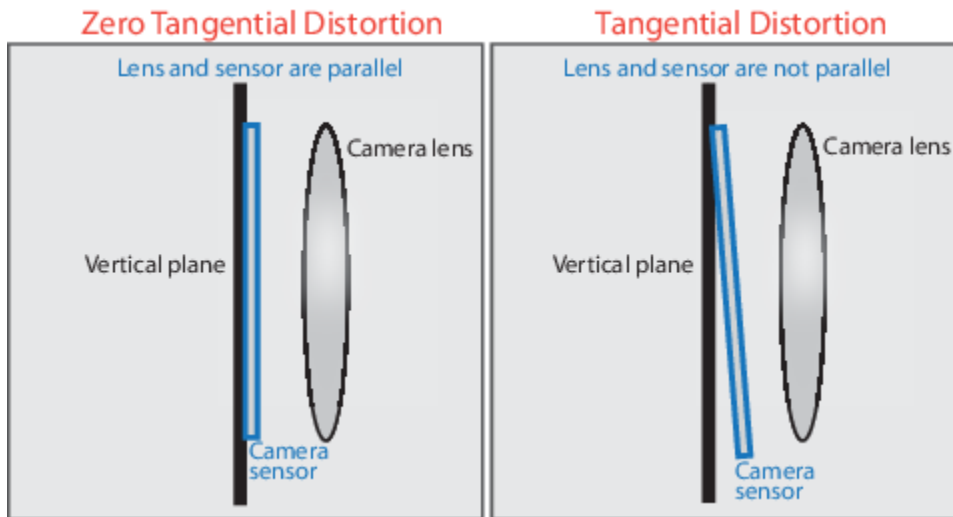
Typically, two coefficients are sufficient for calibration. For severe distortion, such as in wide-angle lenses, you can select 3 coefficients to include k_3 .

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” on page 1-52. 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" on page 1-47
- "Measuring Planar Objects with a Calibrated Camera" on page 1-52
- "Structure From Motion From Two Views" on page 1-37
- "Structure from Motion from Multiple Views" on page 13-46
- "Depth Estimation From Stereo Video" on page 1-61
- "3-D Point Cloud Registration and Stitching" on page 5-54
- "Uncalibrated Stereo Image Rectification" on page 1-78
- Checkerboard pattern

More About

- "Single Camera Calibrator App" on page 13-8
- "Coordinate Systems"

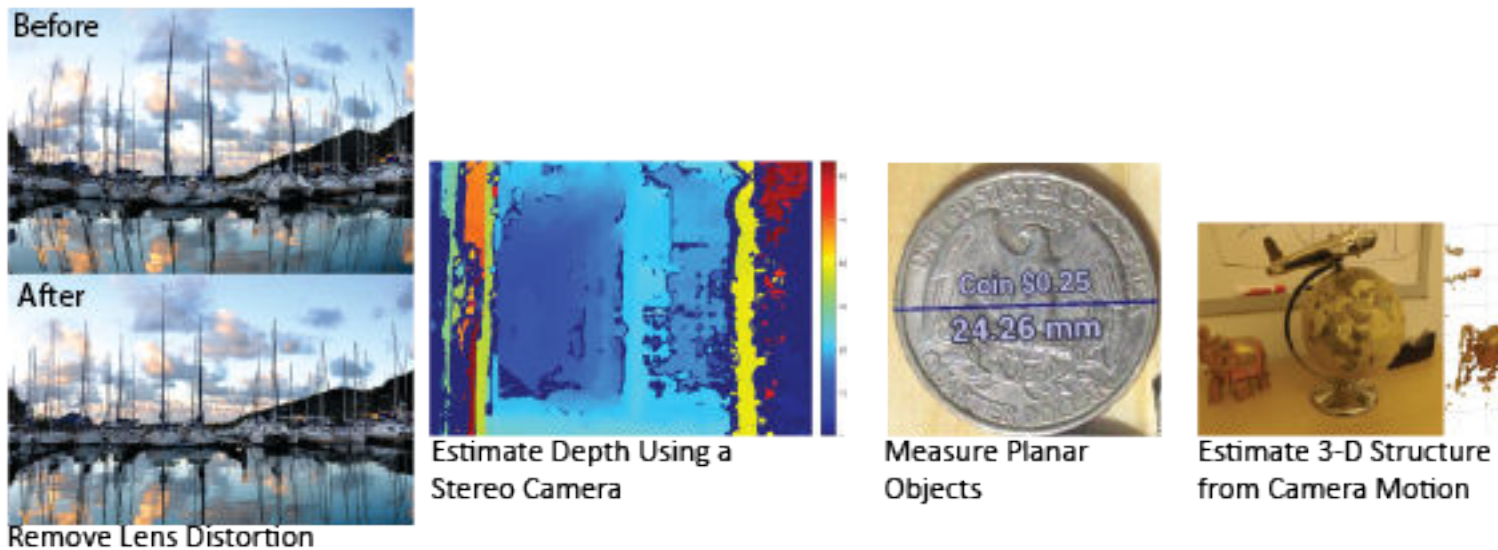
External Websites

- 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:



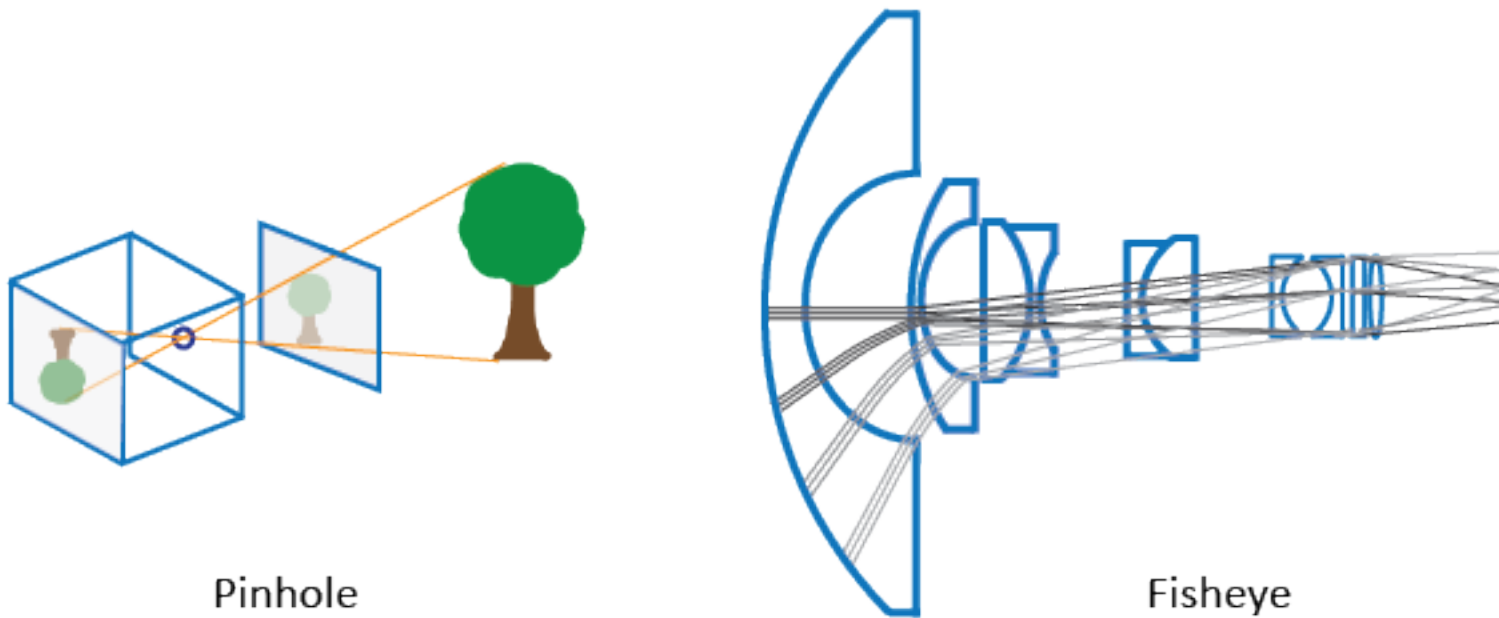
Camera parameters include intrinsics, extrinsics, and distortion coefficients. To estimate the camera parameters, you need to have 3-D world points and their corresponding 2-D image points. You can get these correspondences using multiple images of a calibration pattern, such as a checkerboard. Using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

The Computer Vision Toolbox contains calibration algorithms for the pinhole camera model and the fisheye camera model.

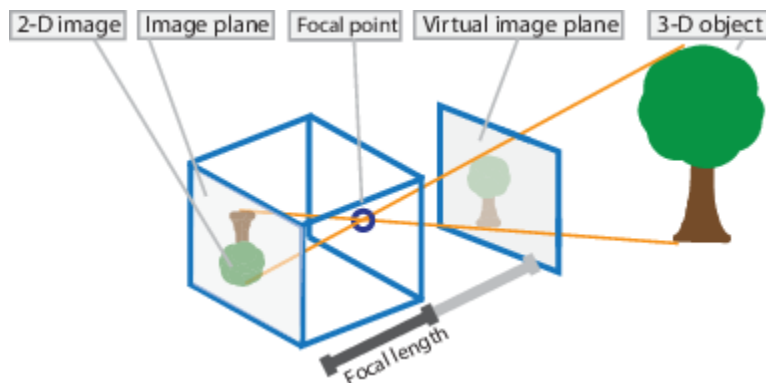


The pinhole calibration algorithm is based on the model proposed by Jean-Yves Bouguet [3]. The model includes, the pinhole camera model [1] and lens distortion [2]. The pinhole camera model does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent a real camera, the full camera model used by the algorithm includes the radial and tangential lens distortion.

Because of the extreme distortion a fisheye lens produces, the pinhole model cannot model a fisheye camera. For details on camera calibration using the fisheye model, see “Fisheye Calibration Basics” on page 13-2.

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.

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

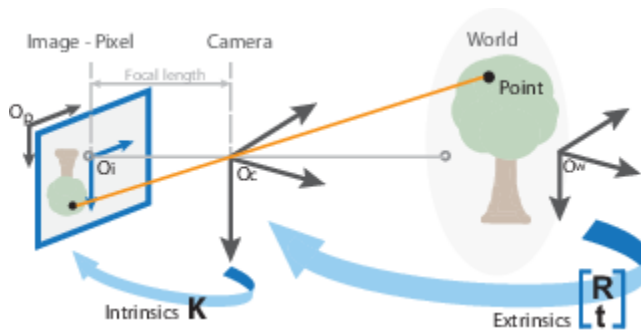
Scale factor
Image points
World points

$$P = \begin{bmatrix} R \\ t \end{bmatrix} K$$

Camera matrix
Extrinsics
Intrinsic matrix

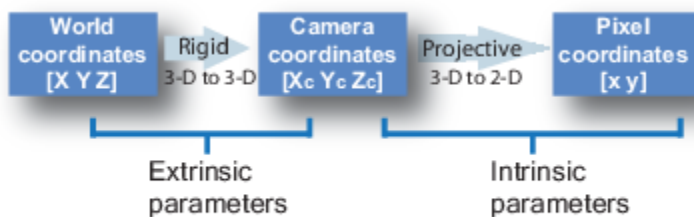
Rotation and translation

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.



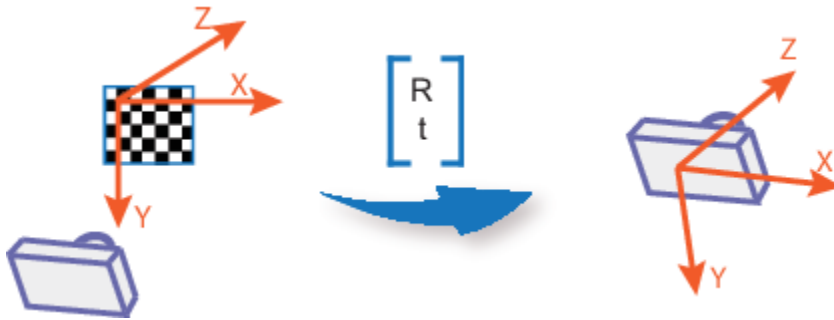
Camera Calibration Parameters

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.

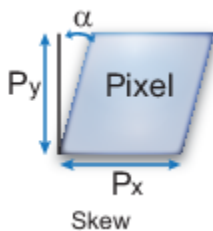


Intrinsic Parameters

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:



$[c_x \ c_y]$ — Optical center (the principal point), in pixels.

(f_x, f_y) — Focal length in pixels.

$$f_x = F/p_x$$

$$f_y = F/p_y$$

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

(p_x, p_y) — Size of the pixel in world units.

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

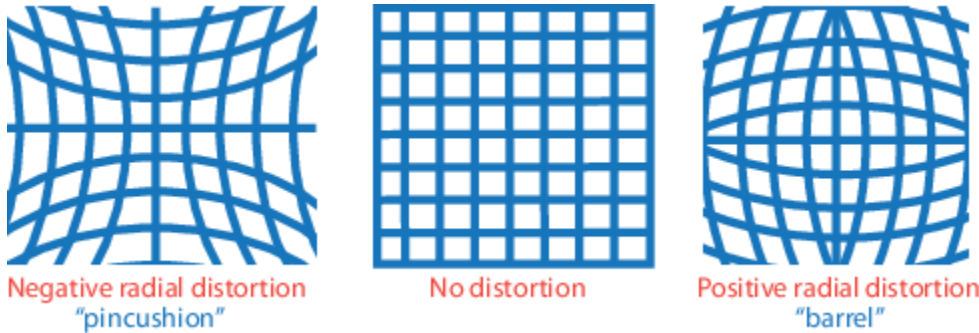
$$s = f_x \tan \alpha$$

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.



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

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

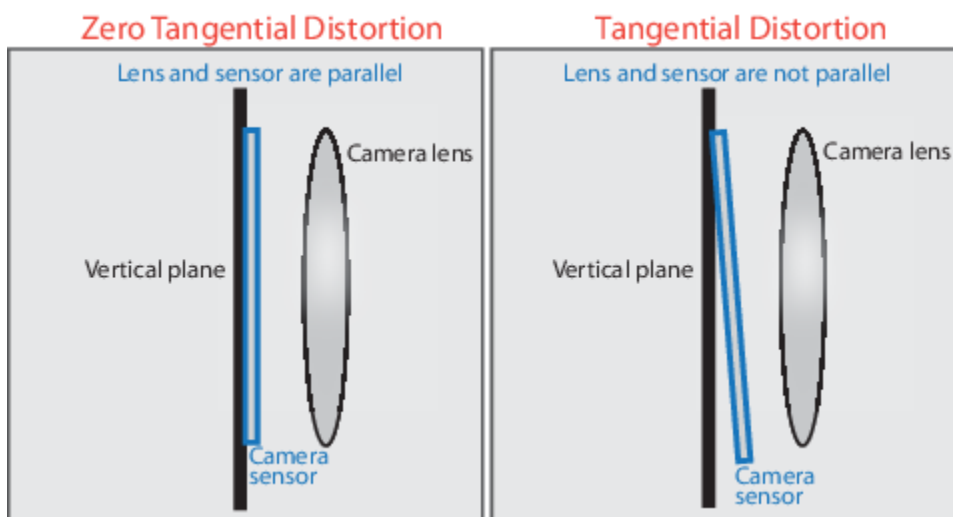
$$y_{\text{distorted}} = y(1 + k_1*r^2 + k_2*r^4 + k_3*r^6)$$

- 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 .

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

Apps

Camera Calibrator | Stereo Camera Calibrator

Related Examples

- "Single Camera Calibrator App" on page 13-8
- "Stereo Camera Calibrator App" on page 13-25
- "Evaluating the Accuracy of Single Camera Calibration" on page 1-47
- "Fisheye Calibration Basics" on page 13-2
- "Configure Monocular Fisheye Camera" (Automated Driving Toolbox)
- "Calibrate a Monocular Camera" (Automated Driving Toolbox)
- "Measuring Planar Objects with a Calibrated Camera" on page 1-52
- "Structure From Motion From Two Views" on page 1-37
- "Structure From Motion From Multiple Views" on page 1-70

Structure from Motion

In this section...

“Structure from Motion from Two Views” on page 13-45

“Structure from Motion from Multiple Views” on page 13-46

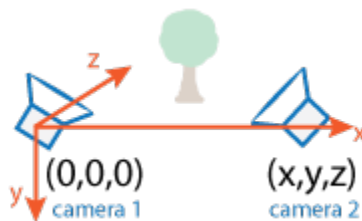
Structure from motion (SfM) is the process of estimating the 3-D structure of a scene from a set of 2-D images. SfM is used in many applications, such as 3-D scanning, augmented reality, and visual simultaneous localization and mapping (SLAM).

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:

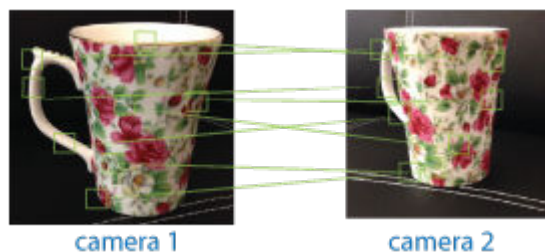
- The size of an object in the scene
- Information from another sensor, for example, an odometer.

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.

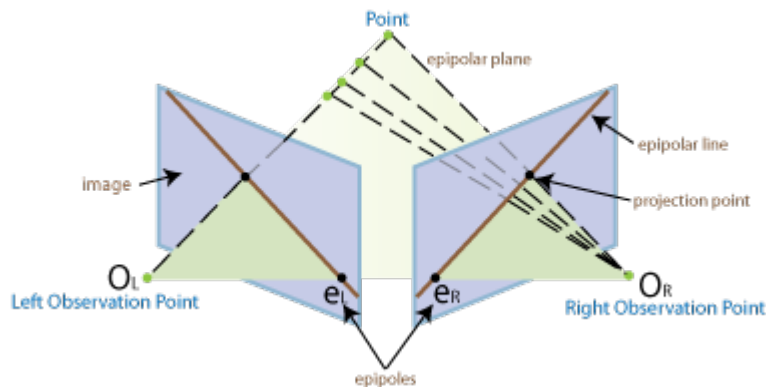


- 1 SfM requires point correspondences between images. Find corresponding points either by matching features or tracking points from image 1 to image 2. Feature tracking techniques, such as Kanade-Lucas-Tomasi (KLT) algorithm, work well when the cameras are close together. As cameras move further apart, the KLT algorithm breaks down, and feature matching can be used instead.



Distance Between Cameras (Baseline)	Method for Finding Point Correspondences	Example
Wide	Match features using <code>matchFeatures</code>	“Find Image Rotation and Scale Using Automated Feature Matching” on page 4-22
Narrow	Track features using <code>vision.PointTracker</code>	“Face Detection and Tracking Using the KLT Algorithm” on page 7-20

- 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.



- 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.
- 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.

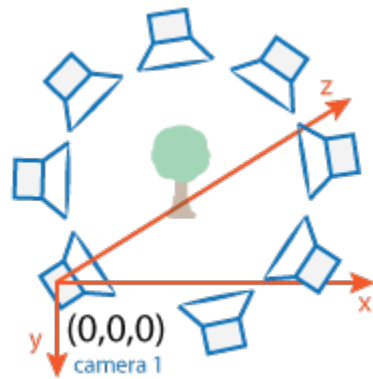
The `triangulate` function takes two camera matrices, which you can compute using `cameraMatrix`.

- Use `pcshow` or `pcplayer` to display the reconstruction. Use `plotCamera` to visualize the camera poses.

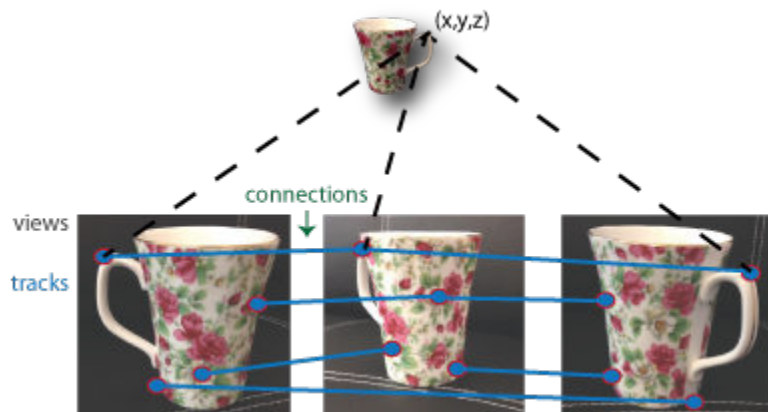
To recover the scale of the reconstruction, you need additional information. One method to recover the scale is to detect an object of a known size in the scene. The “Structure From Motion From Two Views” on page 1-37 example shows how to recover scale by detecting a sphere of a known size in the point cloud of the scene.

Structure from Motion from Multiple Views

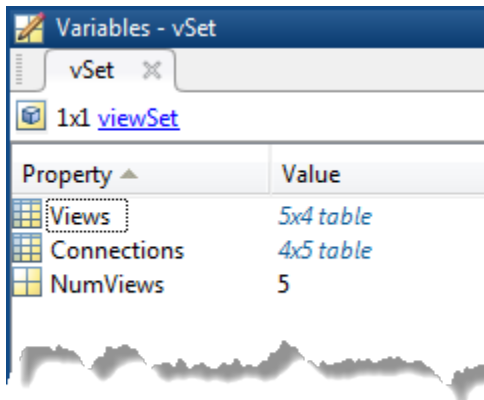
For most applications, such as robotics and autonomous driving, SfM uses more than two views.



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 `imageviewset` 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`. The 3-D point can be stored in a `worldpointset` object. The `worldpointset` object also stores the correspondence between the 3-D points and the 2-D image points across camera views.

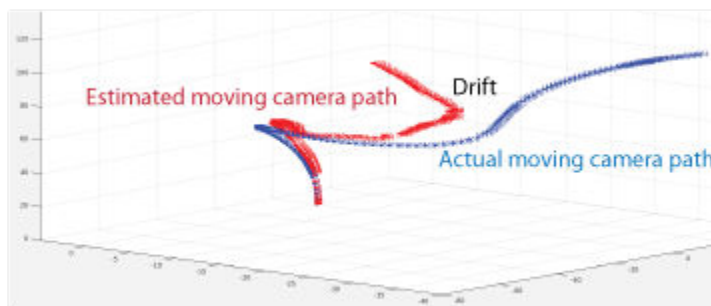


Using the approach in SfM from two views, you can find the pose of camera 2 relative to camera 1. To extend this approach to the multiple view case, find the pose of camera 3 relative to camera 2, and so on. The relative poses must be transformed into a common coordinate system. Typically, all camera poses are computed relative to camera 1 so that all poses are in the same coordinate system. You can use `imageviewset` to manage camera poses. The `imageviewset` 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. You can fix the camera poses and refine only the 3-D point locations using `bundleAdjustmentMotion`. You can also fix the camera poses and refine only the 3-D locations using `bundleAdjustmentStructure`.

Another method of reducing drift is by using pose graph optimization over the `imageviewset` object. Once there is a loop closure detected, add a new connection to the `imageviewset` object and use the `optimizePoses` function to refine the camera poses constrained by relative poses.



The “Structure From Motion From Two Views” on page 1-37 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 `imageviewset` object to store and manage the data associated with each view.

The “Monocular Visual Simultaneous Localization and Mapping” on page 1-18 example shows you how to process image data from a monocular camera to build a map of an indoor environment and estimate the motion of the camera.

See Also

Apps

Camera Calibrator | **Stereo Camera Calibrator**

Functions

bundleAdjustment | bundleAdjustmentMotion | bundleAdjustmentStructure |
cameraMatrix | estimateFundamentalMatrix | matchFeatures | pointTrack |
relativeCameraPose | triangulateMultiview

Objects

imageviewset | vision.PointTracker | worldpointset

See Also**Related Examples**

- “Structure From Motion From Two Views” on page 1-37
- “Structure From Motion From Multiple Views” on page 1-70
- “Monocular Visual Simultaneous Localization and Mapping” on page 1-18

Object Detection

- “Choose an Object Detector” on page 14-2
- “Getting Started with SSD Multibox Detection” on page 14-9
- “Getting Started with Object Detection Using Deep Learning” on page 14-13
- “How Labeler Apps Store Exported Pixel Labels” on page 14-16
- “Anchor Boxes for Object Detection” on page 14-21
- “Getting Started with YOLO v2” on page 14-26
- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 14-30
- “Getting Started with Mask R-CNN for Instance Segmentation” on page 14-36
- “Getting Started with Semantic Segmentation Using Deep Learning” on page 14-43
- “Training Data for Object Detection and Semantic Segmentation” on page 14-45
- “Create Automation Algorithm for Labeling” on page 14-49
- “Label Pixels for Semantic Segmentation” on page 14-53
- “Get Started with the Image Labeler” on page 14-63
- “Choose an App to Label Ground Truth Data” on page 14-75
- “Get Started with the Video Labeler” on page 14-78
- “Use Custom Image Source Reader for Labeling” on page 14-94
- “Use Sublabels and Attributes to Label Ground Truth Data” on page 14-96
- “Temporal Automation Algorithms” on page 14-100
- “View Summary of Ground Truth Labels” on page 14-102
- “Share and Store Labeled Ground Truth Data” on page 14-106
- “Keyboard Shortcuts and Mouse Actions for Image Labeler” on page 14-112
- “Keyboard Shortcuts and Mouse Actions for Video Labeler” on page 14-116
- “Point Feature Types” on page 14-120
- “Local Feature Detection and Extraction” on page 14-126
- “Train a Cascade Object Detector” on page 14-143
- “Train Optical Character Recognition for Custom Fonts” on page 14-156
- “Troubleshoot ocr Function Results” on page 14-160
- “Create a Custom Feature Extractor” on page 14-161
- “Image Retrieval with Bag of Visual Words” on page 14-164
- “Image Classification with Bag of Visual Words” on page 14-167

Choose an Object Detector

The Computer Vision Toolbox provides object detectors to use for finding and classifying objects in an image or video. Train a detector using an object detector function, then use it with machine learning and deep learning to quickly and accurately predict the location of an object in an image.

When choosing a detector, consider whether you need these features::

Application and Performance

- Single vs Multiple classes — Multiple classes require a variation of different classifiers used at multiple locations and scales on the image or video.
- Runtime performance — Detectors vary in performance depending on the time it takes to detect objects in an image. A detector trained for a single class, or a detector trained to detect objects that are similar in pose and shape, will have a faster runtime performance than a deep learning detector trained on multiple objects. More importantly, deep learning is slower because it requires more computations than machine learning or feature-based detection approaches.
- Machine learning — Machine learning uses two types of techniques: **supervised learning**, which trains a model on known input and output data so that it can predict future outputs, and **unsupervised learning**, which finds hidden patterns or intrinsic structures in input data. For more details, see “Machine Learning in MATLAB” (Statistics and Machine Learning Toolbox)
- Deep learning — Implements deep neural networks with algorithms, pretrained models, and apps. You can use convolutional neural networks to perform classification and regression on images. For more details, see “Getting Started with Object Detection Using Deep Learning” on page 14-13.

Deployment

- C/C++ code generation — SSD, YOLO, ACF, and system object-based detectors support MATLAB Coder C and C++ code generation for a variety of hardware platforms, from desktop systems to embedded hardware. For more details, see MATLAB Coder. The R-CNN-based detectors do not support code generation.
- GPU code generation — Deep learning-based detectors support GPU code generation with optimized CUDA[®] by GPU Coder[™] for embedded vision, and autonomous systems. For more details, see GPU Coder.

Use the table to view and compare the object detector functions.

Detector	Multiple Classes Support	Deep Learning Support	Code Generation Support	GPU Support	Example	Description
fasterRCNNObjectDetector	Yes	Yes	No	Yes	“Object Detection Using Faster R-CNN Deep Learning” on page 3-197	<ul style="list-style-type: none"> • Requires GPU for optimal performance. • Use this detector when you need more precise object localization accuracy. • Best performance of the R-CNN family, but slower than YOLO v2 and SSD. <p>Faster R-CNN is a two-stage network. The second stage refines detection proposals produced by the first stage, which helps improve localization at the cost of runtime performance.</p> <p>“Comparison of R-CNN Object Detectors” on page 14-32</p>
fastRCNNObjectDetector	Yes	Yes	No	Yes	Train Fast R-CNN Stop Sign Detector on page 3-273	<ul style="list-style-type: none"> • Consider starting with the fasterRCNNObjectDetector. • Requires GPU for optimal performance. • Use this detector if you have your own method for producing object regions. • Faster than R-CNN, but slower than Faster R-CNN. <p>“Comparison of R-CNN Object Detectors” on page 14-32</p>

Detector	Multiple Classes Support	Deep Learning Support	Code Generation Support	GPU Support	Example	Description
rcnnObjectDetector	Yes	Yes	No	Yes	“Train Object Detector Using R-CNN Deep Learning” on page 3-183	<ul style="list-style-type: none"> • Consider starting with the fasterRCNNObjectDetector. • Requires GPU for optimal performance. • Slowest of the R-CNN-based detectors. <p>This algorithm combines rectangular region proposals with convolutional neural network features. It is a two-stage detection algorithm. The first stage identifies a subset of regions in an image that might contain an object. The second stage classifies the object in each region.</p> <p>“Comparison of R-CNN Object Detectors” on page 14-32</p>
yoloV2ObjectDetector	Yes	Yes	Yes	Yes	“Object Detection Using YOLO v2 Deep Learning” on page 3-123	<ul style="list-style-type: none"> • Consider using SSD or YOLO v3 for better performance across various sizes. • Requires GPU for optimal performance. • Use this detector when better runtime performance is desired and you have objects that do not drastically vary in size or are small in the image. • Better runtime performance compared to Faster R-CNN. <p>YOLO v2 uses a single stage network to perform object detection.</p>

Detector	Multiple Classes Support	Deep Learning Support	Code Generation Support	GPU Support	Example	Description
ssdObjectDetector	Yes	Yes	Yes	Yes	"Object Detection Using SSD Deep Learning" on page 3-23	<ul style="list-style-type: none"> • Requires GPU for optimal performance. • Use this detector when you need to detect objects of various sizes and better runtime performance is desired. • Better runtime performance than Faster R-CNN and YOLO v2. <p>Single shot detector (SSD) uses a single stage detection network to detect objects using multi-scale features.</p>

Detector	Multiple Classes Support	Deep Learning Support	Code Generation Support	GPU Support	Example	Description
acfObjectDetector	No	No	Yes	No	Train ACF-based Stop Sign Detector on page 3-247	<ul style="list-style-type: none"> • A rigid object detector that is suited for single class object detection. • Consider using a deep learning object detector if you need to detect multiple object classes or have objects that belong to the same class but are in different configurations or poses. • Use this detector when the object you want to detect has similar pose and shape, and when runtime performance is critical. • Better runtime performance than deep-learning-based detectors on CPU. <p>ACF works well for a single class that can be easily classified regardless of pose. For example, it would work well to detect a person, who can be recognized in multiple poses, such as sitting, standing, or riding a horse.</p> <p>ACF would not work well for detecting vehicles from various viewpoints, such as front, side, and rear.</p>
peopleDetectorACF	Pretrained	No	Yes	No	“Tracking Pedestrians from a Moving Car” on page 7-40	Use this pretrained detector to detect upright positioned people.
vision.PeopleDetector	Pretrained	No	Yes	No	“Depth Estimation From Stereo Video” on page 1-61	Use this pretrained cascade object detector to detect upright positioned people.

Detector	Multiple Classes Support	Deep Learning Support	Code Generation Support	GPU Support	Example	Description
vision.CascadeObjectDetector	No	No	Yes	No	"Detect Faces in an Image Using the Frontal Face Classification Model"	<ul style="list-style-type: none"> Viola-Jones object detector suitable for rigid object detection. Uses HAAR, HOG, or LBP features. If training a new detector, consider starting with ACF for better performance. Use this detector when a pretrained detector is available for an object class you're interested in detecting, and there is little variation in the object's pose or shape.
Mask R-CNN	Yes	Yes	No	Yes	"Getting Started with Mask R-CNN for Instance Segmentation" on page 14-36	Use this detector when you need to segment individual objects.
YOLO v3	Yes	Yes	Yes	Yes	"Object Detection Using YOLO v3 Deep Learning" on page 3-150	YOLO v3 is a single stage network that uses multi-scale features to better handle detection of objects of various sizes.
vehicleDetectorACF	Pretrained	No	Yes	No	"Track Multiple Vehicles Using a Camera" (Automated Driving Toolbox)	Pretrained ACF detector
vehicleDetectorFasterRCNN	Pretrained	Yes	No	Yes	"Train a Deep Learning Vehicle Detector" (Automated Driving Toolbox)	Pretrained Faster R-CNN detector

Detector	Multiple Classes Support	Deep Learning Support	Code Generation Support	GPU Support	Example	Description
vehicleDetectorYoloV2	Pretrained	Yes	Yes	Yes	“Detect Vehicles Using Monocular Camera and YOLO v2” (Automated Driving Toolbox)	Pretrained YOLO v2 detector

See Also

Apps

[Ground Truth Labeler](#) | [Image Labeler](#) | [Video Labeler](#)

Objects

`acfObjectDetector` | `acfObjectDetectorMonoCamera` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `rcnnObjectDetector` | `ssdObjectDetector` | `vehicleDetectorACF` | `vehicleDetectorFasterRCNN` | `vehicleDetectorYoloV2` | `vision.CascadeObjectDetector` | `yoloV2ObjectDetector`

Functions

`trainACFObjectDetector` | `trainCascadeObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `trainRCNNObjectDetector` | `trainSSDObjectDetector` | `trainYoloV2ObjectDetector`

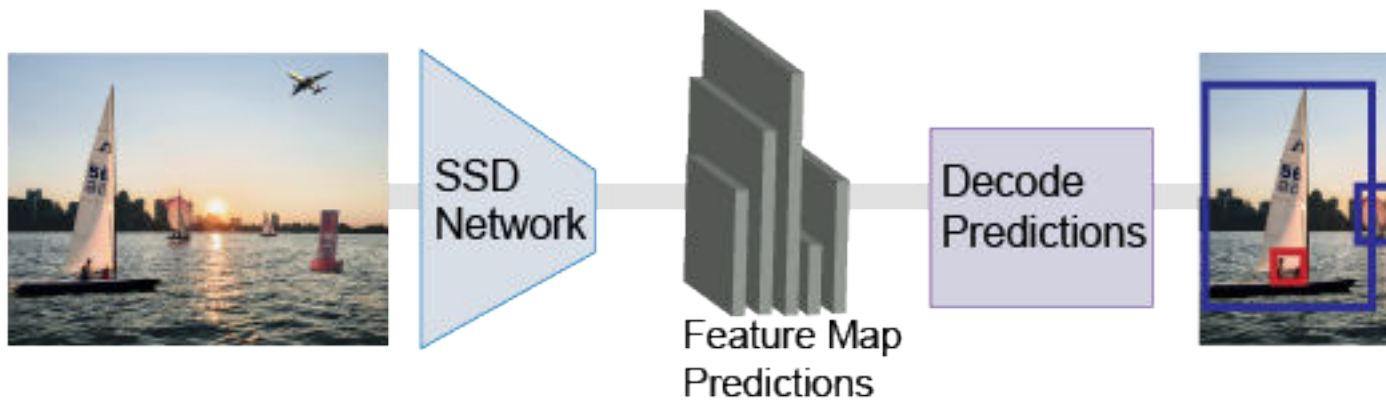
More About

- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 14-30
- “Getting Started with SSD Multibox Detection” on page 14-9
- “Getting Started with Object Detection Using Deep Learning” on page 14-13
- “Getting Started with YOLO v2” on page 14-26
- “Getting Started with Mask R-CNN for Instance Segmentation” on page 14-36

Getting Started with SSD Multibox Detection

The single shot multibox detector (SSD) uses a single stage object detection network that merges detections predicted from multiscale features. The SSD is faster than two-stage detectors, such as the Faster R-CNN detector, and can localize objects more accurately compared to single-scale feature detectors, such as the YOLO v2 detector.

The SSD runs a deep learning CNN on an input image to produce network predictions from multiple feature maps. The object detector gathers and decodes predictions to generate bounding boxes.

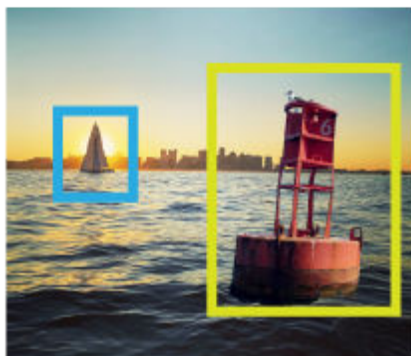


Predict Objects in the Image

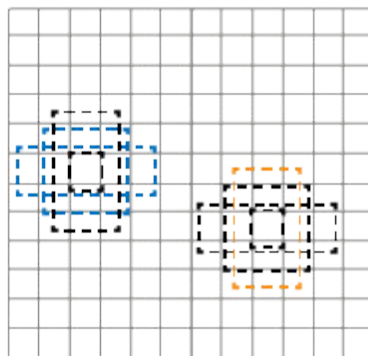
SSD uses anchor boxes to detect classes of objects in an image. For more details, see “Anchor Boxes for Object Detection” on page 14-21. The SSD predicts these two attributes for each anchor box.

- Anchor box offsets — Refine the anchor box position.
- Class probability — Predict the class label assigned to each anchor box.

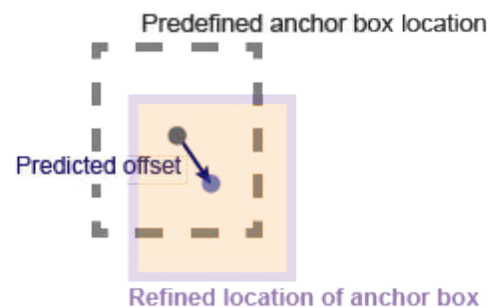
This figure shows predefined anchor boxes (the dotted lines) at each location in a feature map and the refined location after offsets are applied. Matched boxes with a class are in blue and orange.



Ground truth image and bounding boxes



Anchor boxes at each predefined location in each feature map



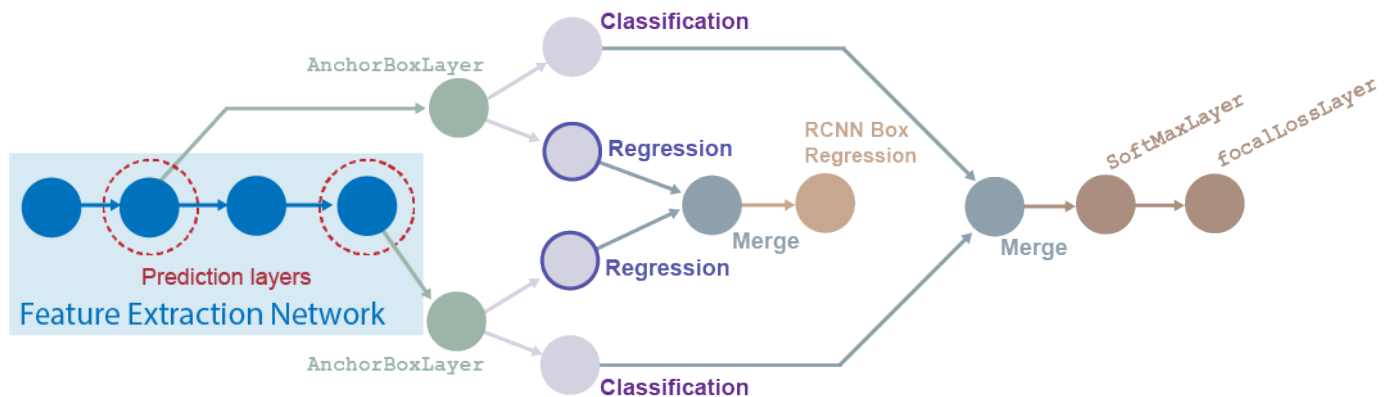
Transfer Learning

With transfer learning, you can use a pretrained CNN as the feature extractor in an SSD detection network. Use the `ssdLayers` function to create an SSD detection network from any pretrained CNN, such as `MobileNet_v2`. For a list of pretrained CNNs, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

You can also design a custom model based on a pretrained image classification CNN. For more details, see “Design an SSD Detection Network” on page 14-10.

Design an SSD Detection Network

You can design a custom SSD model programmatically or use the **Deep Network Designer** app to manually create a network. The app incorporates Computer Vision Toolbox SSD features.



To design an SSD Multibox detection network, follow these steps.

- 1 Start the model with a feature extractor network, which can be initialized from a pretrained CNN or trained from scratch.
- 2 Select prediction layers from the feature extraction network. Any layer from the feature extraction network can be used as a prediction layer. However, to leverage the benefits of using multiscale features for object detection, choose feature maps of different sizes.
- 3 Specify anchor boxes to the prediction layer by attaching an `anchorBoxLayer` to each of the layers.
- 4 Connect the outputs of the `anchorBoxLayer` objects to a classification branch and to a regression branch. The classification branch has at least one convolution layer that predicts the class for each tiled anchor box. The regression branch has at least one convolution layer that predicts anchor box offsets. You can add more layers in the classification and regression branches, however, the final convolution layer (before the merge layer) must have the number of filters according to this table.

Branch	Number of Filters
Classification	Number of anchor boxes + 1 (for background class)
Regression	Four times the number of anchor boxes

- 5 For all prediction layers, combine the outputs of the classification branches by using the `ssdMergeLayer` object. Connect the `ssdMergeLayer` object to a `softmaxLayer` object,

followed by a `focalLossLayer` object. Gather all outputs of the regression branches by using the `ssdMergeLayer` object again. Connect the `ssdMergeLayer` output to an `rcnnBoxRegressionLayer` object.

For more details on creating this type of network, see “Create SSD Object Detection Network” on page 3-115

Train an Object Detector and Detect Objects with an SSD Model

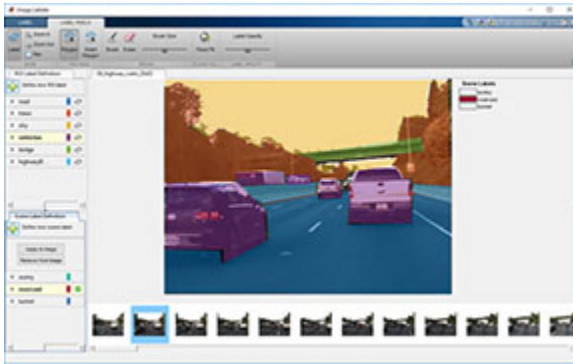
To learn how to train an object detector by using the SSD deep learning technique, see the “Object Detection Using SSD Deep Learning” on page 3-23 example.

Code Generation

To learn how to generate CUDA code using the SSD object detector (created using the `ssdObjectDetector` object), see “Code Generation for Object Detection by Using Single Shot Multibox Detector” on page 2-2.

Label Training Data for Deep Learning

You can use the **Image Labeler**, **Video Labeler**, or **Ground Truth Labeler** (available in Automated Driving Toolbox™) apps to interactively label pixels and export label data for training. The apps can also be used to label rectangular regions of interest (ROIs) for object detection, scene labels for image classification, and pixels for semantic segmentation.



References

- [1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. "SSD: Single Shot MultiBox Detector." In *Computer Vision - ECCV 2016*, edited by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, 9905:21-37. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-46448-0_2.

See Also

Apps

Deep Network Designer | **Ground Truth Labeler** | **Image Labeler** | **Video Labeler**

Objects

`ssdObjectDetector`

Functions

`analyzeNetwork` | `ssdLayers` | `trainSSDObjectDetector`

Related Examples

- “Object Detection Using SSD Deep Learning” on page 3-23
- “Create SSD Object Detection Network” on page 3-115

More About

- “Anchor Boxes for Object Detection” on page 14-21
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

Getting Started with Object Detection Using Deep Learning

Object detection using deep learning provides a fast and accurate means to predict the location of an object in an image. Deep learning is a powerful machine learning technique in which the object detector automatically learns image features required for detection tasks. Several techniques for object detection using deep learning are available such as Faster R-CNN, you only look once (YOLO) v2, and single shot detection (SSD).

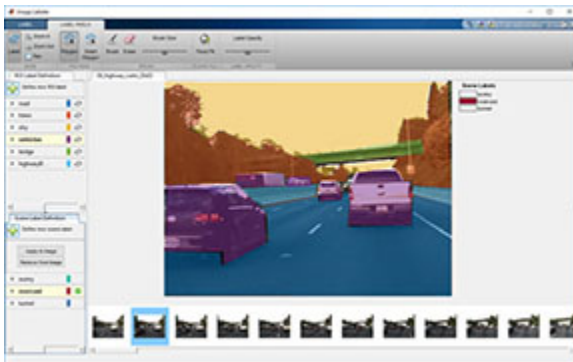


Applications for object detection include:

- Image classification
- Scene understanding
- Self-driving vehicles
- Surveillance

Create Training Data for Object Detection

Use a labeling app to interactively label ground truth data in a video, image sequence, image collection, or custom data source. You can label object detection ground truth using rectangle labels, which define the position and size of the object in the image.



- “Choose an App to Label Ground Truth Data” on page 14-75
- “Training Data for Object Detection and Semantic Segmentation” on page 14-45

Augment and Preprocess Data

Using data augmentation provides a way to use limited data sets for training. Minor changes, such as translation, cropping, or transforming an image, provide, new, distinct, and unique images that you can use to train a robust detector. Datastores are a convenient way to read and augment collections

of data. Use `imageDatastore` and the `boxLabelDatastore` to create datastores for images and labeled bounding box data.

- “Augment Bounding Boxes for Object Detection” (Deep Learning Toolbox)
- “Preprocess Images for Deep Learning” (Deep Learning Toolbox)
- “Preprocess Data for Domain-Specific Deep Learning Applications” (Deep Learning Toolbox)

For more information about augmenting training data using datastores, see “Datastores for Deep Learning” (Deep Learning Toolbox), and “Perform Additional Image Processing Operations Using Built-In Datastores” (Deep Learning Toolbox).

Create Object Detection Network

Each object detector contains a unique network architecture. For example, the Faster R-CNN detector uses a two-stage network for detection, whereas the YOLO v2 detector uses a single stage. Use functions like `fasterRCNNLayers` or `yoloV2Layers` to create a network. You can also design a network layer by layer using the **Deep Network Designer**.

- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)
- “Design a YOLO v2 Detection Network” on page 14-27
- “Design an R-CNN, Fast R-CNN, and a Faster R-CNN Model” on page 14-33

Train Detector and Evaluate Results

Use the `trainFasterRCNNObjectDetector`, `trainYOLOv2ObjectDetector`, `trainSSDObjectDetector` functions to train an object detector. Use the `evaluateDetectionMissRate` and `evaluateDetectionPrecision` functions to evaluate the training results.

- “Train Faster R-CNN Vehicle Detector”
- Train YOLO v2 Object Detector
- “Train SSD Object Detector”

Detect Objects Using Deep Learning Detectors

Detect objects in an image using the trained detector. For example, the partial code shown below uses the trained detector on an image `I`. Use the `detect` object function on `fasterRCNNObjectDetector`, `yoloV2ObjectDetector`, or `ssdObjectDetector` objects to return bounding boxes, detection scores, and categorical labels assigned to the bounding boxes.

```
I = imread(input_image)
[bboxes,scores,labels] = detect(detector,I)
```

- “Object Detection Using YOLO v2 Deep Learning” on page 3-170
- “Object Detection Using SSD Deep Learning” on page 3-23
- “Object Detection Using Faster R-CNN Deep Learning” on page 3-197

See Also

Apps

[Image Labeler](#) | [Video Labeler](#)

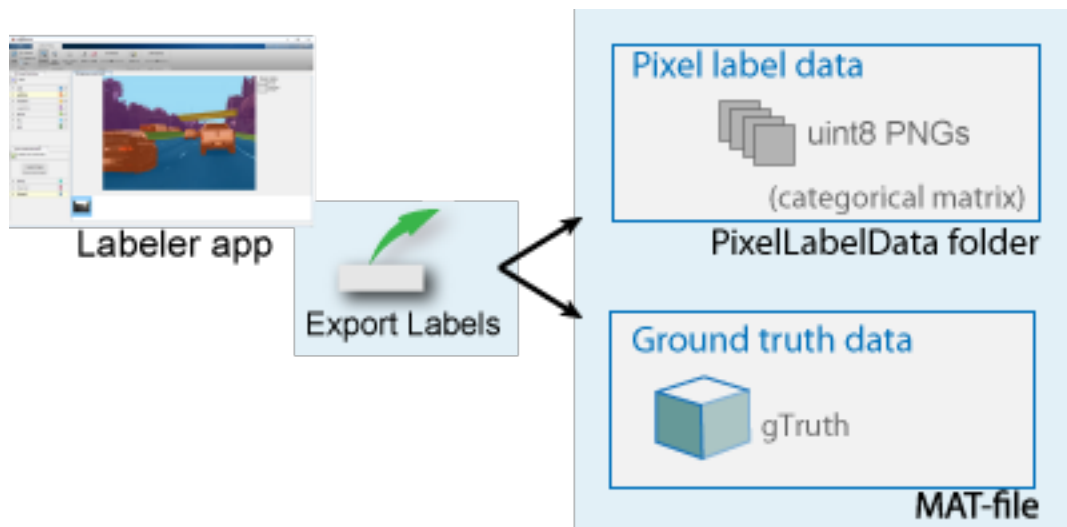
More About

- “Getting Started with YOLO v2” on page 14-26
- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 14-30
- “Getting Started with SSD Multibox Detection” on page 14-9

How Labeler Apps Store Exported Pixel Labels

When you create and export pixel labels from the **Image Labeler**, **Video Labeler**, or **Ground Truth Labeler** (requires Automated Driving Toolbox) app, two sets of data are saved.

- A folder named `PixellabelData`, which contains the PNG files of pixel label information. These labels are encoded as indexed values.
- A MAT-file containing the pixel label data, along with any other label data. This data is stored in a `groundTruth` object, or, if you are using the **Ground Truth Labeler** app, a `groundTruthMultisignal` object. For pixel label data, the object also stores correspondences between image or video frames and the PNG files.



The PNG files within the `PixellabelData` folder are stored as a categorical matrix. The `categorical` matrices contain values assigned to categories. Categorical is a data type. A categorical matrix provides efficient storage and convenient manipulation of nonnumeric data, while also maintaining meaningful names for the values. These matrices are natural representations for semantic segmentation ground truth, where each pixel is one of a predefined category of labels.

Location of Pixel Label Data Folder

The ground truth object stores the folder path and name for the pixel label data folder. The `LabelData` property of the `groundTruth` object or `ROIlabelData` property of the `groundTruthMultisignal` object contains the information in the 'PixelLabelData' column. If you change the location of the pixel data file, you must also update the related information in the ground truth object. You can use the `changeFilePaths` function to update the information.

View Exported Pixel Label Data

The labeler apps store the semantic segmentation ground truth as lossless PNG files, with a `uint8` value representing each category. The app uses the `categorical` function to associate the `uint8` values to a category. To view your pixel data, you can either overlay the categories on images or create a datastore from the labeled images.

View Exported Pixel Label Data By Overlaying Categories on Images

Use the `imread` function with the `categorical` and `labeloverlay` functions. You cannot view the pixel data directly from the categorical matrix. See “View Exported Pixel Label Data” on page 14-17.

View Exported Pixel Label Data from Datastore of Labeled Images

Use the `pixelLabelDatastore` function to create a datastore from a set of labeled images. Use the `read` function to read the pixel label data. See “Read and Display Pixel Label Data” on page 14-18.

Examples

View Exported Pixel Label Data

Read image and corresponding pixel label data that was exported from a labeler app.

```
visiondatadir = fullfile(toolboxdir('vision'),'visiondata');  
  
buildingImage = imread(fullfile(visiondatadir,'building','building1.JPG'));  
buildingLabels = imread(fullfile(visiondatadir,'buildingPixelLabels','Label_1.png'));
```

Define categories for each pixel value in `buildingLabels`.

```
labelIDs = [1,2,3,4];  
labelcats = ["sky" "grass" "building" "sidewalk"];
```

Construct a categorical matrix using the image and the definitions.

```
buildingLabelCats = categorical(buildingLabels,labelIDs,labelcats);
```

Display the categories overlaid on the image.

```
figure  
imshow(labeloverlay(buildingImage,buildingLabelCats))
```



Read and Display Pixel Label Data

Overlay pixel label data on an image.

Set the location of the image and pixel label data.

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

Create an image datastore.

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

Create a pixel label datastore.

```
classNames = ["sky" "grass" "building" "sidewalk"];  
pixelLabelID = [1 2 3 4];  
pxds = pixelLabelDatastore(pxDir,classNames,pixelLabelID);
```

Read the image and pixel label data. `read(pxds)` returns a categorical matrix, C . The element $C(i,j)$ in the matrix is the categorical label assigned to the pixel at the location $l(i,j)$.

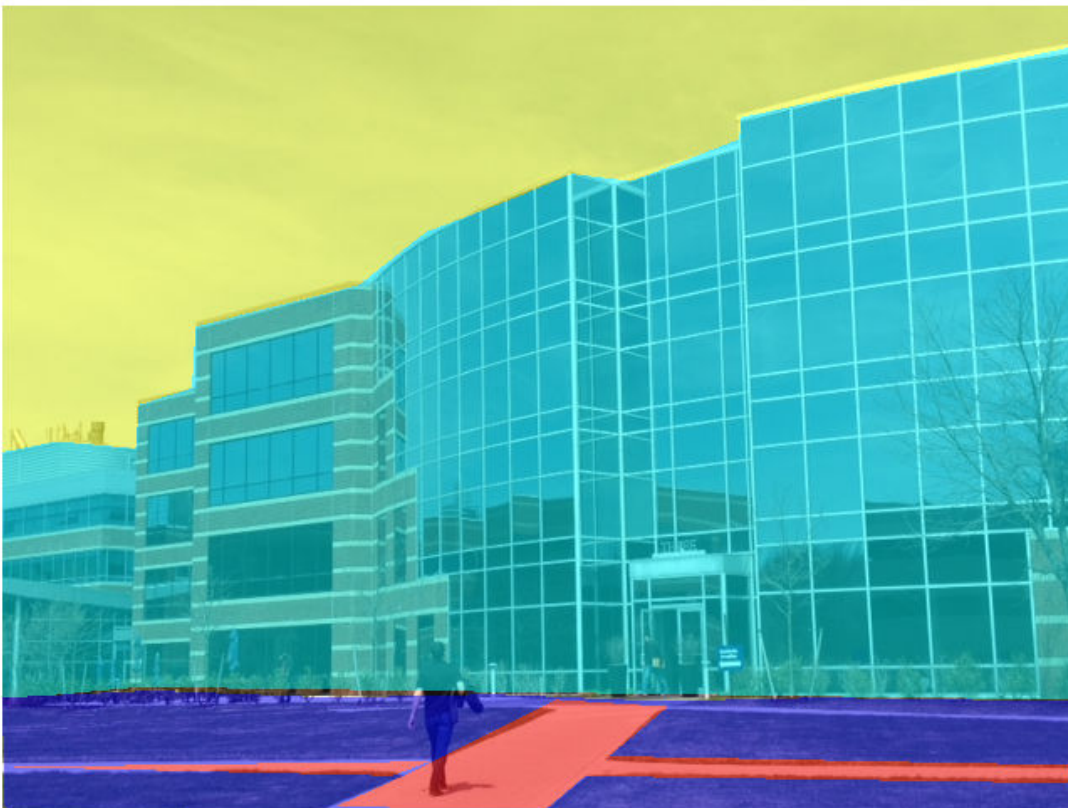
```
I = read(imds);  
C = read(pxds);
```

Display the label categories in C.

```
categories(C{1})  
  
ans = 4x1 cell  
    {'sky'    }  
    {'grass'  }  
    {'building'}  
    {'sidewalk'}
```

Overlay and display the pixel label data onto the image.

```
B = labeloverlay(I,C{1});  
figure  
imshow(B)
```



See Also

Apps

[Ground Truth Labeler](#) | [Image Labeler](#) | [Video Labeler](#)

Objects

[groundTruth](#) | [groundTruthMultisignal](#) | [pixelLabelImageDatastore](#)

Functions

[changeFilePaths \(groundTruth\)](#) | [changeFilePaths \(groundTruthMultisignal\)](#)

More About

- “Label Pixels for Semantic Segmentation” on page 14-53
- “Share and Store Labeled Ground Truth Data” on page 14-106

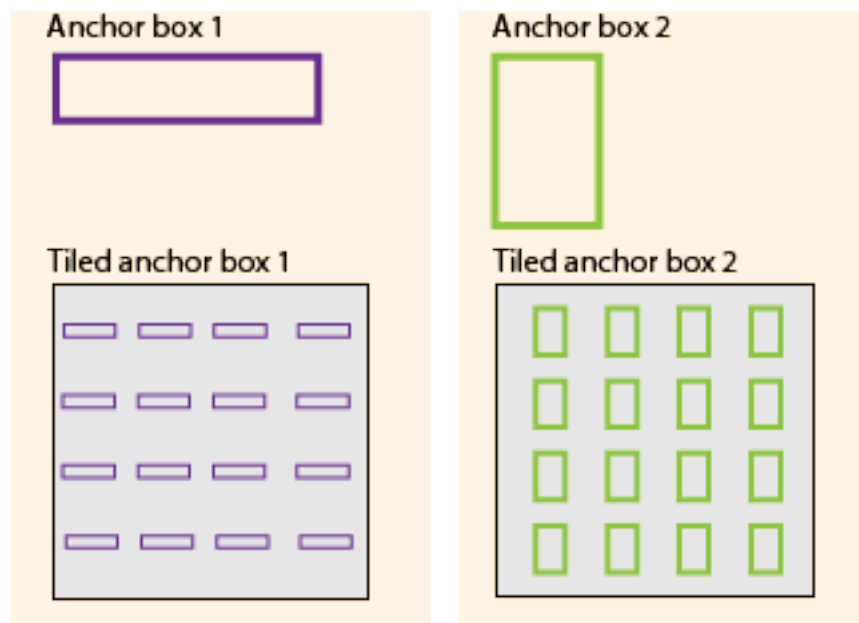
Anchor Boxes for Object Detection

Object detection using deep learning neural networks can provide a fast and accurate means to predict the location and size of an object in an image. Ideally, the network returns valid objects in a timely matter, regardless of the scale of the objects. The use of anchor boxes improves the speed and efficiency for the detection portion of a deep learning neural network framework.

What Is an Anchor Box?

Anchor boxes are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect and are typically chosen based on object sizes in your training datasets. During detection, the predefined anchor boxes are tiled across the image. The network predicts the probability and other attributes, such as background, intersection over union (IoU) and offsets for every tiled anchor box. The predictions are used to refine each individual anchor box. You can define several anchor boxes, each for a different object size. Anchor boxes are fixed initial boundary box guesses.

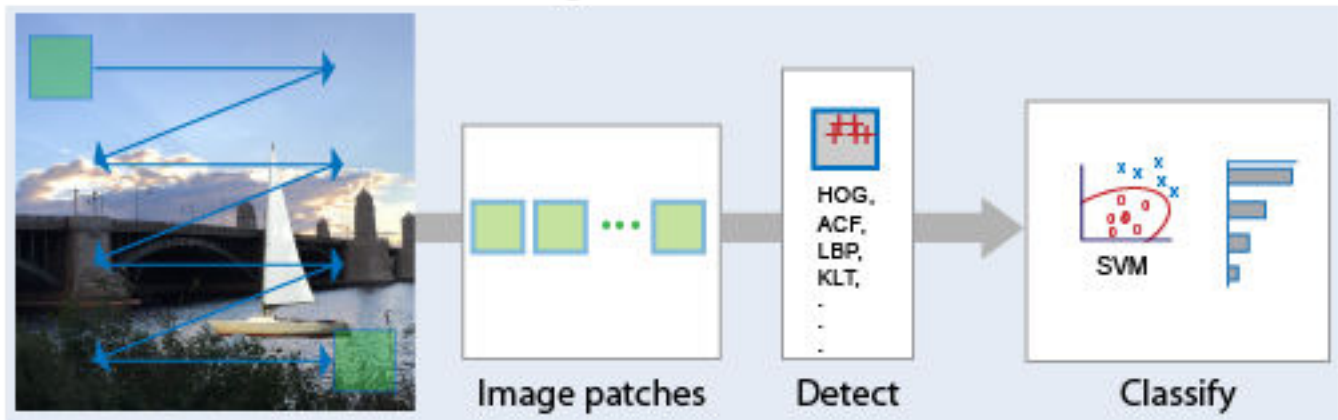
The network does not directly predict bounding boxes, but rather predicts the probabilities and refinements that correspond to the tiled anchor boxes. The network returns a unique set of predictions for every anchor box defined. The final feature map represents object detections for each class. The use of anchor boxes enables a network to detect multiple objects, objects of different scales, and overlapping objects.



Advantage of Using Anchor Boxes

When using anchor boxes, you can evaluate all object predictions at once. Anchor boxes eliminate the need to scan an image with a sliding window that computes a separate prediction at every potential position. Examples of detectors that use a sliding window are those that are based on aggregate channel features (ACF) or histogram of gradients (HOG) features. An object detector that uses anchor boxes can process an entire image at once, making real-time object detection systems possible.

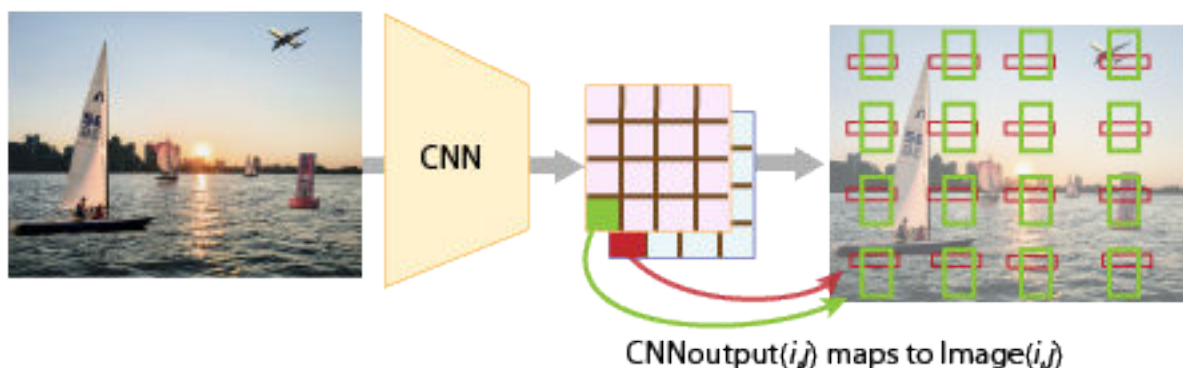
Sliding Window Detector



Because a convolutional neural network (CNN) can process an input image in a convolutional manner, a spatial location in the input can be related to a spatial location in the output. This convolutional correspondence means that a CNN can extract image features for an entire image at once. The extracted features can then be associated back to their location in that image. The use of anchor boxes replaces and drastically reduces the cost of the sliding window approach for extracting features from an image. Using anchor boxes, you can design efficient deep learning object detectors to encompass all three stages (detect, feature encode, and classify) of a sliding-window based object detector.

How Do Anchor Boxes Work?

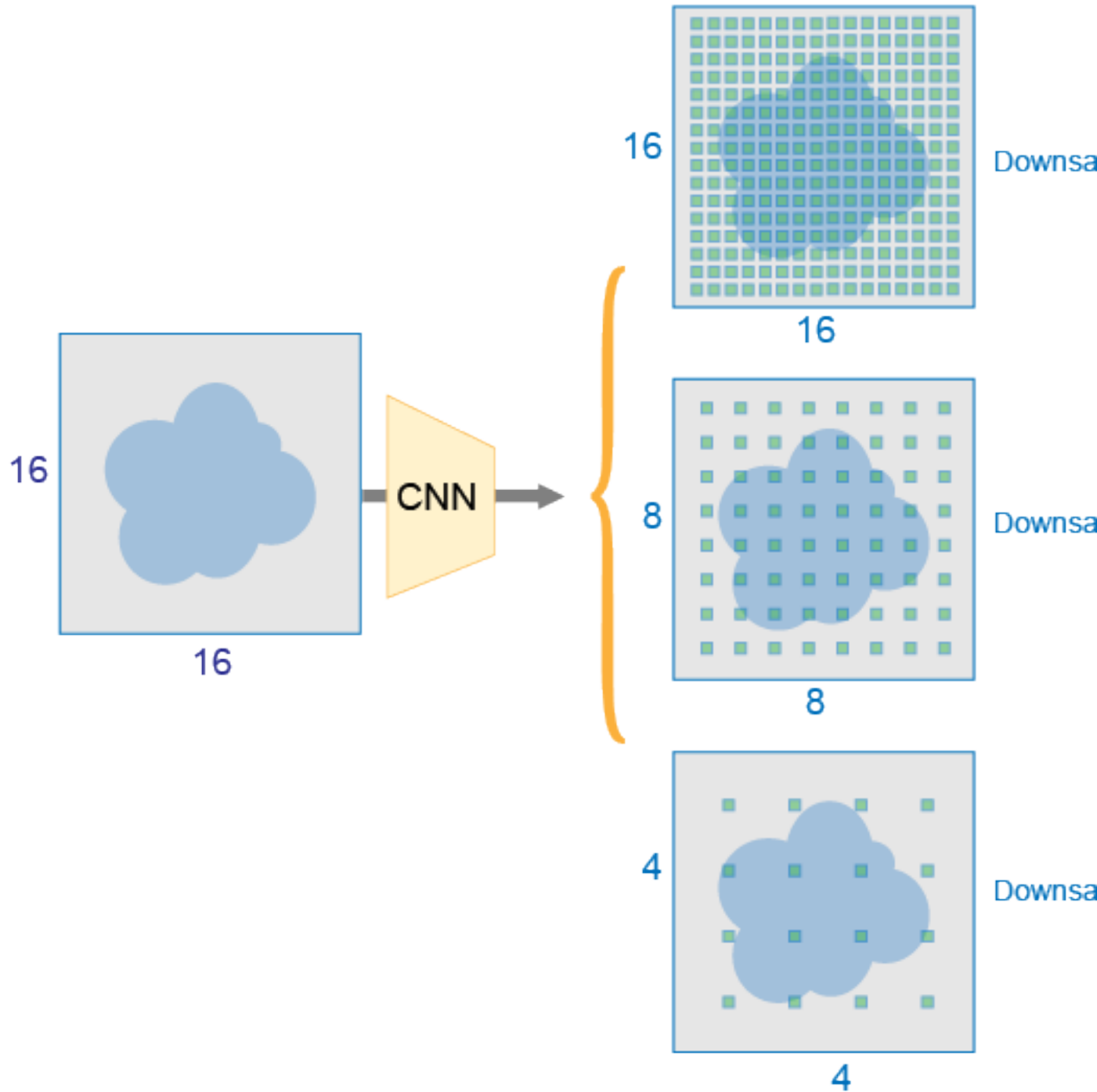
The position of an anchor box is determined by mapping the location of the network output back to the input image. The process is replicated for every network output. The result produces a set of tiled anchor boxes across the entire image. Each anchor box represents a specific prediction of a class. For example, there are two anchor boxes to make two predictions per location in the image below.



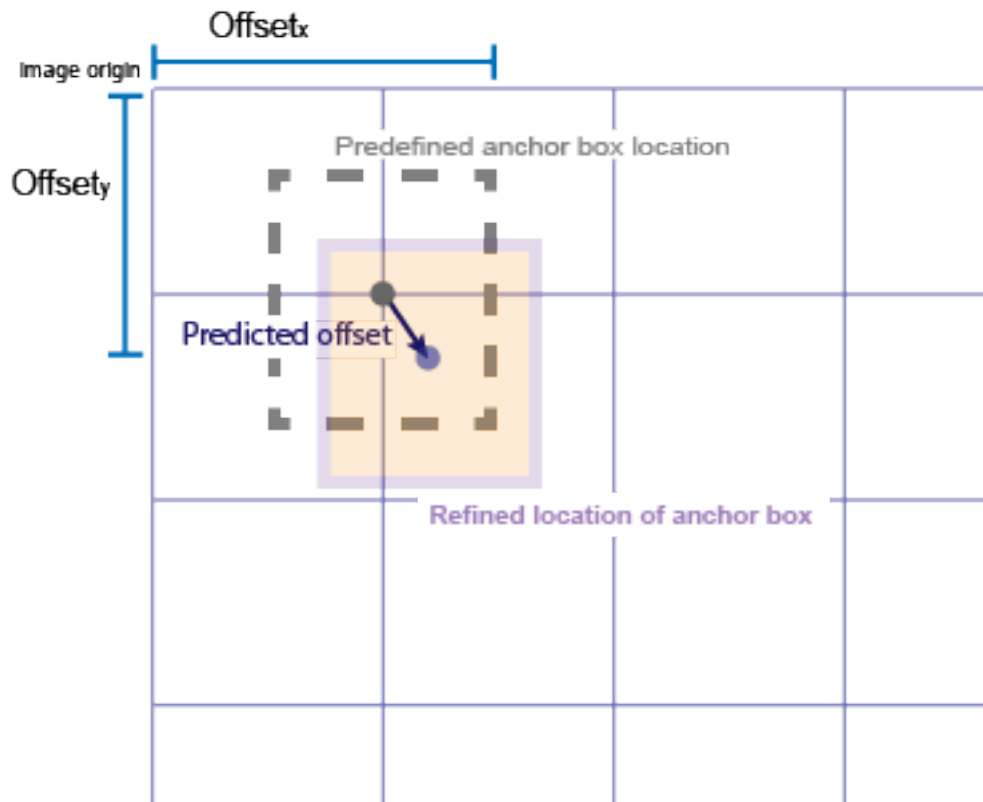
Each anchor box is tiled across the image. The number of network outputs equals the number of tiled anchor boxes. The network produces predictions for all outputs.

Localization Errors and Refinement

The distance, or stride, between the tiled anchor boxes is a function of the amount of downsampling present in the CNN. Downsampling factors between 4 and 16 are common. These downsampling factors produce coarsely tiled anchor boxes, which can lead to localization errors.



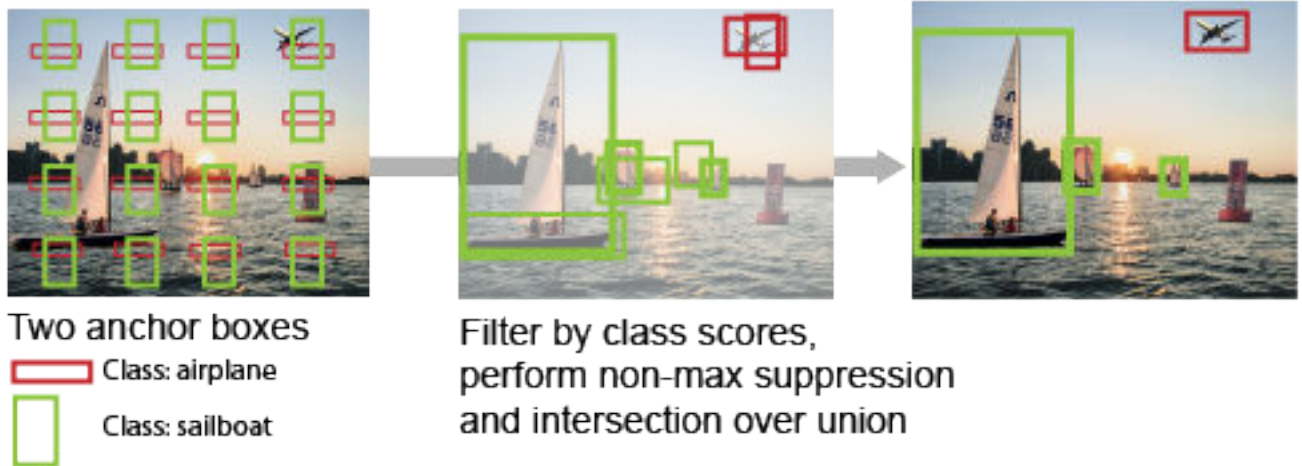
To fix localization errors, deep learning object detectors learn offsets to apply to each tiled anchor box refining the anchor box position and size.



Downsampling can be reduced by removing downsampling layers. To reduce downsampling, lower the 'Stride' property of the convolution or max pooling layers, (such as `convolution2dLayer` and `maxPooling2dLayer`.) You can also choose a feature extraction layer earlier in the network. Feature extraction layers from earlier in the network have higher spatial resolution but may extract less semantic information compared to layers further down the network

Generate Object Detections

To generate the final object detections, tiled anchor boxes that belong to the background class are removed, and the remaining ones are filtered by their confidence score. Anchor boxes with the greatest confidence score are selected using nonmaximum suppression (NMS). For more details about NMS, see the `selectStrongestBboxMulticlass` function.



Anchor Box Size

Multiscale processing enables the network to detect objects of varying size. To achieve multiscale detection, you must specify anchor boxes of varying size, such as 64-by-64, 128-by-128, and 256-by-256. Specify sizes that closely represent the scale and aspect ratio of objects in your training data. For an example of estimating sizes, see [Estimate Anchor Boxes From Training Data](#) on page 3-146.

See Also

Related Examples

- “Create YOLO v2 Object Detection Network” on page 3-180
- “Train Object Detector Using R-CNN Deep Learning” on page 3-183
- “Object Detection Using Faster R-CNN Deep Learning” on page 3-197
- [Estimate Anchor Boxes From Training Data](#) on page 3-146

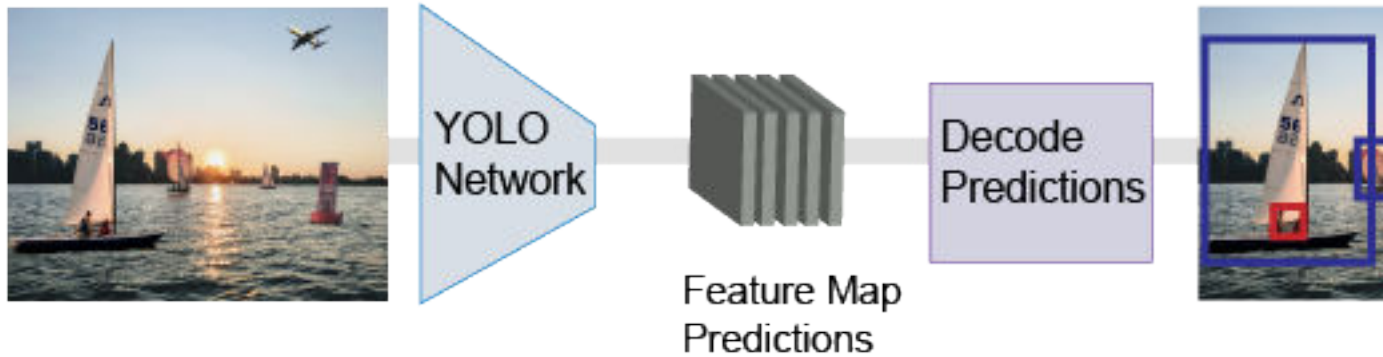
More About

- “Getting Started with YOLO v2” on page 14-26
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

Getting Started with YOLO v2

The you-only-look-once (YOLO) v2 object detector uses a single stage object detection network. YOLO v2 is faster than other two-stage deep learning object detectors, such as regions with convolutional neural networks (Faster R-CNNs).

The YOLO v2 model runs a deep learning CNN on an input image to produce network predictions. The object detector decodes the predictions and generates bounding boxes.

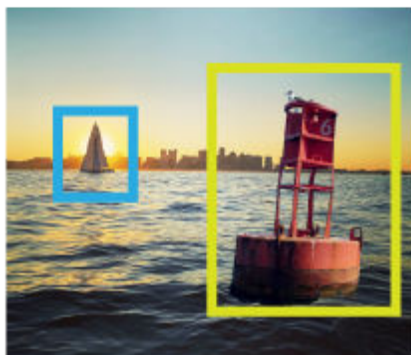


Predicting Objects in the Image

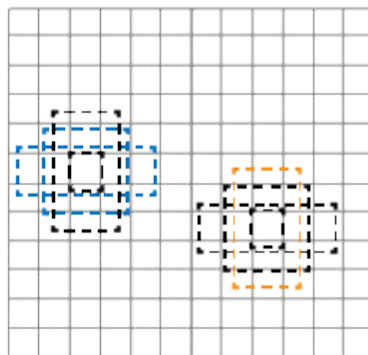
YOLO v2 uses anchor boxes to detect classes of objects in an image. For more details, see “Anchor Boxes for Object Detection” on page 14-21. The YOLO v2 predicts these three attributes for each anchor box:

- Intersection over union (IoU) — Predicts the objectness score of each anchor box.
- Anchor box offsets — Refine the anchor box position
- Class probability — Predicts the class label assigned to each anchor box.

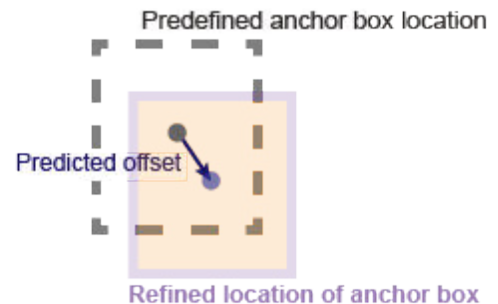
The figure shows predefined anchor boxes (the dotted lines) at each location in a feature map and the refined location after offsets are applied. Matched boxes with a class are in color.



Ground truth image and bounding boxes



Anchor boxes at each predefined location in each feature map



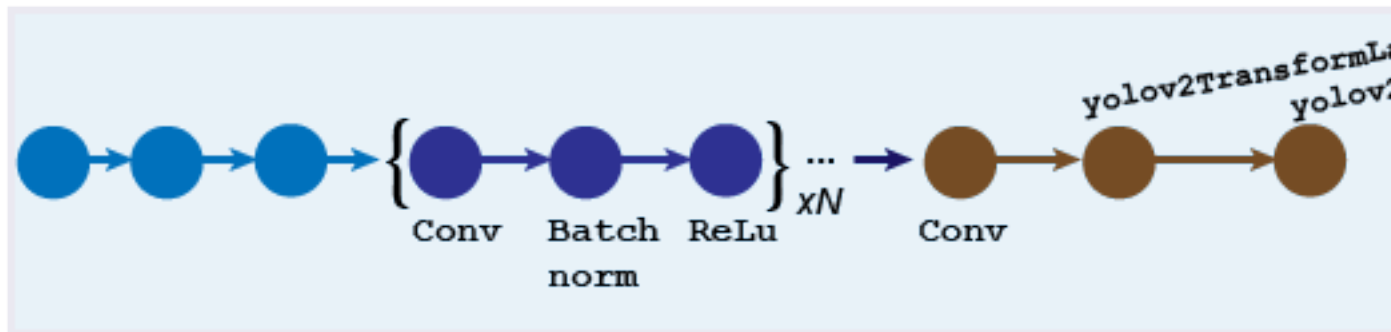
Transfer Learning

With transfer learning, you can use a pretrained CNN as the feature extractor in a YOLO v2 detection network. Use the `yolov2Layers` function to create a YOLO v2 detection network from any pretrained CNN, for example `MobileNet_v2`. For a list of pretrained CNNs, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

You can also design a custom model based on a pretrained image classification CNN. For more details, see “Design a YOLO v2 Detection Network” on page 14-27.

Design a YOLO v2 Detection Network

You can design a custom YOLO v2 model layer by layer. The model starts with a feature extractor network, which can be initialized from a pretrained CNN or trained from scratch. The detection subnetwork contains a series of Conv, Batch norm, and ReLu layers, followed by the transform and output layers, `yolov2TransformLayer` and `yolov2OutputLayer` objects, respectively. `yolov2TransformLayer` transforms the raw CNN output into a form required to produce object detections. `yolov2OutputLayer` defines the anchor box parameters and implements the loss function used to train the detector.



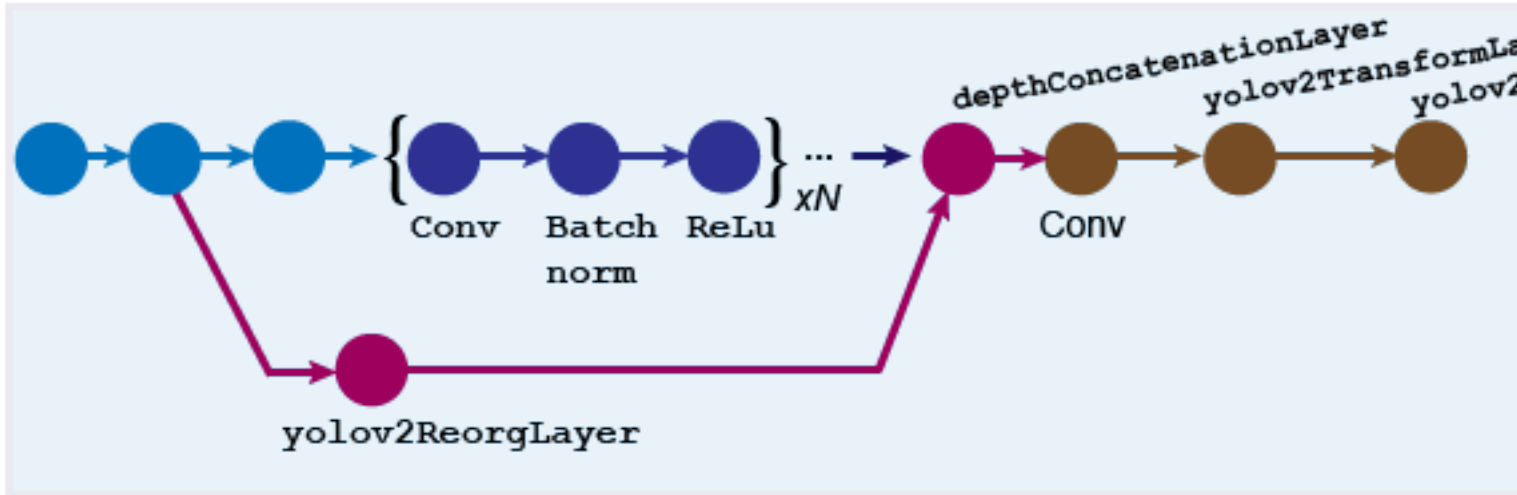
You can also use the **Deep Network Designer** app to manually create a network. The designer incorporates Computer Vision Toolbox YOLO v2 features.

Design a YOLO v2 Detection Network with a Reorg Layer

The reorganization layer (created using the `spaceToDepthLayer` object) and the depth concatenation layer (created using the `depthConcatenationLayer` object) are used to combine low-level and high-level features. These layers improve detection by adding low-level image information and improving detection accuracy for smaller objects. Typically, the reorganization layer is attached to a layer within the feature extraction network whose output feature map is larger than the feature extraction layer output.

Tip

- Adjust the 'BlockSize' property of the `spaceToDepthLayer` object such that its output size matches the input size of the `depthConcatenationLayer` object.
 - To simplify designing a network, use the interactive **Deep Network Designer** app and the `analyzeNetwork` function.
-



For more details on how to create this kind of network, see “Create YOLO v2 Object Detection Network” on page 3-180.

Train an Object Detector and Detect Objects with a YOLO v2 Model

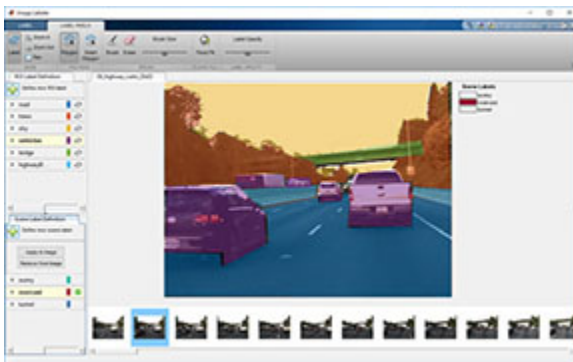
To learn how to train an object detector by using the YOLO deep learning technique with a CNN, see the “Object Detection Using YOLO v2 Deep Learning” on page 3-170 example.

Code Generation

To learn how to generate CUDA code using the YOLO v2 object detector (created using the `yolov2ObjectDetector` object) see “Code Generation for Object Detection by Using YOLO v2” on page 2-5.

Label Training Data for Deep Learning

You can use the **Image Labeler**, **Video Labeler**, or **Ground Truth Labeler** (available in Automated Driving Toolbox) apps to interactively label pixels and export label data for training. The apps can also be used to label rectangular regions of interest (ROIs) for object detection, scene labels for image classification, and pixels for semantic segmentation.



References

- [1] Redmon, J. and A. Farhadi. "YOLO9000: Better, Faster, Stronger." *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6517–6525. Honolulu, HI: CVPR 2017.
- [2] Redmon, J., S. Divvala, R. Girshick, and A. Farhadi. "You only look once: Unified, real-time object detection." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779–788. Las Vegas, NV: CVPR, 2016.

See Also

Apps

[Deep Network Designer](#) | [Ground Truth Labeler](#) | [Image Labeler](#) | [Video Labeler](#)

Objects

[depthConcatenationLayer](#) | [spaceToDepthLayer](#) | [yolov2objectDetector](#) | [yolov2outputLayer](#) | [yolov2transformLayer](#)

Functions

[analyzeNetwork](#) | [trainYOLOv2objectDetector](#)

Related Examples

- "Train Object Detector Using R-CNN Deep Learning" on page 3-183
- "Object Detection Using YOLO v2 Deep Learning" on page 3-170
- "Code Generation for Object Detection by Using YOLO v2" on page 2-5

More About

- "Anchor Boxes for Object Detection" on page 14-21
- "Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN" on page 14-30
- "Deep Learning in MATLAB" (Deep Learning Toolbox)
- "Pretrained Deep Neural Networks" (Deep Learning Toolbox)

Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN

Object detection is the process of finding and classifying objects in an image. One deep learning approach, regions with convolutional neural networks (R-CNN), combines rectangular region proposals with convolutional neural network features. R-CNN is a two-stage detection algorithm. The first stage identifies a subset of regions in an image that might contain an object. The second stage classifies the object in each region.

Applications for R-CNN object detectors include:

- Autonomous driving
- Smart surveillance systems
- Facial recognition

Computer Vision Toolbox provides object detectors for the R-CNN, Fast R-CNN, and Faster R-CNN algorithms.

Instance segmentation expands on object detection to provide pixel-level segmentation of individual detected objects. Computer Vision Toolbox provides layers that support a deep learning approach for instance segmentation called Mask R-CNN. For more information, see “Getting Started with Mask R-CNN for Instance Segmentation” on page 14-36.

Object Detection Using R-CNN Algorithms

Models for object detection using regions with CNNs are based on the following three processes:

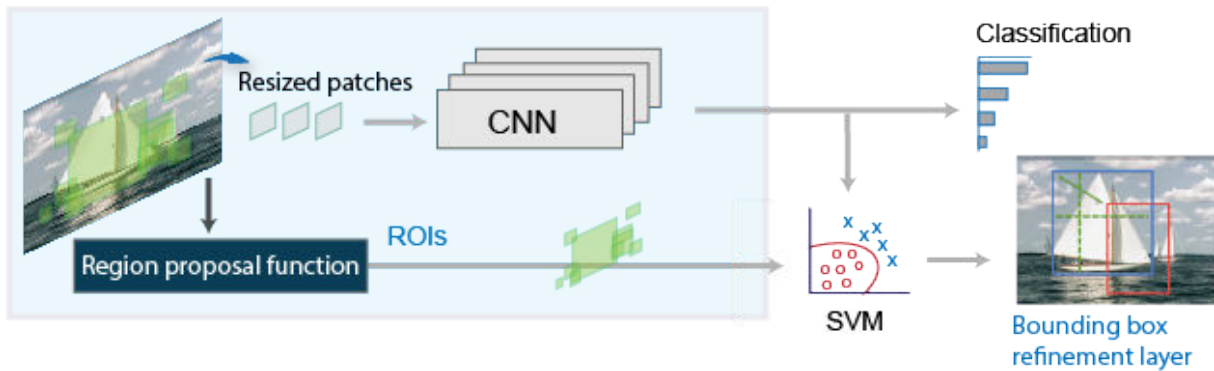
- Find regions in the image that might contain an object. These regions are called region proposals.
- Extract CNN features from the region proposals.
- Classify the objects using the extracted features.

There are three variants of an R-CNN. Each variant attempts to optimize, speed up, or enhance the results of one or more of these processes.

R-CNN

The R-CNN detector [2] first generates region proposals using an algorithm such as Edge Boxes[1]. The proposal regions are cropped out of the image and resized. Then, the CNN classifies the cropped and resized regions. Finally, the region proposal bounding boxes are refined by a support vector machine (SVM) that is trained using CNN features.

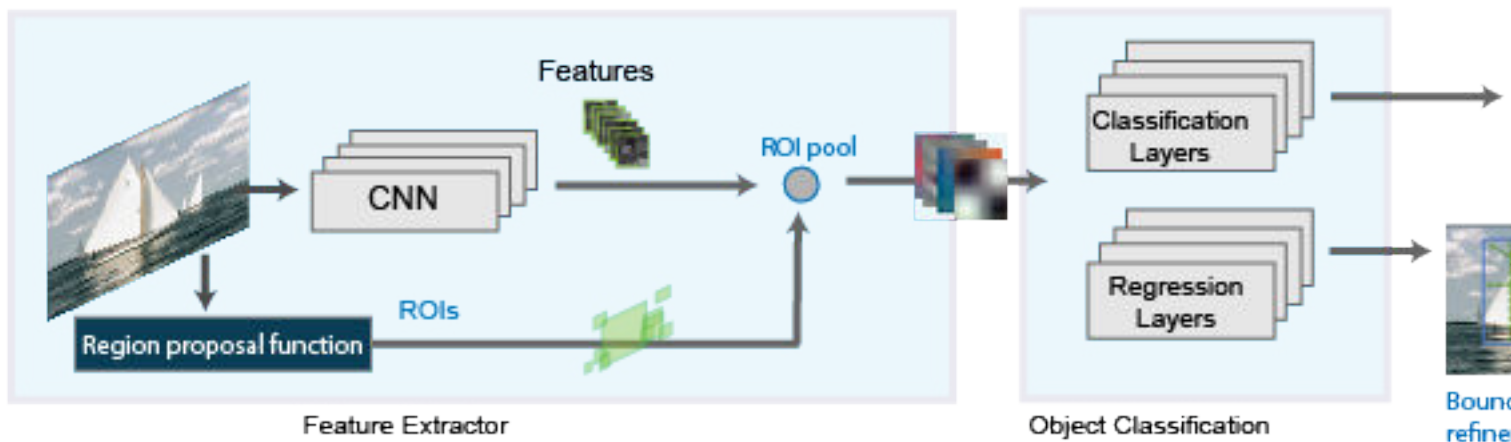
Use the `trainRCNNObjectDetector` function to train an R-CNN object detector. The function returns an `rcnnObjectDetector` object that detects objects in an image.



Fast R-CNN

As in the R-CNN detector, the Fast R-CNN[3] detector also uses an algorithm like Edge Boxes to generate region proposals. Unlike the R-CNN detector, which crops and resizes region proposals, the Fast R-CNN detector processes the entire image. Whereas an R-CNN detector must classify each region, Fast R-CNN pools CNN features corresponding to each region proposal. Fast R-CNN is more efficient than R-CNN, because in the Fast R-CNN detector, the computations for overlapping regions are shared.

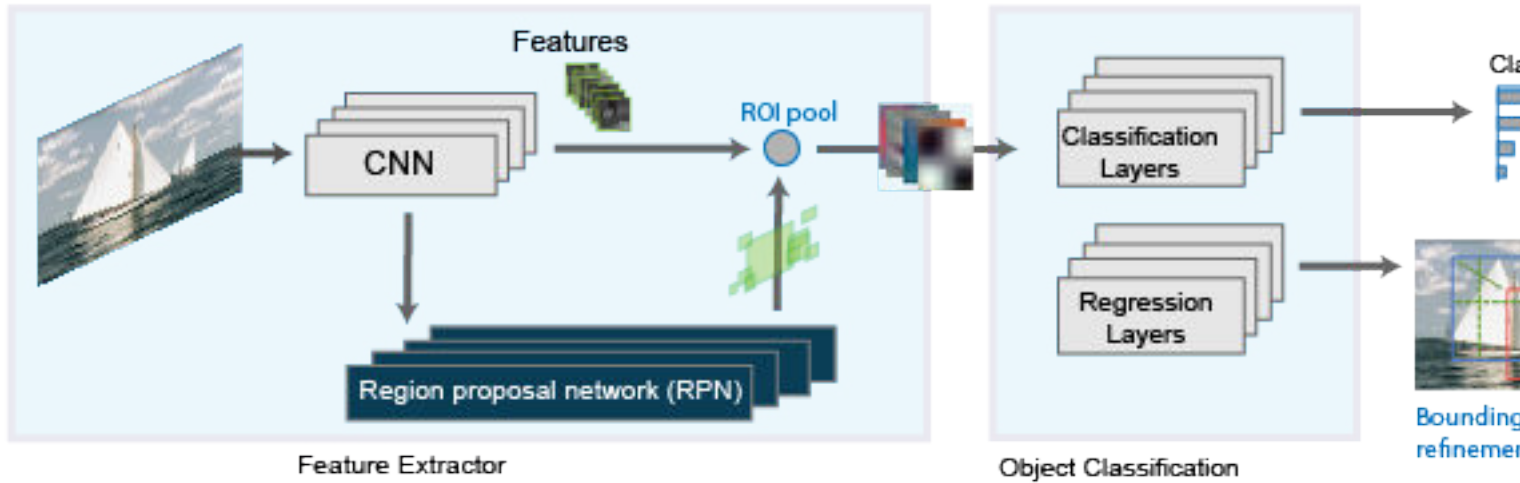
Use the `trainFastRCNNObjectDetector` function to train a Fast R-CNN object detector. The function returns a `fastRCNNObjectDetector` that detects objects from an image.



Faster R-CNN

The Faster R-CNN[4] detector adds a region proposal network (RPN) to generate region proposals directly in the network instead of using an external algorithm like Edge Boxes. The RPN uses “Anchor Boxes for Object Detection” on page 14-21. Generating region proposals in the network is faster and better tuned to your data.

Use the `trainFasterRCNNObjectDetector` function to train a Faster R-CNN object detector. The function returns a `fasterRCNNObjectDetector` that detects objects from an image.



Comparison of R-CNN Object Detectors

This family of object detectors uses region proposals to detect objects within images. The number of proposed regions dictates the time it takes to detect objects in an image. The Fast R-CNN and Faster R-CNN detectors are designed to improve detection performance with a large number of regions.

R-CNN Detector	Description
<code>trainRCNNObjectDetector</code>	<ul style="list-style-type: none"> • Slow training and detection • Allows custom region proposal
<code>trainFastRCNNObjectDetector</code>	<ul style="list-style-type: none"> • Allows custom region proposal
<code>trainFasterRCNNObjectDetector</code>	<ul style="list-style-type: none"> • Optimal run-time performance • Does not support a custom region proposal

Transfer Learning

You can use a pretrained convolution neural network (CNN) as the basis for an R-CNN detector, also referred to as transfer learning. See “Pretrained Deep Neural Networks” (Deep Learning Toolbox). Use one of the following networks with the `trainRCNNObjectDetector`, `trainFasterRCNNObjectDetector`, or `trainFastRCNNObjectDetector` functions. To use any of these networks you must install the corresponding Deep Learning Toolbox™ model:

- 'alexnet'
- 'vgg16'
- 'vgg19'
- 'resnet50'
- 'resnet101'
- 'inceptionv3'
- 'googlenet'
- 'inceptionresnetv2'

- 'squeezenet'

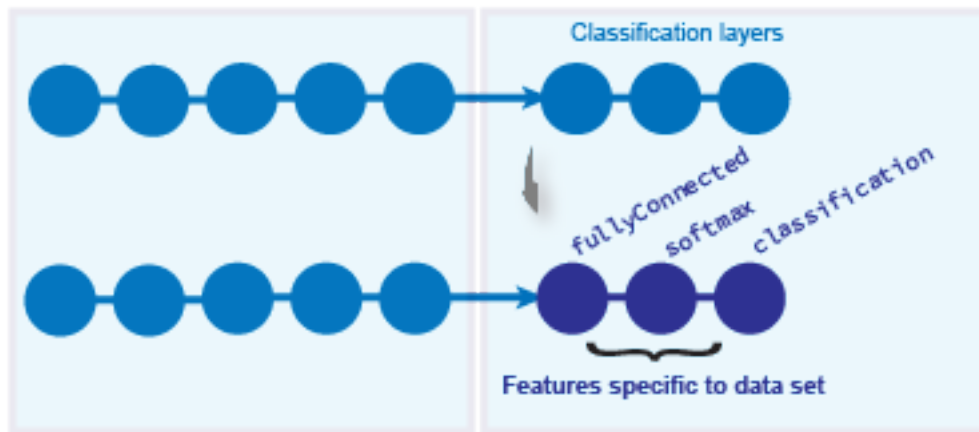
You can also design a custom model based on a pretrained image classification CNN. See the “Design an R-CNN, Fast R-CNN, and a Faster R-CNN Model” on page 14-33 section and the **Deep Network Designer** app.

Design an R-CNN, Fast R-CNN, and a Faster R-CNN Model

You can design custom R-CNN models based on a pretrained image classification CNN. You can also use the **Deep Network Designer** to build, visualize, and edit a deep learning network.

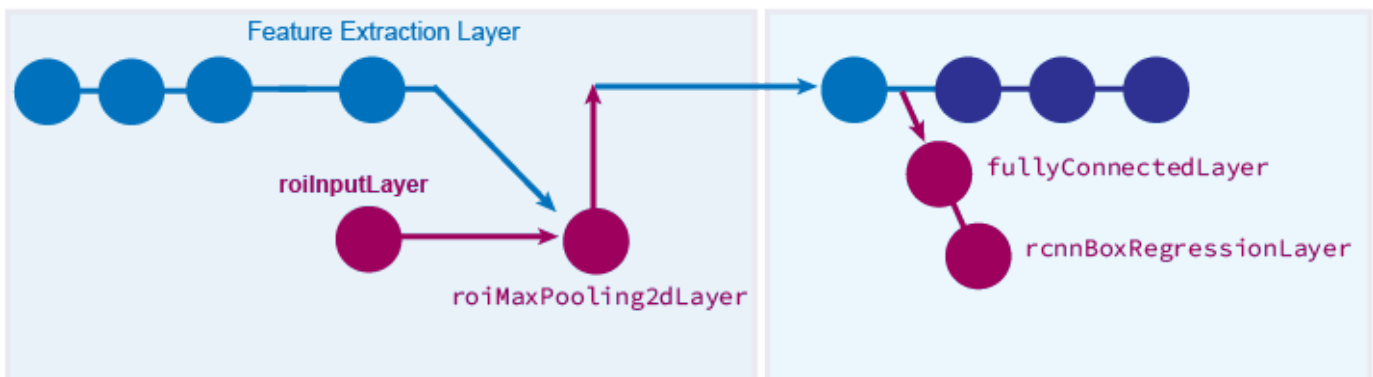
- 1 The basic R-CNN model starts with a pretrained network. The last three classification layers are replaced with new layers that are specific to the object classes you want to detect.

For an example of how to create an R-CNN object detection network, see “Create R-CNN Object Detection Network”



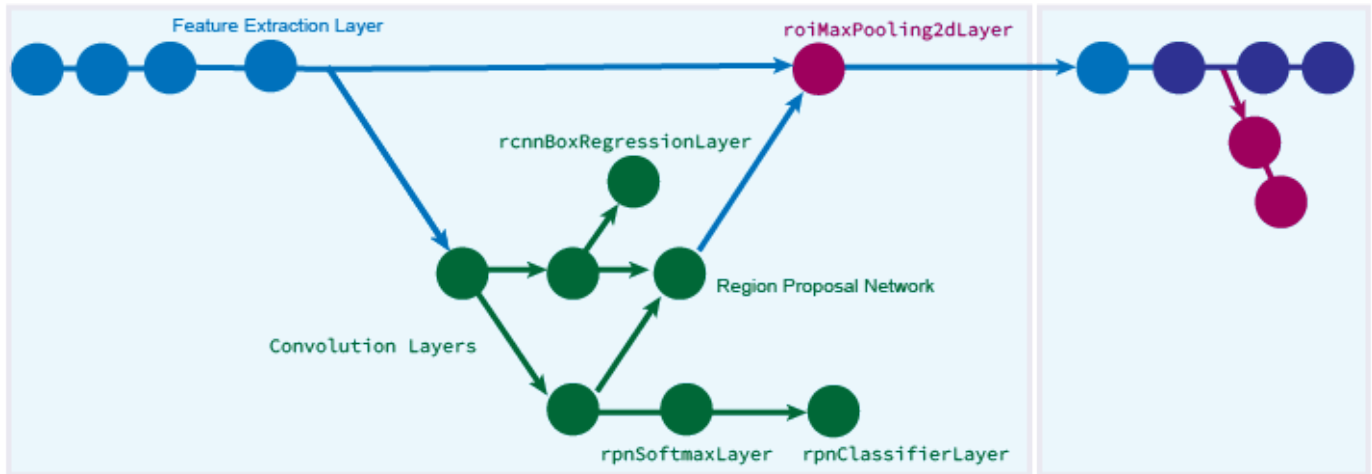
- 2 The Fast R-CNN model builds on the basic R-CNN model. A box regression layer is added to improve on the position of the object in the image by learning a set of box offsets. An ROI pooling layer is inserted into the network to pool CNN features for each region proposal.

For an example of how to create a Fast R-CNN object detection network, see “Create Fast R-CNN Object Detection Network”



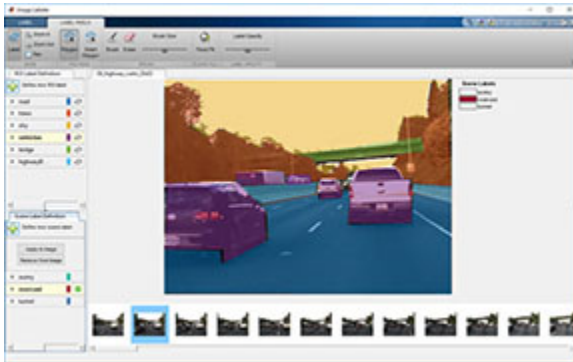
- 3 The Faster R-CNN model builds on the Fast R-CNN model. A region proposal network is added to produce the region proposals instead of getting the proposals from an external algorithm.

For an example of how to create a Faster R-CNN object detection network, see “Create Faster R-CNN Object Detection Network”



Label Training Data for Deep Learning

You can use the **Image Labeler**, **Video Labeler**, or **Ground Truth Labeler** (available in Automated Driving Toolbox) apps to interactively label pixels and export label data for training. The apps can also be used to label rectangular regions of interest (ROIs) for object detection, scene labels for image classification, and pixels for semantic segmentation.



References

- [1] Zitnick, C. Lawrence, and P. Dollar. "Edge boxes: Locating object proposals from edges." *Computer Vision-ECCV*. Springer International Publishing. Pages 391-4050. 2014.
- [2] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *CVPR '14 Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Pages 580-587. 2014
- [3] Girshick, Ross. "Fast r-cnn." *Proceedings of the IEEE International Conference on Computer Vision*. 2015

- [4] Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *Advances in Neural Information Processing Systems* . Vol. 28, 2015.

See Also

Apps

Deep Network Designer | **Ground Truth Labeler** | **Image Labeler** | **Video Labeler**

Functions

`fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `rcnnObjectDetector` |
`trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` |
`trainRCNNObjectDetector`

Related Examples

- "Train Object Detector Using R-CNN Deep Learning" on page 3-183
- "Object Detection Using Faster R-CNN Deep Learning" on page 3-197

More About

- "Anchor Boxes for Object Detection" on page 14-21
- "Deep Learning in MATLAB" (Deep Learning Toolbox)
- "Pretrained Deep Neural Networks" (Deep Learning Toolbox)

Getting Started with Mask R-CNN for Instance Segmentation


Instance segmentation is an enhanced type of object detection that generates a segmentation map for each detected instance of an object. Instance segmentation treats individual objects as distinct entities, regardless of the class of the objects. In contrast, semantic segmentation considers all objects of the same class as belonging to a single entity.

Several deep learning algorithms exist to perform instance segmentation. One popular algorithm is Mask R-CNN, which expands on the Faster R-CNN network to perform pixel-level segmentation on the detected objects [1]. The Mask R-CNN algorithm can accommodate multiple classes and overlapping objects.

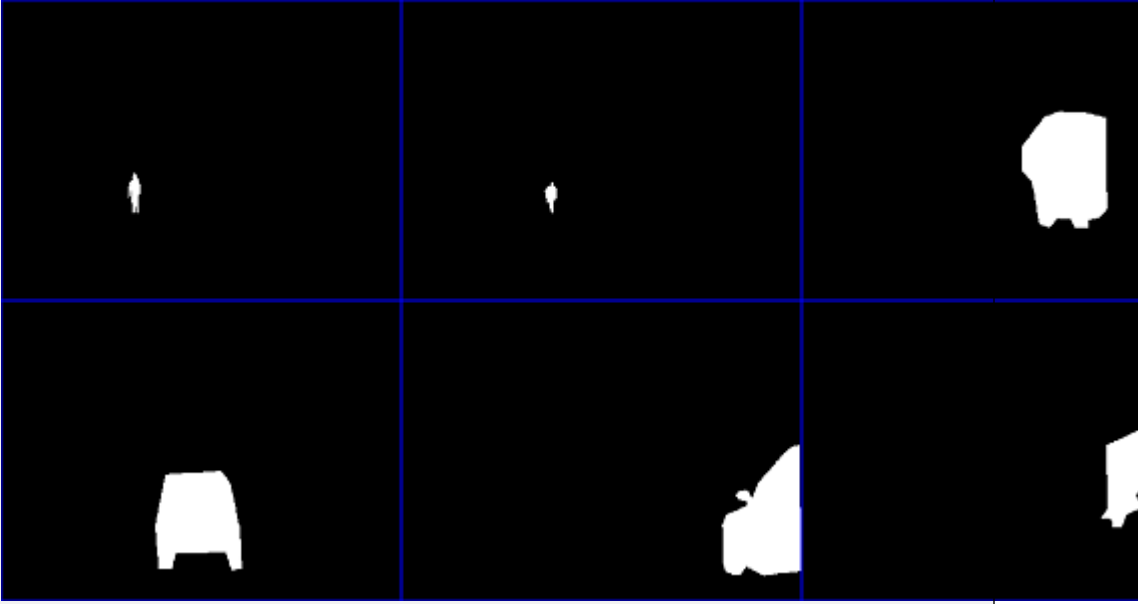
For an example that shows how to train a Mask R-CNN using Computer Vision Toolbox, see Multiclass Instance Segmentation using Mask R-CNN.

Training Data

To train a Mask R-CNN, you need the following data.

Data	Description
RGB image	<p data-bbox="492 915 1471 978">RGB images that serve as network inputs, specified as H-by-W-by-3 numeric arrays.</p> <p data-bbox="492 999 1471 1062">For example, this sample RGB image is a modified image from the CamVid data set [2] that has been edited to remove personally identifiable information.</p> 

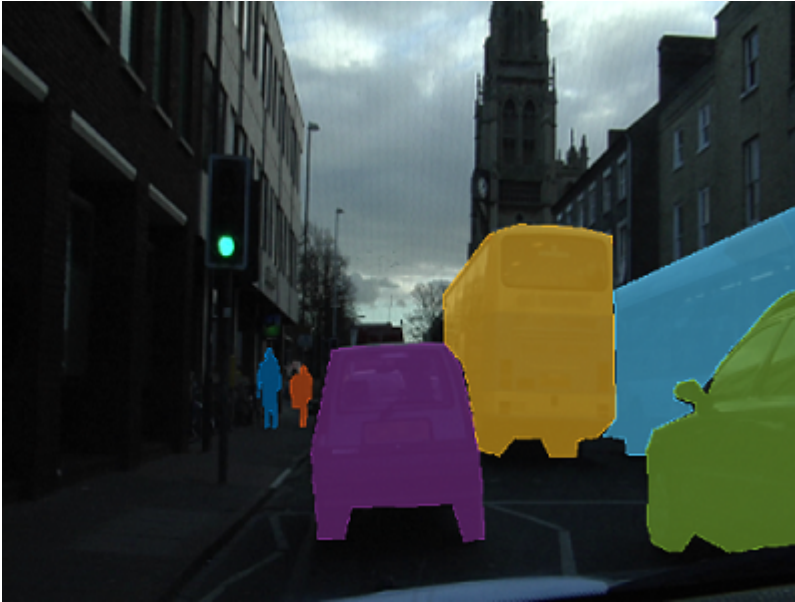
Data	Description
Ground-truth bounding boxes	<p>Bounding boxes for objects in the RGB images, specified as a <i>NumObjects</i>-by-4 matrix, with rows in the format [<i>x y w h</i>]).</p> <p>For example, the <code>bboxes</code> variable shows the bounding boxes of six objects in the sample RGB image.</p> <pre>bboxes = 394 442 36 101 436 457 32 88 619 293 209 281 460 441 210 234 862 375 190 314 816 271 235 305</pre>
Instance labels	<p>Label of each instance, specified as a <i>NumObjects</i>-by-1 string vector or a <i>NumObjects</i>-by-1 cell array of character vectors.)</p> <p>For example, the <code>labels</code> variable shows the labels of six objects in the sample RGB image.</p> <pre>labels = 6×1 cell array {'Person' } {'Person' } {'Vehicle'} {'Vehicle'} {'Vehicle'} {'Vehicle'}</pre>

Data	Description
Instance masks	<p>Masks for instances of objects. Mask data comes in two formats:</p> <ul style="list-style-type: none"> • Binary masks, specified as a logical array of size H-by-W-by-$NumObjects$. Each mask is the segmentation of one instance in the image. • Polygon coordinates, specified as a $NumObjects$-by-2 cell array. Each row of the array contains the (x,y) coordinates of a polygon along the boundary of one instance in the image. <p>The Mask R-CNN network requires binary masks, not polygon coordinates. To convert polygon coordinates to binary masks, use the <code>poly2mask</code> function. The <code>poly2mask</code> function sets pixels that are inside the polygon to 1 and sets pixels outside the polygon to 0. This code shows how to convert polygon coordinates in the <code>masks_polygon</code> variable to binary masks of size h-by-w-by-$numObjects$.</p> <pre> denseMasks = false([h,w,numObjects]); for i = 1:numObjects denseMasks(:,:,i) = poly2mask(masks_polygon{i}(:,1),masks_polygon{i}(:,2),h,w); end </pre> <p>For example, this montage shows the binary masks of six objects in the sample RGB image.</p> 

Visualize Training Data

To display the instance masks over the image, use the `insertObjectMask`. You can specify a color map so that each instance appears in a different color. This sample code shows how display the instance masks in the `masks` variable over the RGB image in the `im` variable using the `lines` color map.

```
imOverlay = insertObjectMask(im,masks,'Color',lines(numObjects));  
imshow(imOverlay);
```



To show the bounding boxes with labels over the image, use the `showShape` function. This sample code shows how to show labeled rectangular shapes with bounding box size and position data in the `bboxes` variable and label data in the `labels` variable.

```
imshow(imOverlay)  
showShape("rectangle",bboxes,"Label",labels,"Color","red");
```



Preprocess Data

Format and Resize Data

Use a datastore to read data. The datastore must return data as a 1-by-4 cell array in the format {RGB images, bounding boxes, labels, masks}. The size of the images, bounding boxes, and masks must match the input size of the network. If you need to resize the data, then you can use the `imresize` to resize the RGB images and masks, and the `bbboxresize` function to resize the bounding boxes.

For more information, see “Datastores for Deep Learning” (Deep Learning Toolbox).

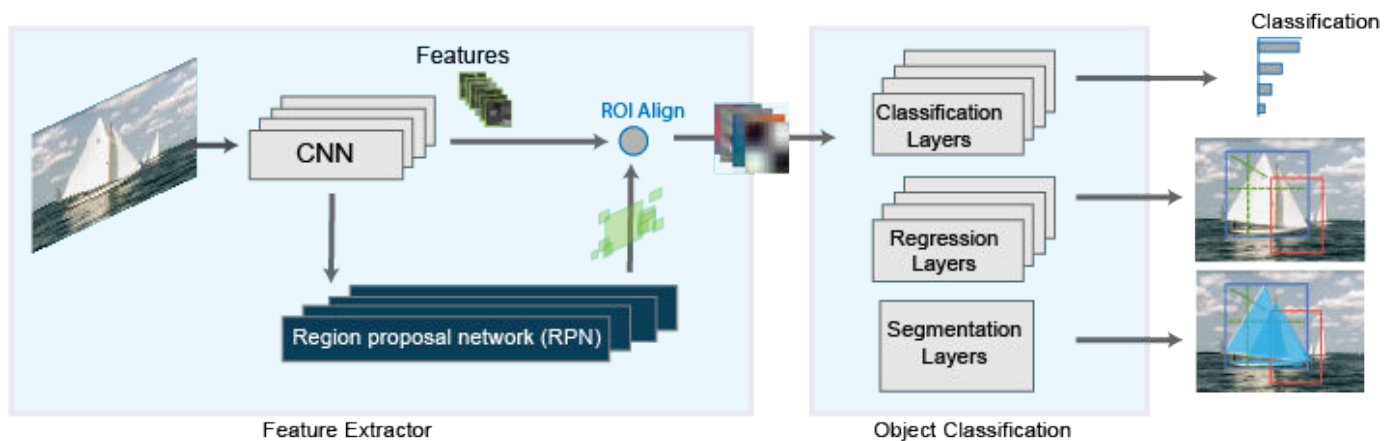
Form Mini-Batches of Data

Training a Mask R-CNN network requires a custom training loop. To manage the mini-batching of observations in a custom training loop, create a `minibatchqueue` object from the datastore. The `minibatchqueue` object casts data to a `darray` object that enables auto differentiation in deep learning applications. If you have a supported GPU, then a `minibatchqueue` object also moves data to the GPU.

The next function yields the next mini-batch of data from the `minibatchqueue`.

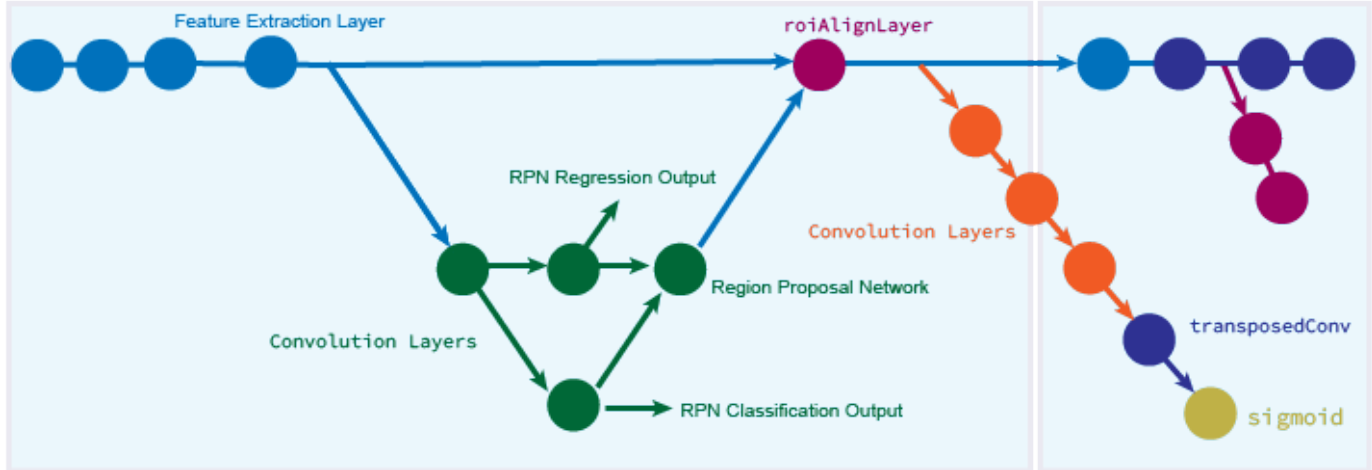
Mask R-CNN Network Architecture

The Mask R-CNN network consists of two stages. The first is a region proposal network (RPN), which predicts object proposal bounding boxes based on anchor boxes. The second stage is an R-CNN detector that refines these proposals, classifies them, and computes the pixel-level segmentation for these proposals.



The Mask R-CNN model builds on the Faster R-CNN model, which you can create using `fasterRCNNLayers`. Replace the ROI max pooling layer with an `roiAlignLayer` that provides more accurate sub-pixel level ROI pooling. The Mask R-CNN network also adds a mask branch for pixel level object segmentation. For more information about the Faster R-CNN network, see “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 14-30.

This diagram shows a modified Faster R-CNN network on the left and a mask branch on the right.



Train Mask R-CNN Network

Train the model in a custom training loop. For each iteration:

- Read the data for current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and a custom helper function that calculates the gradients and overall loss for batches of training data.
- Update the network learnable parameters using a function such as `adamupdate` or `sgdupdate`.

For an example that shows how to train a Mask R-CNN using Computer Vision Toolbox, see [Multiclass Instance Segmentation using Mask R-CNN](#).

References

- [1] He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. "Mask R-CNN." *ArXiv:1703.06870 [Cs]*, January 24, 2018. <https://arxiv.org/pdf/1703.06870>.
- [2] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A High-Definition Ground Truth Database." *Pattern Recognition Letters* 30, no. 2 (January 2009): 88-97. <https://doi.org/10.1016/j.patrec.2008.04.005>.

See Also

Apps
Image Labeler

Functions
fasterRCNNLayers

Objects
minibatchqueue | roiAlignLayer

See Also

More About

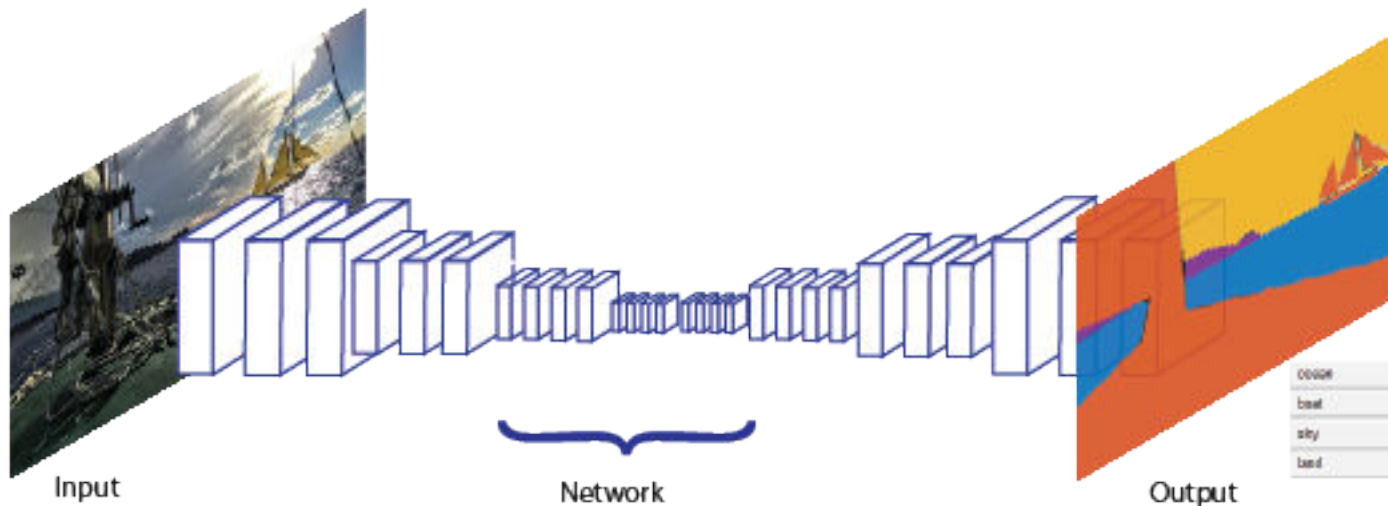
- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 14-30
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Datastores for Deep Learning” (Deep Learning Toolbox)
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)
- “Specify Training Options in Custom Training Loop” (Deep Learning Toolbox)
- “Train Network Using Custom Training Loop” (Deep Learning Toolbox)

External Websites

- Multiclass Instance Segmentation using Mask R-CNN

Getting Started with Semantic Segmentation Using Deep Learning

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

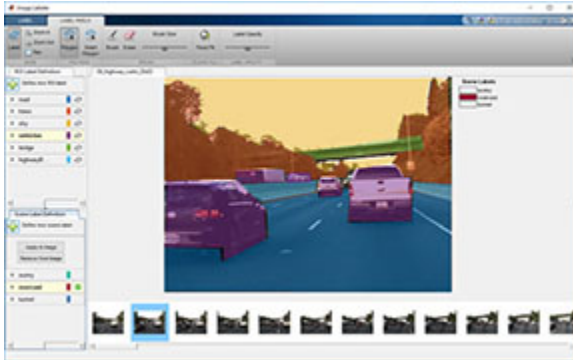
The steps for training a semantic segmentation network are as follows:

1. "Analyze Training Data for Semantic Segmentation"
2. "Create a Semantic Segmentation Network"
3. "Train A Semantic Segmentation Network"
4. "Evaluate and Inspect the Results of Semantic Segmentation"

Label Training Data for Semantic Segmentation

Large datasets enable faster and more accurate mapping to a particular input (or input aspect). Using data augmentation provides a means of leveraging limited datasets for training. Minor changes, such as translation, cropping, or transforming an image provides new distinct and unique images. See "Augment Images for Deep Learning Workflows Using Image Processing Toolbox" (Deep Learning Toolbox)

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

Apps

Image Labeler

Functions

`evaluateSemanticSegmentation` | `fcnLayers` | `pixelLabelDatastore` | `segnetLayers` | `semanticSegmentationMetrics` | `semanticseg` | `unet3dLayers` | `unetLayers`

Objects

`pixelClassificationLayer` | `pixelLabelImageDatastore`

See Also

Related Examples

- “Augment Pixel Labels for Semantic Segmentation” (Deep Learning Toolbox)
- “Import Pixel Labeled Dataset For Semantic Segmentation”
- “Semantic Segmentation Using Deep Learning” on page 3-43
- “Label Pixels for Semantic Segmentation” on page 14-53
- “Define Custom Pixel Classification Layer with Tversky Loss” on page 8-58
- “Semantic Segmentation Using Dilated Convolutions” on page 8-54
- “Calculate Segmentation Metrics in Block-Based Workflow” on page 3-59

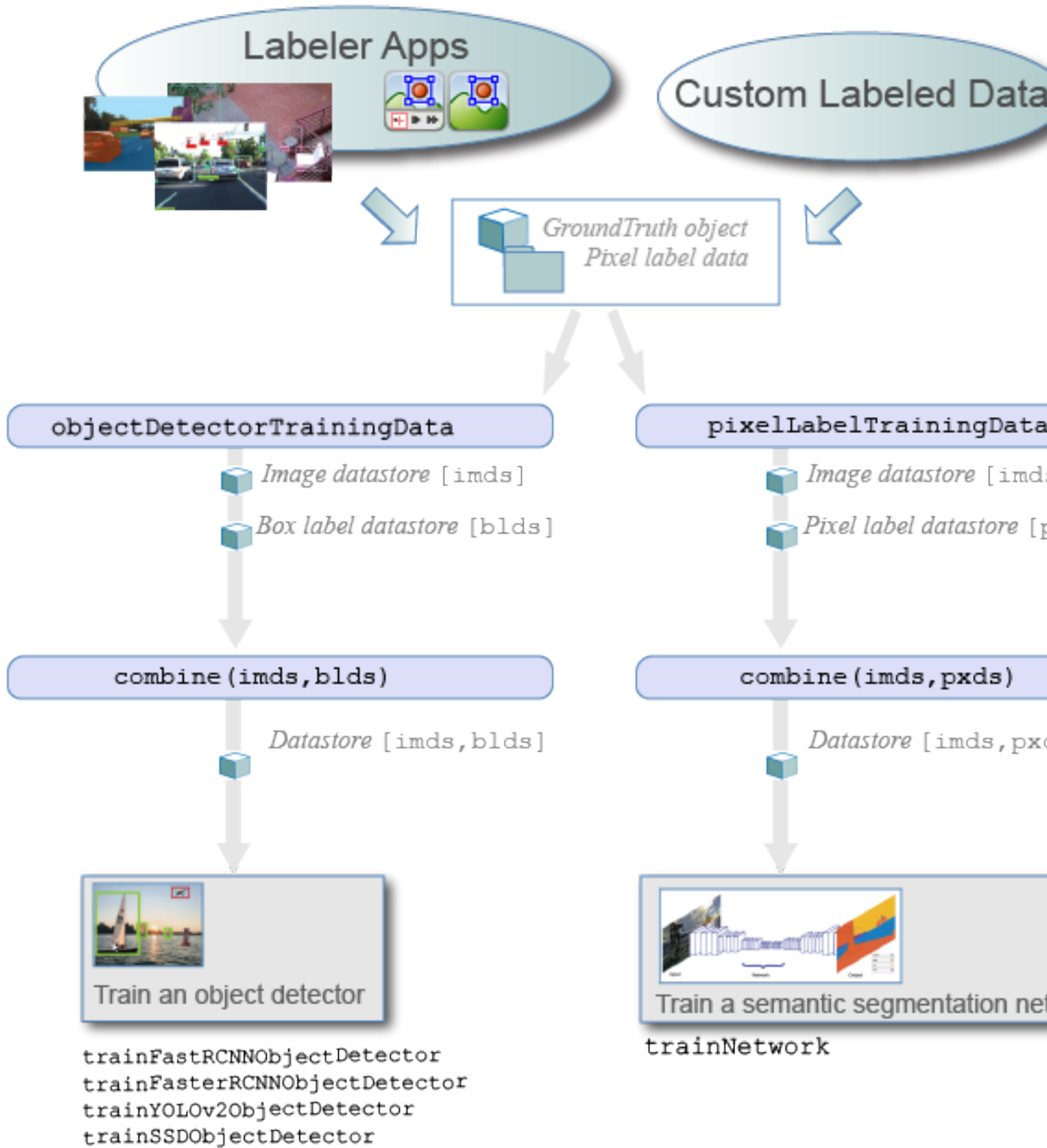
More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Training Data for Object Detection and Semantic Segmentation

You can use a labeling app and Computer Vision Toolbox objects and functions to train algorithms from ground truth data. Use the labeling app to interactively label ground truth data in a video, image sequence, image collection, or custom data source. Then, use the labeled data to create training data to train an object detector or to train a semantic segmentation network.

This workflow applies to the **Image Labeler** and **Video Labeler** apps only. To create training data for the **Ground Truth Labeler** app in Automated Driving Toolbox, use the `gatherLabelData` function.



1 Load data for labeling

- **Image Labeler** — Load an image collection from a file or `ImageDatastore` object into the app.
- **Video Labeler** — Load a video, image sequence, or a custom data source into the app.

2 Label data and select an automation algorithm: Create ROI and scene labels within the app. For more details, see:

- **Image Labeler** — “Get Started with the Image Labeler” on page 14-63
- **Video Labeler** — “Get Started with the Video Labeler” on page 14-78

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 Labeling” on page 14-49.

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. If your data source consists of multiple image collections, label the entire set of image collections to obtain an array of `groundTruth` objects. For details about sharing `groundTruth` objects, see “Share and Store Labeled Ground Truth Data” on page 14-106.**4 Create training data:** To create training data from the `groundTruth` object, use one of these functions:

- Training data for object detectors — Use the `objectDetectorTrainingData` function.
- Training data for semantic segmentation networks — Use the `pixelLabelTrainingData` function.

For objects created using a video file or custom data source, the `objectDetectorTrainingData` and `pixelLabelTrainingData` functions write images to disk for `groundTruth`. Sample the ground truth data by specifying a sampling factor. Sampling mitigates overtraining an object detector on similar samples.

5 Train algorithm:

- Object detectors — Use one of several Computer Vision Toolbox object detectors. For a list of detectors, see “Object Detection Using Features” and “Object Detection using Deep Learning”. For object detectors specific to automated driving, see the Automated Driving Toolbox object detectors listed in “Visual Perception” (Automated Driving Toolbox).
- Semantic segmentation network — For details on training a semantic segmentation network, see “Getting Started with Semantic Segmentation Using Deep Learning” on page 14-43.

See Also**Apps****Image Labeler** | **Video Labeler****Functions**

`objectDetectorTrainingData` | `pixelLabelTrainingData` | `semanticseg` |
`trainACFObjectDetector` | `trainFasterRCNNObjectDetector` | `trainRCNNObjectDetector` |
`trainRCNNObjectDetector` | `trainSSDObjectDetector` | `trainYOLOv2ObjectDetector`

Objects`groundTruth` | `groundTruthDataSource`

More About

- “Get Started with the Image Labeler” on page 14-63
- “Get Started with the Video Labeler” on page 14-78
- “Create Automation Algorithm for Labeling” on page 14-49
- “Getting Started with Object Detection Using Deep Learning” on page 14-13
- “Getting Started with Semantic Segmentation Using Deep Learning” on page 14-43
- “Getting Started with Point Clouds Using Deep Learning” on page 9-2
- “Anchor Boxes for Object Detection” on page 14-21

Create Automation Algorithm for Labeling

The **Image Labeler**, **Video Labeler**, **Lidar Labeler**, and **Ground Truth Labeler** apps enable you to label ground truth for a variety of data sources. You can use an automation algorithm to automatically label your data by creating and importing a custom automation algorithm.

Create New Algorithm

The `vision.labeler.AutomationAlgorithm` class enables you to define a custom label automation algorithm for use in the labeling apps. 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, you must first define a class for your algorithm and save it to the appropriate folder.

Create 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)
```

The resulting folder is located at `/local/MyProject/+vision/+labeler`.

Define Class That Inherits from AutomationAlgorithm Class

At the MATLAB command prompt, enter the appropriate command to open the labeling app:

- `imageLabeler`
- `videoLabeler`
- `lidarLabeler`
- `groundTruthLabeler`

Then, load a data source, create at least one label definition, and on the app toolstrip, select **Select Algorithm > Add Algorithm > Create New Algorithm**. In the `vision.labeler.AutomationAlgorithm` class template that opens, define your custom automation algorithm. Follow the instructions in the header and comments in the class.

If the algorithm is time-dependent, that is, has a dependence on the timestamp of execution, your custom automation algorithm must also inherit from the `vision.labeler.mixin.Temporal` class. For more details on implementing time-dependent, or temporal, algorithms, see “Temporal Automation Algorithms” on page 14-100.

Save Class File to Automation Folder

To use your custom algorithm from within the labeling app, save the file to the `+vision/+labeler` folder that you created. Make sure that this folder is on the MATLAB search path. To add a folder to the path, use the `addpath` function.

Refresh Algorithm List in Labeling App

To start using your custom algorithm, refresh the algorithm list so that the algorithm displays in the labeling app. On the app toolbar, select **Select Algorithm Refresh list**.

Import Existing Algorithm

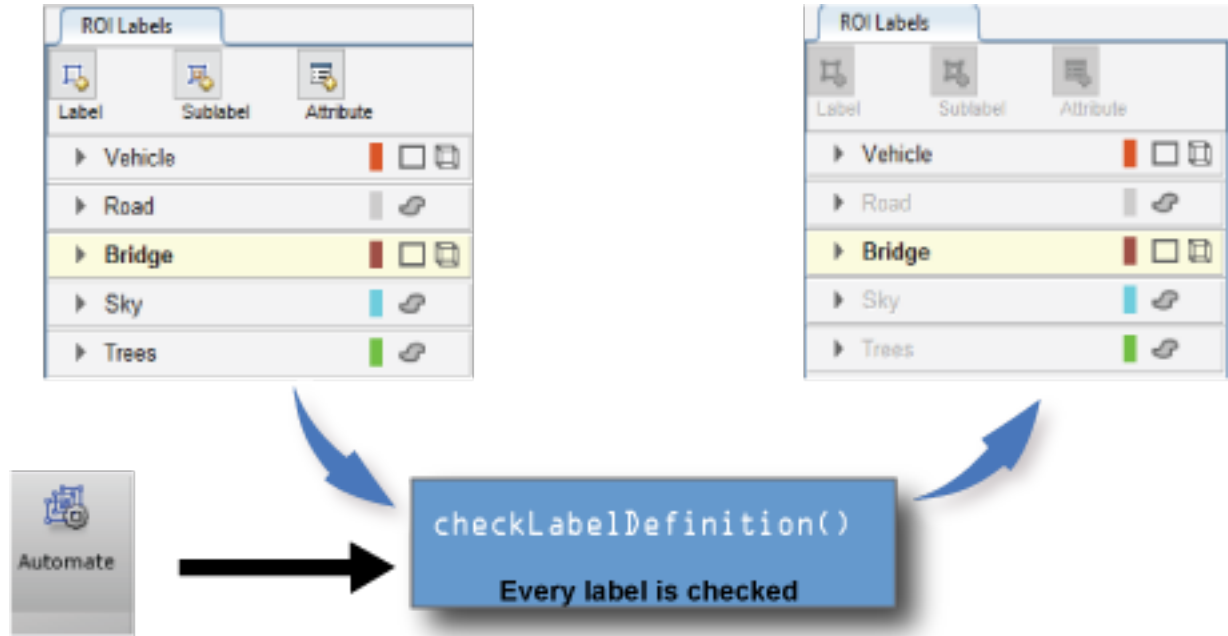
To import an existing custom algorithm into a labeling app, on the app toolbar, select **Select Algorithm > Add Algorithm > Import Algorithm** and then refresh the list.

Custom Algorithm Execution

When you run an automation session in a labeling app, the properties and methods in your automation algorithm class control the behavior of the app.

Check Label Definitions

When you click **Automate**, the app checks each label definition in the **ROI Labels** and **Scene Labels** panes 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 this 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`.



Control Settings

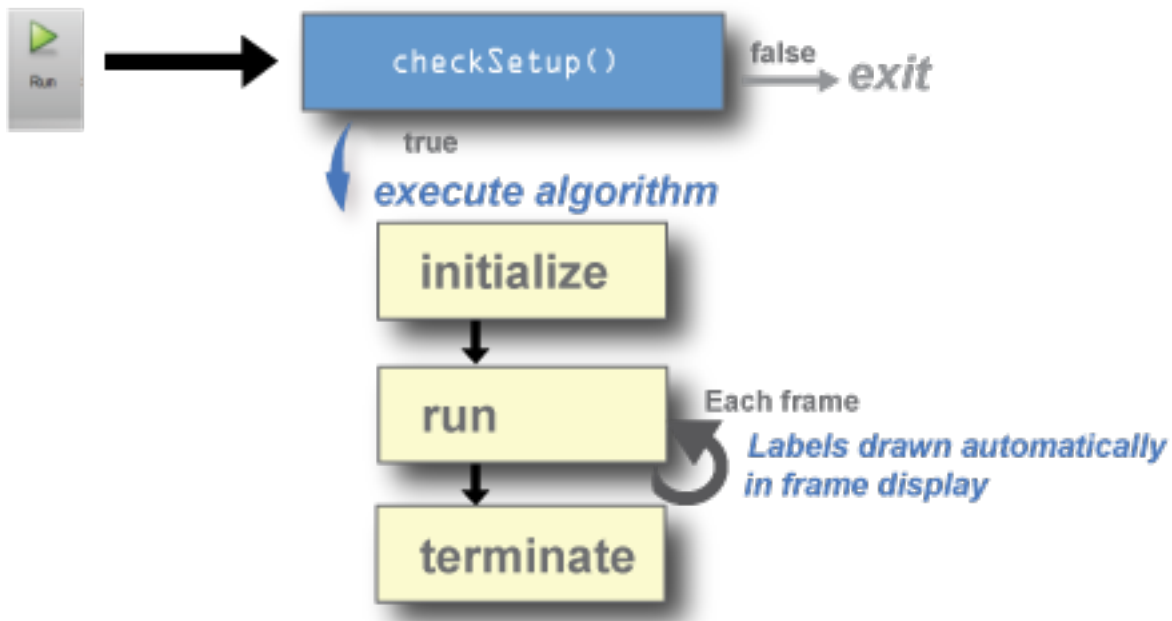
After you select the algorithm, click **Automate** to start an automation session. Then, click **Settings**, which enables you to modify custom app settings. To control the **Settings** options, use the `settingsDialog` method.



Control Algorithm Execution

When you open an automation algorithm session in the app and then click **Run**, the app calls the `checkSetup` method to check if it is ready for execution. If the method returns `false`, the app does not execute the automation algorithm. If the method returns `true`, the app calls the `initialize` method and then the `run` method on every frame selected for automation. Then, at the end of the automation run, the app calls the `terminate` method.

The diagram shows this flow of execution for the labeling apps.



- 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.
- Use the `initialize` method to initialize the state for your custom algorithm by using the frame.
- Use the `run` method to implement the core of the algorithm that computes and returns labels for each frame.
- Use the `terminate` method to clean up or terminate the state of the automation algorithm after the algorithm runs.

See Also

Apps

Ground Truth Labeler | Image Labeler | Lidar Labeler | Video Labeler

Functions

`vision.labeler.AutomationAlgorithm` | `vision.labeler.mixin.Temporal`

Related Examples

- “Automate Ground Truth Labeling of Lane Boundaries” (Automated Driving Toolbox)
- “Automate Ground Truth Labeling for Semantic Segmentation” (Automated Driving Toolbox)
- “Automate Attributes of Labeled Objects” (Automated Driving Toolbox)

More About

- “Get Started with the Image Labeler” on page 14-63
- “Get Started with the Video Labeler” on page 14-78
- “Get Started with the Lidar Labeler” (Lidar Toolbox)
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Temporal Automation Algorithms” on page 14-100

Label Pixels for Semantic Segmentation

The **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** (requires Automated Driving Toolbox) apps enable 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, video, or image sequence into a labeling app and defining pixel ROI labels. For more details, see:

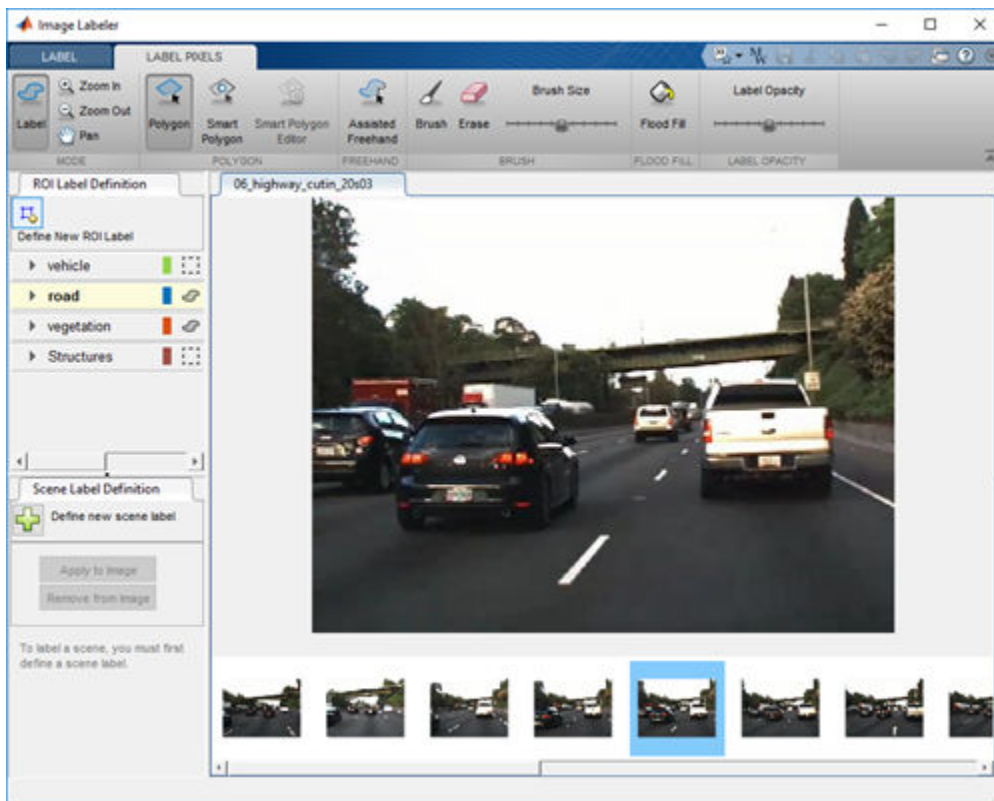
- **Image Labeler** — “Get Started with the Image Labeler” on page 14-63
- **Video Labeler** — “Get Started with the Video Labeler” on page 14-78
- **Ground Truth Labeler** — “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)

This example shows pixel labeling with the **Image Labeler**. You use the same tools to label videos and image sequences with the **Video Labeler** or **Ground Truth Labeler**.

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**, **Assisted Freehand** and **Brush**.
- First label distant objects with a rough estimation of object borders. Then, label nearer objects with more precise object borders.

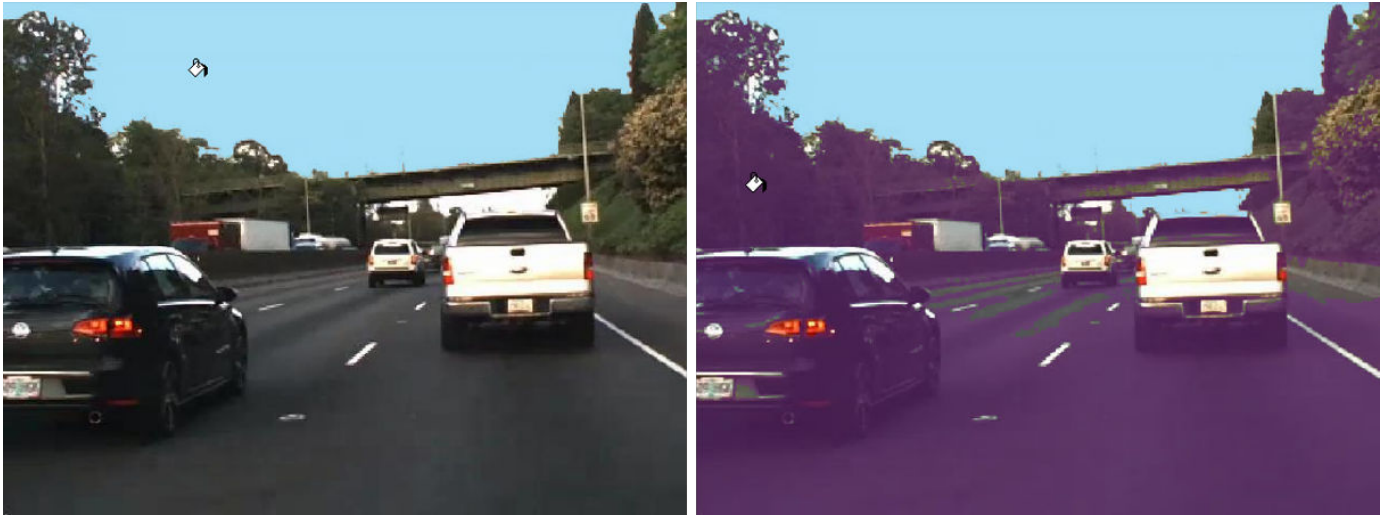


Label Pixels Using Flood Fill Tool

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

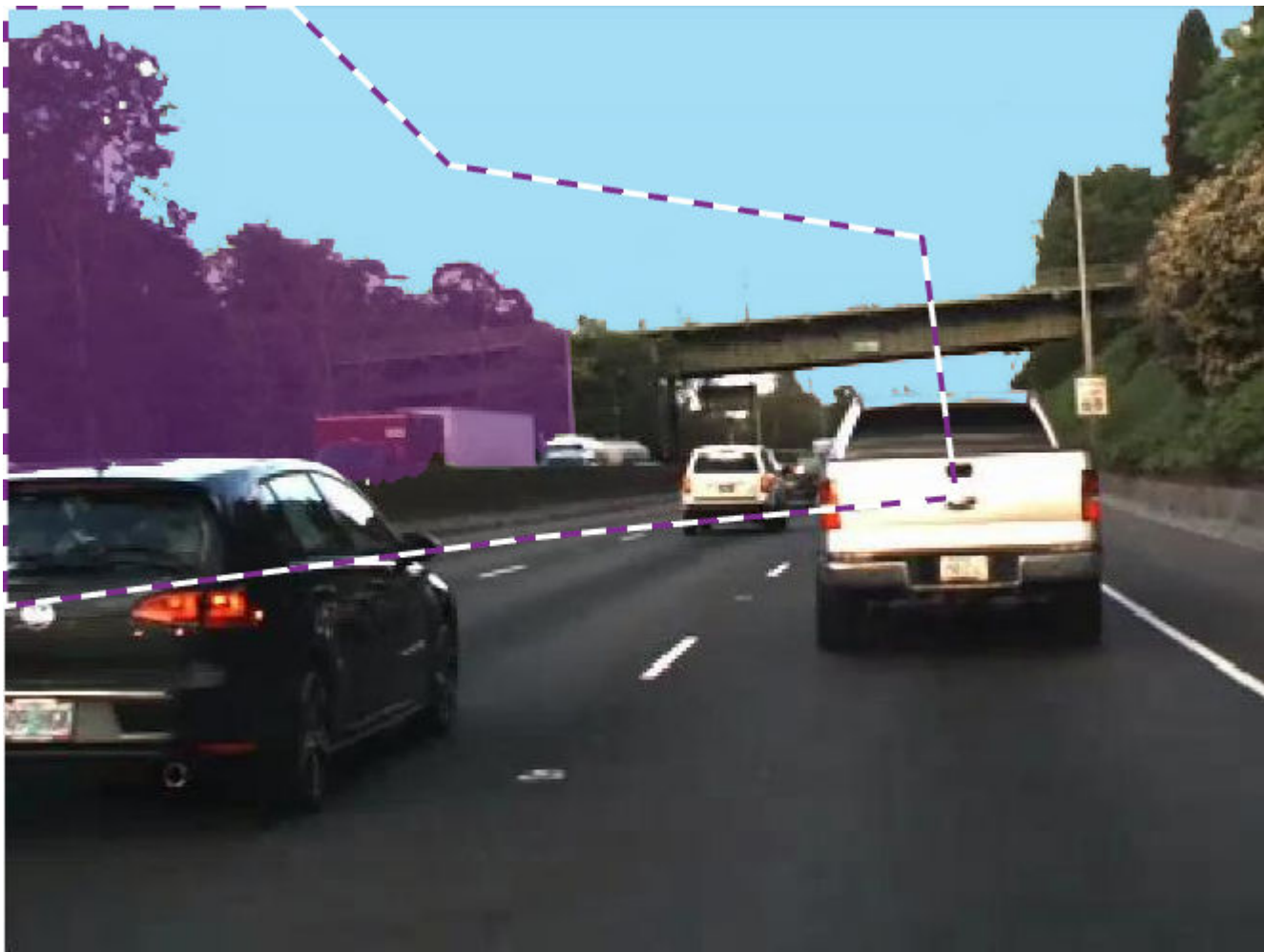
The **Smart Polygon** tool estimates the shape of an object of interest within a polygon that you draw. The tool is useful when the shape of the object is not a simple polygon. This example uses **Smart Polygon** to label the vegetation, which has a complicated boundary with the sky.

To label pixels using **Smart Polygon**:

- 1 Select the tool and a label. The pointer changes to a crosshair \oplus .
- 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.

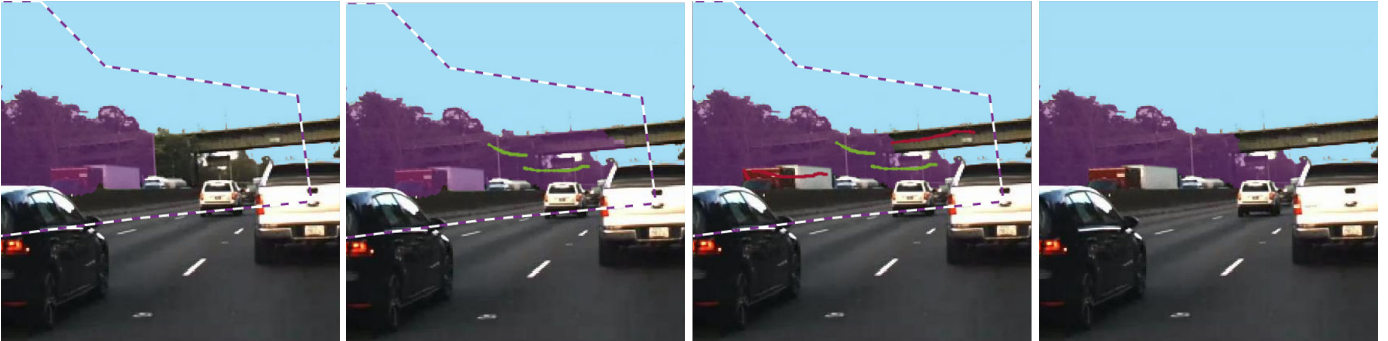


Smart Polygon Actions

Goal	Control
Move vertex	Click and drag the vertex.
Add vertex	<ul style="list-style-type: none"> 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 .

- 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.

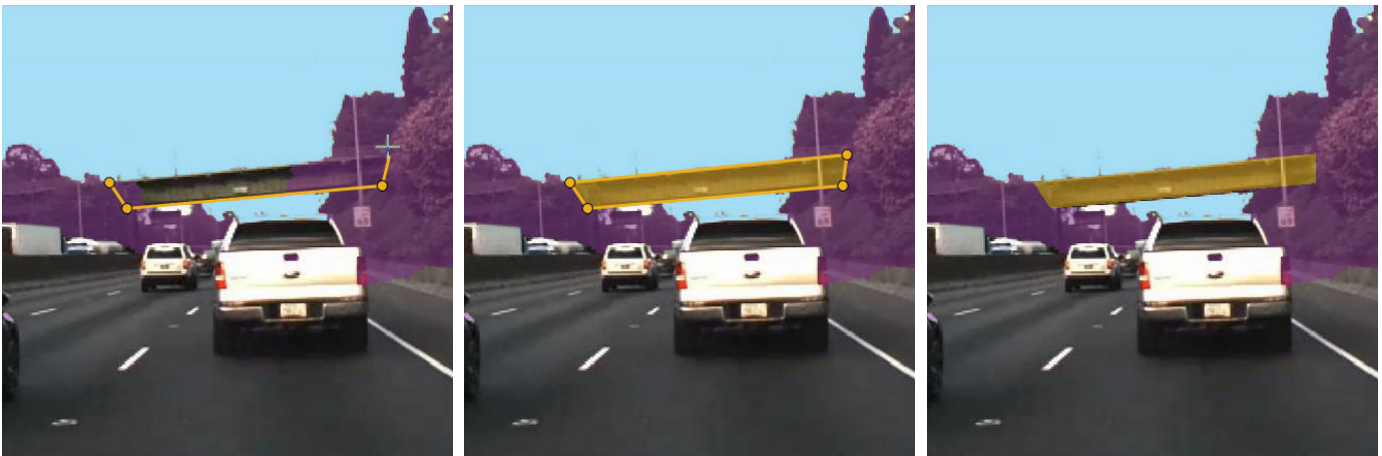
- Select **Erase Marks** to remove foreground or background marks that are no longer needed.
- See Tips on page 14-61 for additional suggestions on using the **Smart Polygon** tool.



- 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.



Label Pixels Using Assisted Freehand Tool

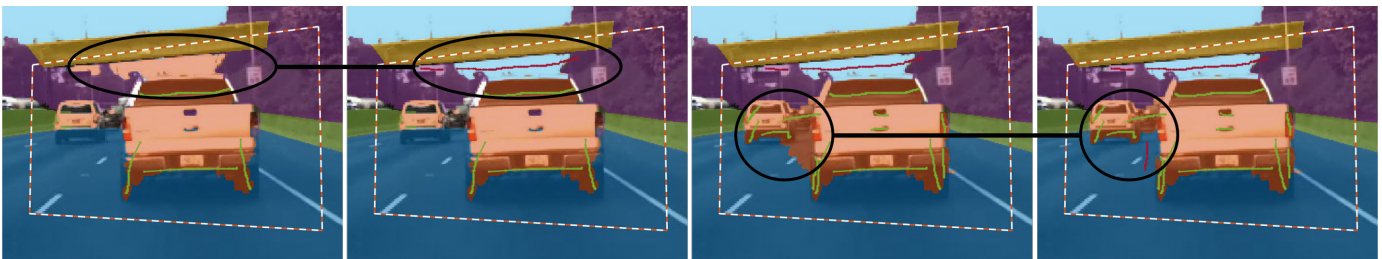
The **Assisted Freehand** tool enables you to draw an ROI that automatically follows the edge of the subject in the underlying image. You can also adjust the size and position of the ROI by using your mouse.



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.



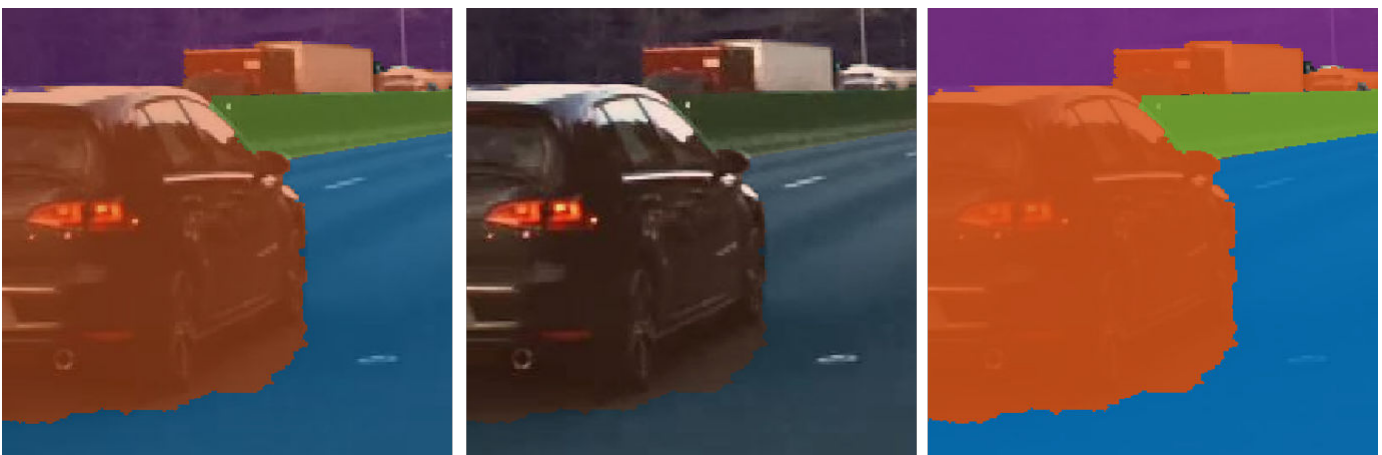
The **Erase** tool removes pixel labels when you draw over the image with the mouse.

Visualize Pixel Labels

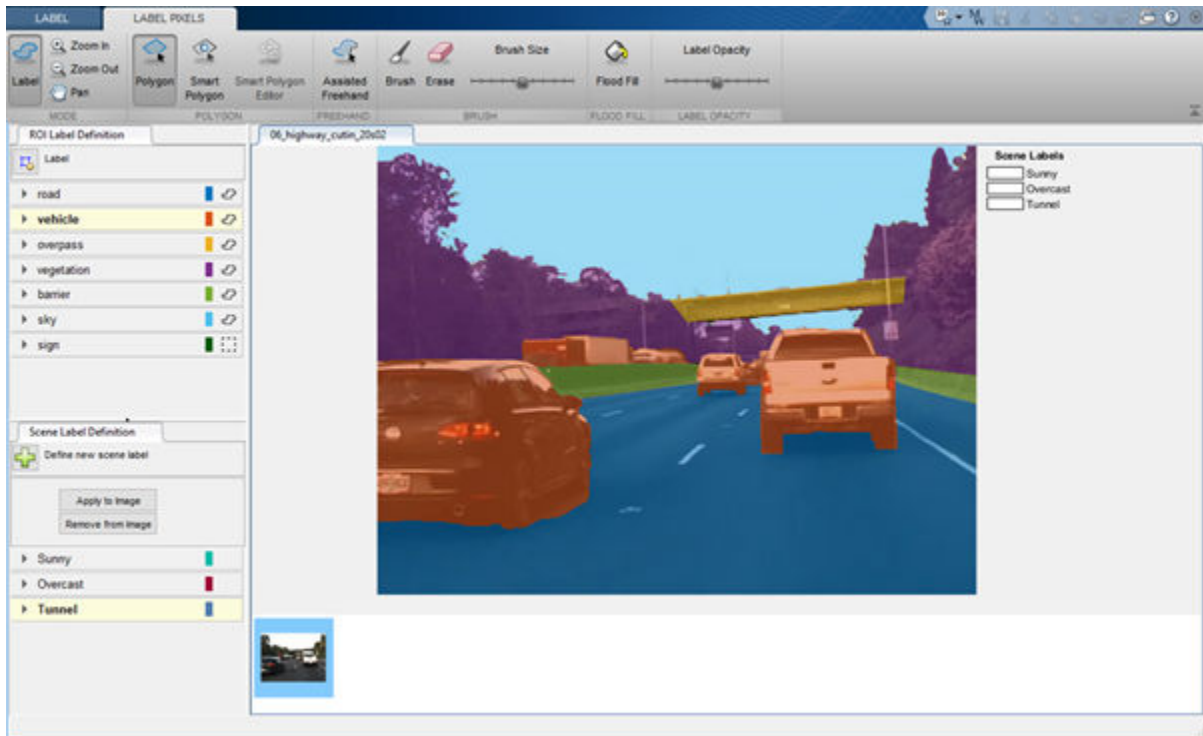
You can modify the view of the image to facilitate pixel labeling. The **Zoom In**, **Zoom Out**, and **Pan** options enable you to zoom and pan the image with the mouse. To resume pixel labeling, click the **Label** icon.

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**. To delete all pixels in a frame, press **Ctrl+Shift+Delete**.
- To cut or copy all pixels in a frame, press **Ctrl+Shift+X** or **Ctrl+Shift+C**. To paste the cut or copied pixels, press **Ctrl+Shift+V**.
- 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.
- To fill all or all remaining pixels, select an ROI label from your list and press **Shift+Click** (you can use left- or right-click).

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

Ground Truth Labeler | **Image Labeler** | **Video Labeler**

More About

- "Get Started with the Image Labeler" on page 14-63
- "Get Started with the Video Labeler" on page 14-78
- "Get Started with the Ground Truth Labeler" (Automated Driving Toolbox)
- "How Labeler Apps Store Exported Pixel Labels" on page 14-16

Get Started with the Image Labeler

The **Image Labeler** app provides an easy way to mark rectangular region of interest (ROI) labels, polyline ROI labels, pixel ROI labels, and scene labels in a video or image sequence. This example gets you started using the app by showing you how to:

- Manually label an image frame from an image collection.
- Automatically label across image frames using an automation algorithm.
- Export the labeled ground truth data.

ROI and Scene Label Definitions

- An ROI label corresponds to either a rectangular, polyline, 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.

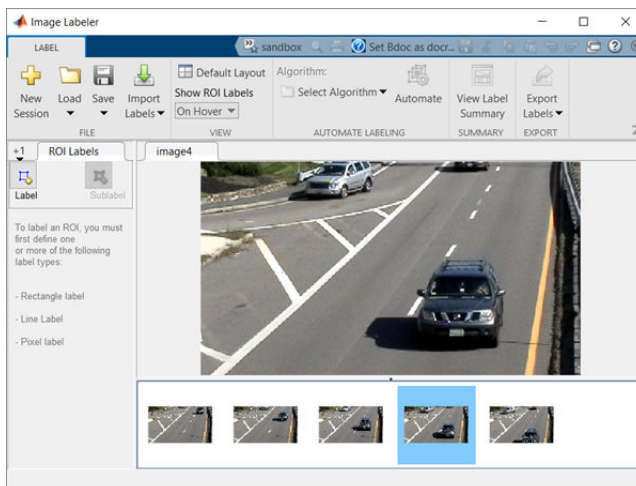
Load Unlabeled Data

Open the app and load a collection of images. You can load images stored in a datastore, from a folder, or load a previous labeler session. The images must be readable by `imread`.

```
imageFolder = fullfile(toolboxdir('vision'),'visiondata','stopSignImages')
imds = imageDatastore(imageFolder)
imageLabeler(imds)
```

```
imageFolder = fullfile(toolboxdir('vision'),'visiondata','stopSignImages')
imageLabeler(imageFolder)
```

Alternatively, open the app from the **Apps** tab, under **Image Processing and Computer Vision**. Then, from the **Load** menu, load an images data source.


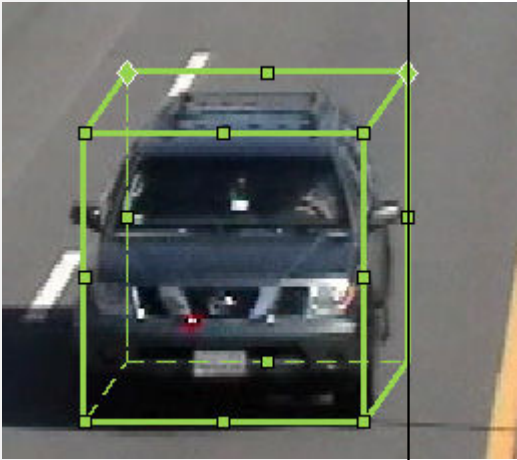




Create Label Definitions

Define the labels you intend to draw. In this example, you define labels directly within the app. To define labels from the MATLAB command line instead, use the `labelDefinitionCreator`.

Create ROI Labels

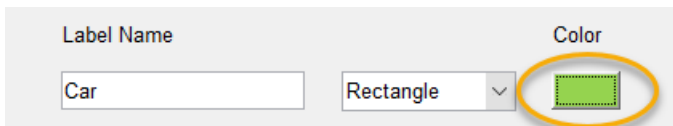
An ROI label is a label that corresponds to a region of interest (ROI). You can define these types of ROI labels.

ROI Label	Description	Example: Driving Scene
<p>Rectangle</p>	<p>Draw rectangular ROI labels (bounding boxes) around objects.</p>	<p>Vehicles, pedestrians, road signs</p> 
<p>Projected cuboid</p>	<p>Draw cuboidal ROI labels (3-D bounding boxes).</p>	
<p>Line</p>	<p>Draw linear ROI labels to represent lines. To draw a polyline ROI, use two or more points.</p>	<p>Lane boundaries, guard rails, road curbs</p> 

ROI Label	Description	Example: Driving Scene
Pixel label	Assign labels to pixels for semantic segmentation. You can label pixels manually using polygons, brushes, or flood fill. For more on pixel labeling, see “Label Pixels for Semantic Segmentation” on page 14-53.	Vehicles, road surface, trees, pavement 

In this example, you define a vehicle group for labeling types of vehicles, and then create a Rectangle ROI label for a Car and a Truck. Optionally, you can use the **Show ROI Labels** drop-down menu to select **On Hover**, **Always**, or **Never** to control how the ROI label names appear during labeling. By default, the names will appear when you hover on an ROI.

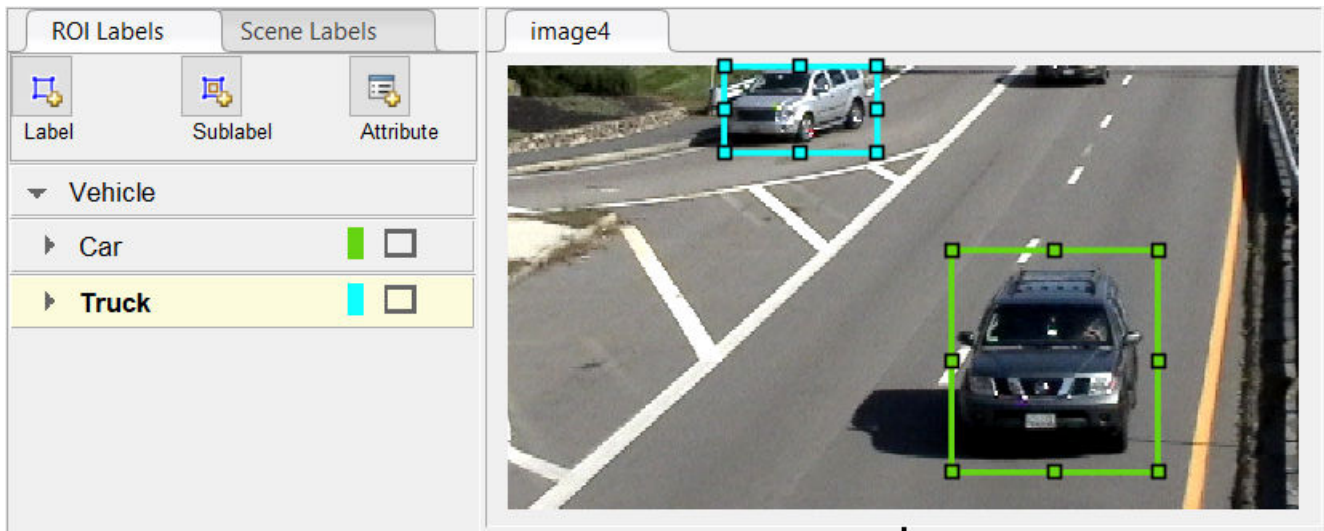
- 1 In the **ROI Labels** pane on the left, click **Label**.
- 2 Create a Rectangle label named Car.
- 3 Optionally, change the label color by clicking the preview color.



- 4 From the Group drop-down menu, select **New Group** and name the group **Vehicle**
- 5 Click **OK**.

The **Vehicle** group name appears in the **ROI Labels** pane with the label **Car** created. You can move a label in the list to a different position or group in the list by left-clicking and dragging the label up or down.

- 6 Add a second label. Click **Label**. Name the label **Truck** and make sure the **Vehicle** group is selected. Click **OK**.
- 7 Use the mouse to draw rectangular **Car** ROIs around the two vehicles.



Create Sublabels

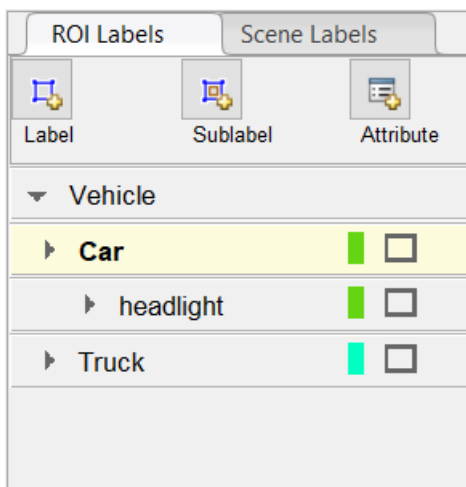
A sublabel is a type of ROI label that corresponds to a parent ROI label. Each sublabel must belong to, or be a child of, a specific label defined in the **ROI Labels** pane. For example, in a driving scene, a vehicle label might have sublabels for headlights, license plates, or wheels.

Define a sublabel for headlights.

- 1 In the **ROI Labels** pane on the left, click the **Car** label.
- 2 Click **Sublabel**.
- 3 Create a Rectangle sublabel named **headlight** and optionally write a description. Click **OK**.

The **headlight** sublabel appears in the **ROI Labels** pane. The sublabel is nested under the selected ROI label, **Car**, and has the same color as its parent label.

You can add multiple sublabels under a label. You can also drag-and-drop the sublabels to reorder them in the list. Right-click any label for additional edits.

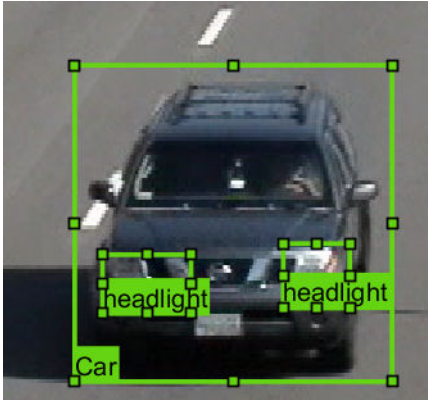


- 4 In the **ROI Labels** pane, select the **headlight** sublabel.

- In the image frame, select the **Car** label. The label turns yellow when selected. You must select the **Car** label (parent ROI) before you can add a sublabel to it.

Draw **headlight** sublabels for each of the cars.

- Repeat the previous steps to label the headlights of the other car. To draw the labels more precisely, use the pan and zoom options located in the upper-right corner of the labeling window.



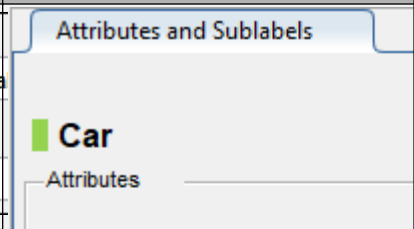
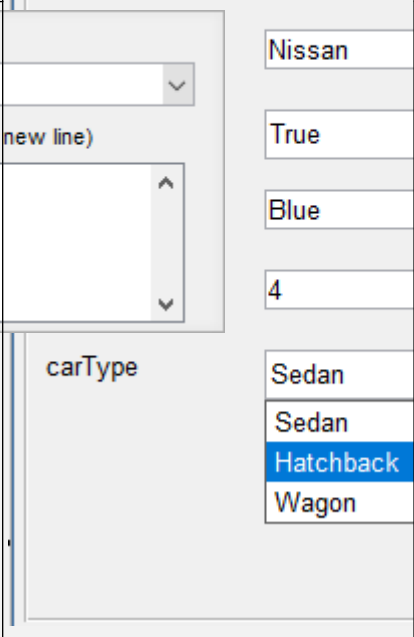
Sublabels can only be used with rectangular or polyline ROI labels and cannot have their own sublabels. For more details on working with sublabels, see “Use Sublabels and Attributes to Label Ground Truth Data” on page 14-96.

Create Attributes

An attribute provides further categorization of an ROI label or sublabel. Attributes specify additional information about a drawable label. For example, in a driving scene, attributes might include the type or color of a vehicle.

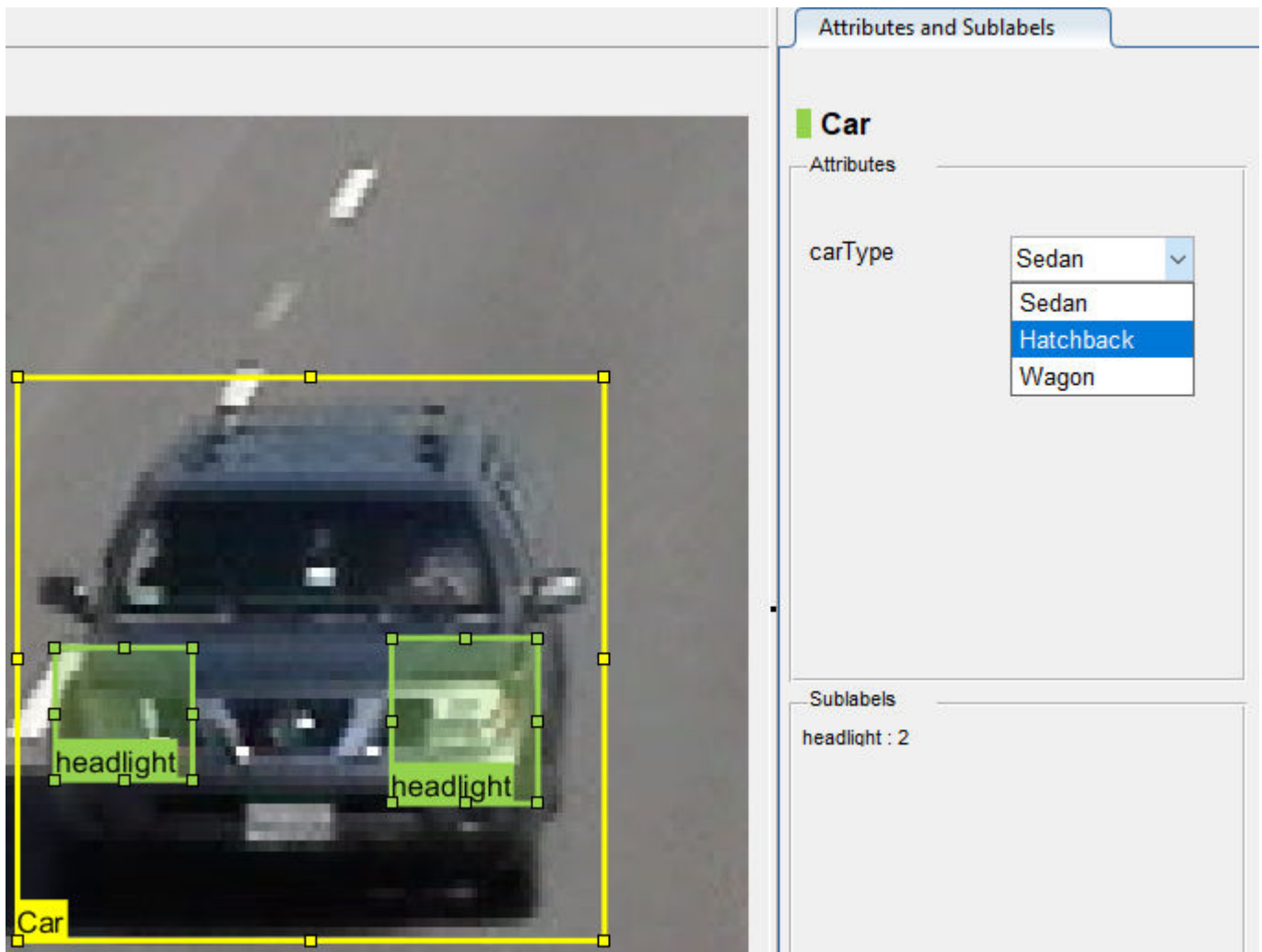
You can define these types of attributes.

Attribute Type	Sample Attribute Definition	Sample Default Values
Numeric Value	Attribute Name <input type="text" value="numDoors"/> Default Scalar Value (Optional) <input type="text" value="4"/>	
String	Attribute Name <input type="text" value="color"/> Default Value (Optional) <input type="text"/>	

Attribute Type	Sample Attribute Definition	Sample Default Values
Logical	Attribute Name <input type="text" value="inMotion"/> Logical Default Value (Optional) <input type="text" value="True"/>	
List	Attribute Name <input type="text" value="carType"/> List List Items (Each item must appear on a new line) <input type="text" value="Sedan"/> <input type="text" value="Hatchback"/> <input type="text" value="Wagon"/>	

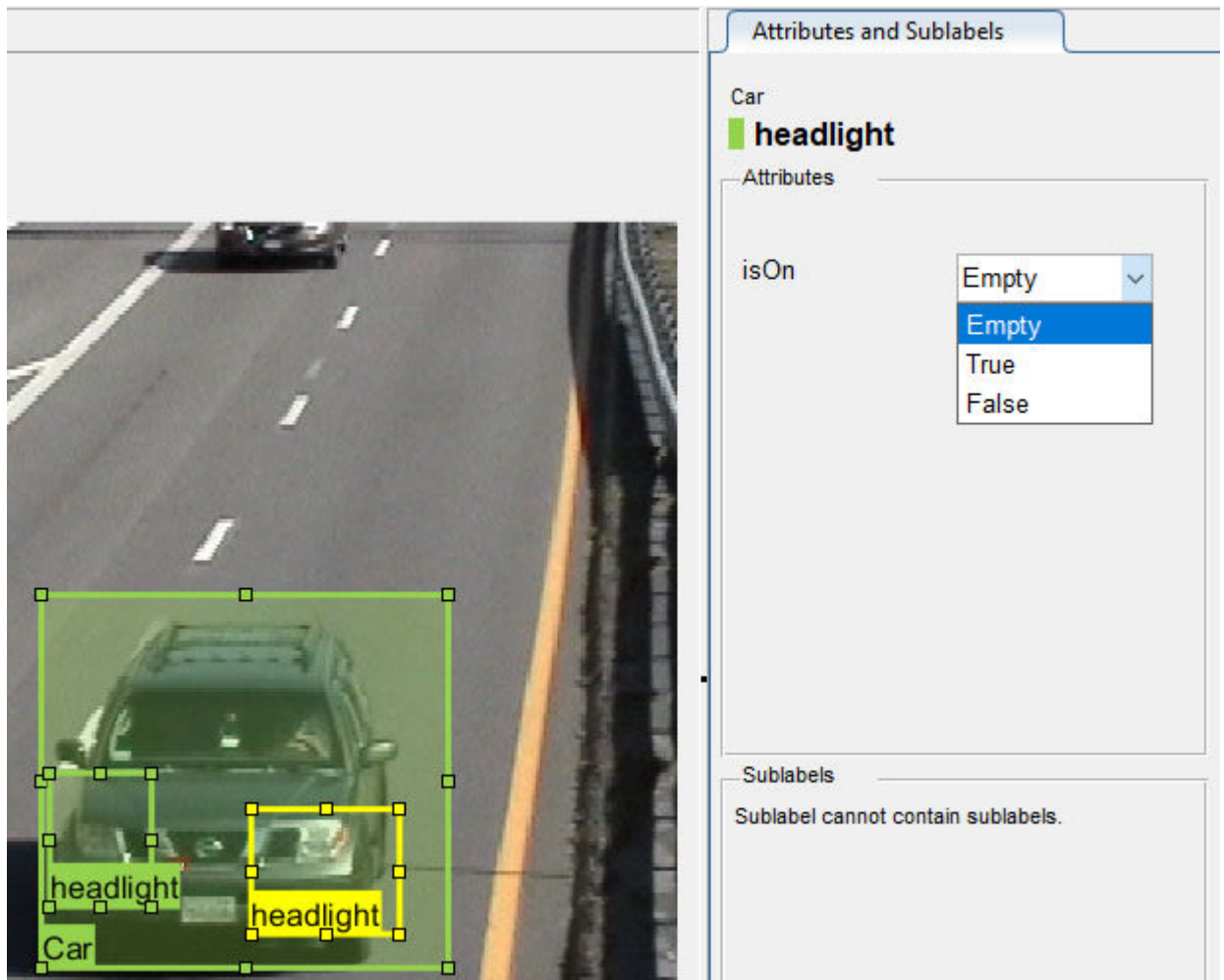
Add an attribute for the vehicle type.

- 1 In the **ROI Labels** pane on the left, select the **Car** label and click **Attribute**.
- 2 In the **Attribute Name** box, type `carType`. Set the attribute type to **List**.
- 3 In the **List Items** section, type different types of cars, such as Sedan, Hatchback, and Wagon, each on its own line. Optionally give the attribute a description, and click **OK**.
- 4 In the first frame of the video, select a **Car** ROI label. In the **Attributes and Sublabels** pane, select the appropriate **carType** attribute value for that vehicle.
- 5 Repeat the previous step to assign a **carType** attribute to the other vehicle.



You can also add attributes to sublabels. Add an attribute for the **headlight** sublabel that tells whether the headlight is on.

- 1 In the **ROI Labels** pane on the left, select the **headlight** sublabel and click **Attribute**.
- 2 In the **Attribute Name** box, type `isOn`. Set the attribute type to **Logical**. Leave the **Default Value** set to **Empty**, optionally write a description, and click **OK**.
- 3 Select a headlight in the video frame. Set the appropriate **isOn** attribute value, or leave the attribute value set to **Empty**.
- 4 Repeat the previous step to set the **isOn** attribute for the other headlights.



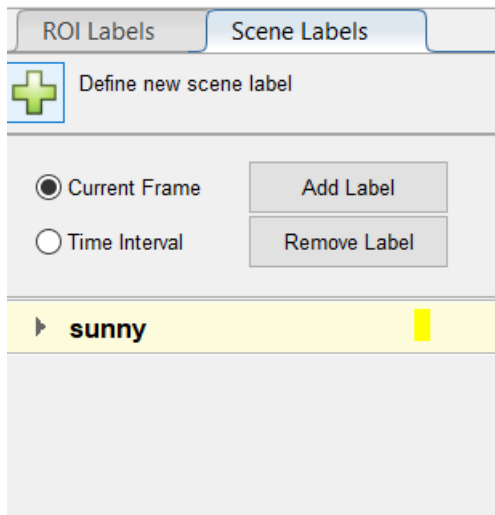
To delete an attribute, right-click an ROI label or sublabel, and select the attribute to delete. Deleting the attribute removes attribute information from all previously created ROI label annotations.

Create Scene Labels

A scene label defines additional information for the entire scene. Use scene labels to describe conditions, such as lighting and weather, or events, such as lane changes.

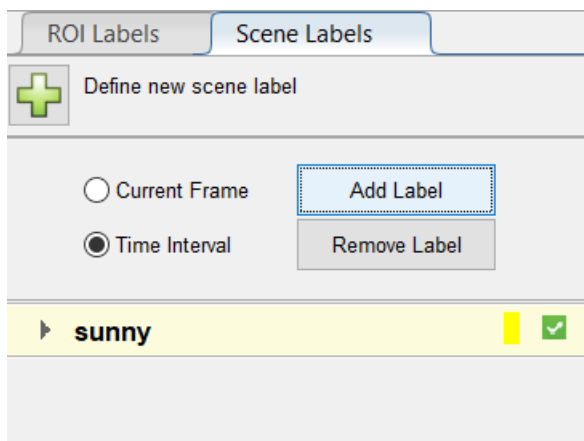
Create a scene label to use in the video.

- 1 Select the **Scene Labels** tab on the left.
- 2 Click the **Define new scene label** button, and create a scene label named sunny. Make sure **Group** is set to None. Click **OK**.



The **Scene Labels** pane shows the scene label definition.

- 3 You can apply the label to just the current frame or to an interval of frames. With the sunny scene label definition still selected in the **Scene Labels** pane, select **Time Interval**.
- 4 Click the **Add Label**. A checkmark appears for the sunny scene label indicating that the label now applies to all frames in the time interval.



- 5 To edit or delete a scene label, right-click on the label and select either **Edit Label** or **Delete Label**.

Label Ground Truth

So far, you have labeled only one frame in the video. To label the remaining frames, choose one of these options.

Label Ground Truth Manually

When you click the right arrow key to advance to the next frame, the ROI labels from the previous frame do not carry over. Only the **sunny** scene label applies to each frame, because this label was applied over the entire time interval.

Advance frame by frame and draw the label and sublabel ROIs manually. Also update the attribute information for these ROIs.

Label Ground Truth Using Automation Algorithm

To speed up the labeling process, you can use an automation algorithm within the app. You can either define your own automation algorithm, see “Create Automation Algorithm for Labeling” on page 14-49 or use a built-in automation algorithm. In this example, you label the ground truth using a built-in algorithm.

After using an automation algorithm you can manually label the remaining frames with sublabel and attribute information.

To further evaluate your labels, you can view a visual summary of the labeled ground truth. From the app toolbar, select **View Label Summary**. Use this summary to compare the frames, frequency of labels, and scene conditions. For more details, see “View Summary of Ground Truth Labels” on page 14-102. This summary does not support sublabels or attributes.

Export Labeled Ground Truth

You can export the labeled ground truth to a MAT-file or to a variable in the MATLAB workspace. In both cases, the labeled ground truth is stored as a `groundTruth` object. You can use this object to train a deep-learning-based computer vision algorithm. For more details, see “Training Data for Object Detection and Semantic Segmentation” on page 14-45.

Note If you export pixel data, the pixel label data and ground truth data are saved in separate files but in the same folder. For considerations when working with exported pixel labels, see “How Labeler Apps Store Exported Pixel Labels” on page 14-16.

In this example, you export the labeled ground truth to the MATLAB workspace. From the app toolbar, select **Export Labels > To Workspace**. The exported MATLAB variable is `gTruth`.

Display the properties of the exported `groundTruth` object. The information in your exported object might differ from the information shown here.

```
gTruth
gTruth =
    groundTruth with properties:
        DataSource: [1x1 groundTruthDataSource]
        LabelDefinitions: [2x6 table]
        LabelData: [531x3 timetable]
```

Data Source

`DataSource` is a `groundTruthDataSource` object containing the path to the images or video and timestamps. Display the properties of this object.

```
gTruth.DataSource
ans =
```

groundTruthDataSource for a video file with properties

```
Source: ...matlab\toolbox\vision\visiondata\visiontraffic.avi
TimeStamps: [531x1 duration]
```

Label Definitions

`LabelDefinitions` is a table containing information about the label definitions. This table does not contain information about the labels that are drawn on the video frames. To save the label definitions in their own MAT-file, from the app toolbar, select **Save > Label Definitions**. You can then import these label definitions into another app session by selecting **Import Files**.

Display the label definitions table. Each row contains information about an ROI label definition or a scene label definition. If you exported pixel label data, the `LabelDefinitions` table also includes a `PixelLabelID` column containing the ID numbers for each pixel label definition.

```
gTruth.LabelDefinitions
```

```
ans =
  3x6 table
```

Name	Type	LabelColor	Group	Description	Hierarchy
{'Car' }	Rectangle	{1x3 double}	{'Vehicle'}	{0x0 char}	{1x1 struct}
{'Truck'}	Rectangle	{1x3 double}	{'Vehicle'}	{0x0 char}	{0x0 double}
{'Sunny'}	Scene	{1x3 double}	{'Weather'}	{0x0 char}	{0x0 double}

Within `LabelDefinitions`, the `Hierarchy` column stores information about the sublabel and attribute definitions of a parent ROI label.

Display the sublabel and attribute information for the Car label.

```
gTruth.LabelDefinitions.Hierarchy{1}
```

```
ans =

  struct with fields:
    numDoors: [1x1 struct]
    color: [1x1 struct]
    inMotion: [1x1 struct]
    carType: [1x1 struct]
    headlight: [1x1 struct]
    Type: Rectangle
    Description: ''
```

Display information about the `headlight` sublabel.

```
gTruth.LabelDefinitions.Hierarchy{1}.headlight
```

```
ans =

  struct with fields:
    Type: Rectangle
    Description: ''
    Color: [0.5862 0.8276 0.3103]
    isOn: [1x1 struct]
```

Display information about the `carType` attribute.

```
gTruth.LabelDefinitions.Hierarchy{1}.carType
ans =
  struct with fields:
    ListItems: {3x1 cell}
    Description: ''
```

Save App Session

From the app toolstrip, select **Save** and save a MAT-file of the app session. The saved session includes the data source, label definitions, and labeled ground truth. It also includes your session preferences, such as the layout of the app. To change layout options, select **Layout**.

At any time during a session, you can select **New Session** to start a new session. You have the option of saving the current session or cancelling.

The app session MAT-file is separate from the ground truth MAT-file that is exported when you select **Export > From File**. To share labeled ground truth data, as a best practice, share the ground truth MAT-file containing the `groundTruth` object, not the app session MAT-file. For more details, see “Share and Store Labeled Ground Truth Data” on page 14-106.

See Also

Apps
Image Labeler

Objects

`groundTruth` | `groundTruthDataSource` | `imageDatastore` | `labelDefinitionCreator` | `vision.labeler.AutomationAlgorithm`

More About

- “Training Data for Object Detection and Semantic Segmentation” on page 14-45
- “Keyboard Shortcuts and Mouse Actions for Image Labeler” on page 14-112
- “Use Sublabels and Attributes to Label Ground Truth Data” on page 14-96
- “Label Pixels for Semantic Segmentation” on page 14-53
- “Create Automation Algorithm for Labeling” on page 14-49

Choose an App to Label Ground Truth Data

You can use Computer Vision Toolbox, Automated Driving Toolbox, Lidar Toolbox™, and Signal Processing Toolbox™ apps to label ground truth data. Use this labeled data to validate or train algorithms such as image classifiers, object detectors, semantic segmentation networks, and deep learning applications. The choice of labeling app depends on several factors, including the supported data sources, labels, and types of automation.

One key consideration is the type of data that you want to label.

- If your data is an image collection, use the **Image Labeler** app. An image collection is an unordered set of images that can vary in size. For example, you can use the app to label images of books for training a classifier.
- If your data is a single video or image sequence, use the **Video Labeler** app. An image sequence is an ordered set of images that resembles a video. For example, you can use this app to label a video or image sequence of cars driving on a highway for training an object detector.
- If your data includes multiple time-overlapped signals, such as videos, image sequences, or lidar signals, use the **Ground Truth Labeler** app. For example, you can label data for a single scene captured by multiple sensors mounted on a vehicle.
- If your data is only a lidar signal, use the **Lidar Labeler**. For example, you can use this app to label data captured from a point cloud sensor.
- If your data consists of single-channel or multichannel one-dimensional signals, use the **Signal Labeler**. For example, you can label biomedical, speech, communications, or vibration data. To perform audio-specific tasks, such as speech detection, speech-to-text transcription, and recording new audio, use the **Audio Labeler** app.

This table summarizes the key features of the labeling apps.

Labeling App	Data Sources	Label Support	Automation	Additional Features
Image Labeler	<ul style="list-style-type: none"> • Image collections 	<ul style="list-style-type: none"> • Rectangle regions of interest (ROIs) • Projected cuboid (ROIs) • Line ROIs • Pixel ROIs • Sublabels • Attributes • Scenes 	<ul style="list-style-type: none"> • Built-in automation algorithms • Custom automation algorithms 	<ul style="list-style-type: none"> • View visual summary of labeled data

Labeling App	Data Sources	Label Support	Automation	Additional Features
Video Labeler	<ul style="list-style-type: none"> Videos Image sequences Custom image data sources 	<ul style="list-style-type: none"> Rectangle ROIs Projected cuboid (ROIs) Line ROIs Pixel ROIs Sublabels Attributes Scenes 	<ul style="list-style-type: none"> Built-in automation algorithms Custom automation algorithms Temporal automation algorithms 	<ul style="list-style-type: none"> View visual summary of labeled data
Ground Truth Labeler	<ul style="list-style-type: none"> Videos Image sequences Custom image data sources Point cloud sequences (PCD or PLY files) Velodyne® lidar files Rosbags (requires ROS Toolbox) 	<ul style="list-style-type: none"> Rectangle ROIs Projected cuboid (ROIs) Cuboid ROIs Line ROIs Pixel ROIs Sublabels Attributes Scenes 	<ul style="list-style-type: none"> Built-in automation algorithms, including vehicle and lane detection algorithms and a point cloud temporal interpolation algorithm Custom automation algorithms Temporal automation algorithms 	<ul style="list-style-type: none"> View visual summary of labeled data Connect external tool to app for displaying time-synchronized signals, such as lidar or CAN bus data Customize loading interface to support additional data sources
Lidar Labeler	<ul style="list-style-type: none"> Point cloud sequences (PCD or PLY files) Velodyne lidar files LAS/LAZ file sequences Rosbags (requires ROS Toolbox) 	<ul style="list-style-type: none"> Cuboid ROIs Attributes Scenes 	<ul style="list-style-type: none"> Built-in automation algorithms, including a lidar object tracker and point cloud temporal interpolator Custom automation algorithms Temporal automation algorithms 	<ul style="list-style-type: none"> View the cuboid labels in top, side, and front views Save and reuse custom camera views Connect to external tool to display time-synchronized signals for ease of labeling, such as videos, to use as a reference while labeling

Labeling App	Data Sources	Label Support	Automation	Additional Features
Signal Labeler	<ul style="list-style-type: none"> Numeric arrays, MATLAB timetables, and labeledSignalSet objects in the MATLAB workspace MAT-files and CSV files 	<ul style="list-style-type: none"> Time-based ROIs Time-based points Attributes Sublabels 	<ul style="list-style-type: none"> Built-in peak labeling Custom automation algorithms 	<ul style="list-style-type: none"> Expand, collapse, and browse details of labeled data View signal spectra and spectrograms Label ROIs and points using the spectrogram Label signals in bulk Use Label Viewer to view and compare labels
Audio Labeler	<ul style="list-style-type: none"> Audio files (WAVE, OGG, FLAC, AU, AIFF, AIFC, MP3, MPEG-4 AAC) labeledSignalSet objects in the MATLAB workspace or in MAT-files 	<ul style="list-style-type: none"> Time-based ROIs File-level labels 	<ul style="list-style-type: none"> Speech detection Speech-to-text transcription (requires Audio Toolbox™ extended functionality for speech2text) 	<ul style="list-style-type: none"> Audio playback Audio recording Inspect audio file information

See Also

More About

- “Get Started with the Image Labeler” on page 14-63
- “Get Started with the Video Labeler” on page 14-78
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Get Started with the Lidar Labeler” (Lidar Toolbox)
- “Using Signal Labeler App” (Signal Processing Toolbox)
- “Label Audio Using Audio Labeler” (Audio Toolbox)

Get Started with the Video Labeler

The **Video Labeler** app provides an easy way to mark rectangular region of interest (ROI) labels, polyline ROI labels, pixel ROI labels, and scene labels in a video or image sequence. This example gets you started using the app by showing you how to:

- Manually label an image frame from a video.
- Automatically label across image frames using an automation algorithm.
- Export the labeled ground truth data.

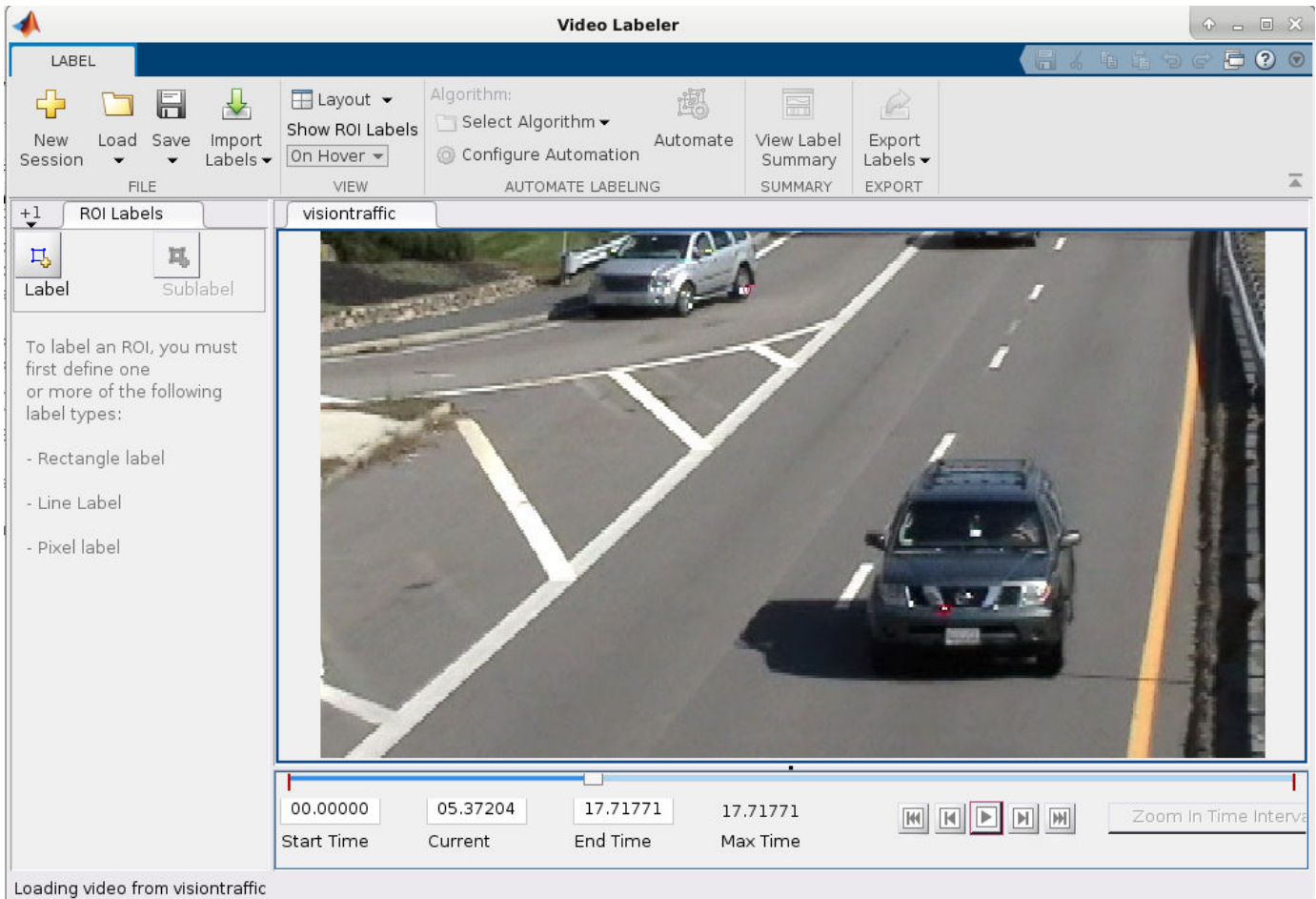
Load Unlabeled Data

Open the app and load a video of vehicles driving on a highway . Videos must be in a file format readable by `VideoReader`.

```
videoLabeler('visiontraffic.avi')
```

Alternatively, open the app from the **Apps** tab, under **Image Processing and Computer Vision**. Then, from the **Load** menu, load a video data source.

Explore the video. Click the Play button  to play the entire video, or use the slider  to navigate between frames.



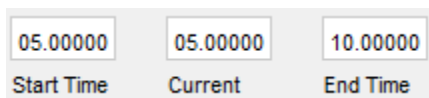
The app also enables you to load image sequences, with corresponding timestamps, by selecting **Load > Image Sequence**. The app supports all image file formats supported by `imread`. To read additional file formats, you can create an `imageDatastore` and use the `ReadFcn` property.

To load a custom data source that cannot be read by `VideoReader` or `imread`, see “Use Custom Image Source Reader for Labeling” on page 14-94.

Set Time Interval to Label

You can label the entire video or start with a portion of the video. In this example, you label a five-second time interval within the loaded video. In the text boxes below the video, enter these times in seconds:

- 1 In the **Current Time** box, type 5 and press **Enter**.
- 2 In the **Start Time** box, type 5 so that the slider is at the start of the time interval.
- 3 In the **End Time** box, type 10.



Optionally, to make adjustments to the time interval, click and drag the red interval flags.



The entire app is now set up to focus on this specific time interval. The video plays only within this interval, and labeling and automation algorithms apply only to this interval. You can change the interval at any time by moving the flags.

To expand the time interval to fill the entire playback section, click **Zoom in Time Interval**.



Create Label Definitions

Define the labels you intend to draw. In this example, you define labels directly within the app. To define labels from the MATLAB command line instead, use the `labelDefinitionCreator`.

Create ROI Labels

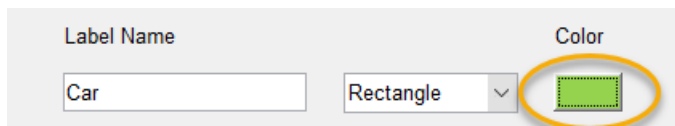
An ROI label is a label that corresponds to a region of interest (ROI). You can define these types of ROI labels.

ROI Label	Description	Example: Driving Scene
Rectangle	Draw rectangular ROI labels (bounding boxes) around objects.	
Projected cuboid	Draw cuboidal ROI labels (3-D bounding boxes).	

ROI Label	Description	Example: Driving Scene
Line	Draw linear ROI labels to represent lines. To draw a polyline ROI, use two or more points.	Lane boundaries, guard rails, road curbs 
Pixel label	Assign labels to pixels for semantic segmentation. You can label pixels manually using polygons, brushes, or flood fill. For more on pixel labeling, see “Label Pixels for Semantic Segmentation” on page 14-53.	Vehicles, road surface, trees, pavement 

In this example, you define a **vehicle** group for labeling types of vehicles, and then create a **Rectangle** ROI label for a Car and a Truck. Optionally, you can use the **Show ROI Labels** drop-down menu to select **On Hover**, **Always**, or **Never** to control how the ROI label names appear during labeling. By default, the names will appear when you hover on an ROI.

- 1 In the **ROI Labels** pane on the left, click **Label**.
- 2 Create a **Rectangle** label named Car.
- 3 Optionally, change the label color by clicking the preview color.

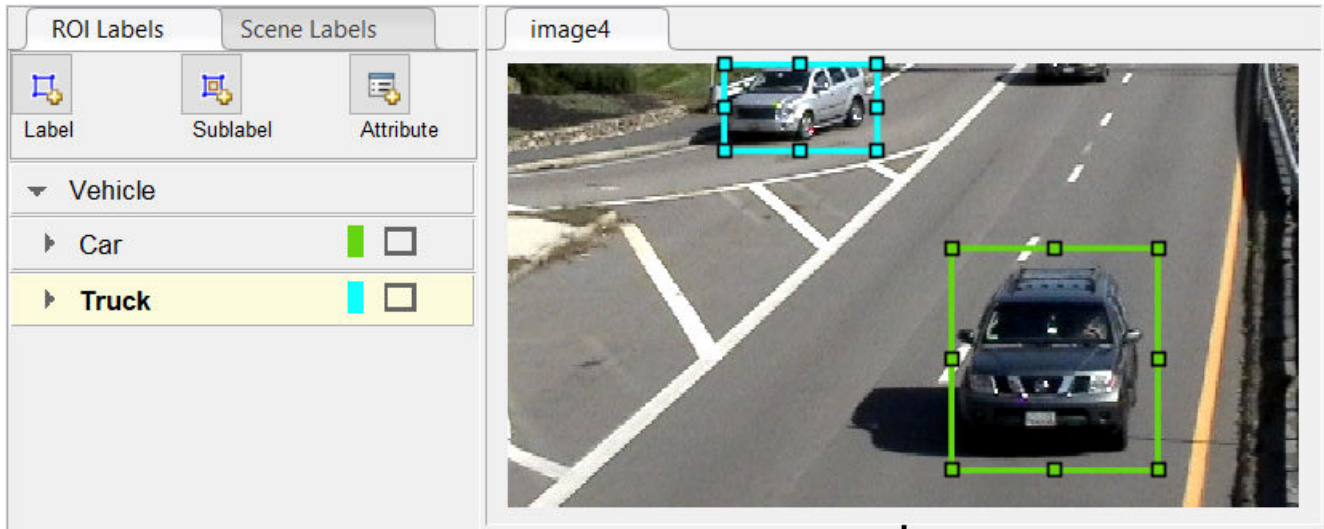


- 4 From the Group drop-down menu, select **New Group** and name the group **Vehicle**
- 5 Click **OK**.

The **Vehicle** group name appears in the **ROI Labels** pane with the label **Car** created. You can move a label in the list to a different position or group in the list by left-clicking and dragging the label up or down.

- 6 Add a second label. Click **Label**. Name the label **Truck** and make sure the **Vehicle** group is selected. Click **OK**.

- Use the mouse to draw rectangular **Car** ROIs around the two vehicles.



Create Sublabels

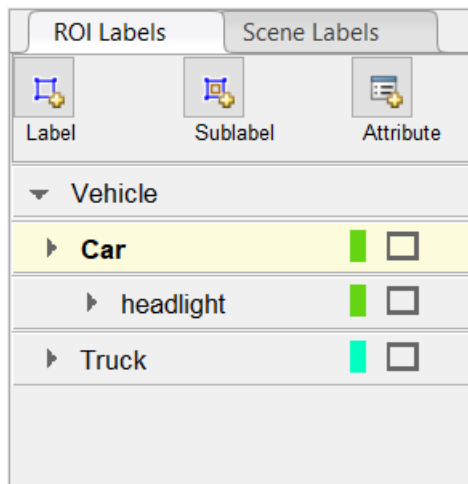
A sublabel is a type of ROI label that corresponds to a parent ROI label. Each sublabel must belong to, or be a child of, a specific label defined in the **ROI Labels** pane. For example, in a driving scene, a vehicle label might have sublabels for headlights, license plates, or wheels.

Define a sublabel for headlights.

- In the **ROI Labels** pane on the left, click the **Car** label.
- Click **Sublabel**.
- Create a Rectangle sublabel named `headlight` and optionally write a description. Click **OK**.

The **headlight** sublabel appears in the **ROI Labels** pane. The sublabel is nested under the selected ROI label, **Car**, and has the same color as its parent label.

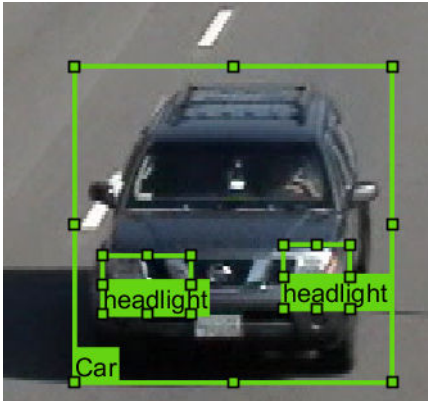
You can add multiple sublabels under a label. You can also drag-and-drop the sublabels to reorder them in the list. Right-click any label for additional edits.



- 4 In the **ROI Labels** pane, select the **headlight** sublabel.
- 5 In the image frame, select the **Car** label. The label turns yellow when selected. You must select the **Car** label (parent ROI) before you can add a sublabel to it.

Draw **headlight** sublabels for each of the cars.

- 6 Repeat the previous steps to label the headlights of the other car. To draw the labels more precisely, use the pan and zoom options located in the upper-right corner of the labeling window.



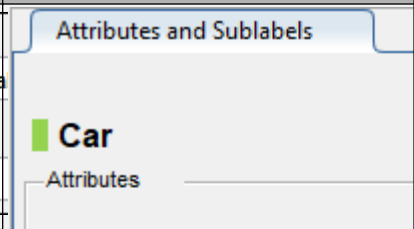
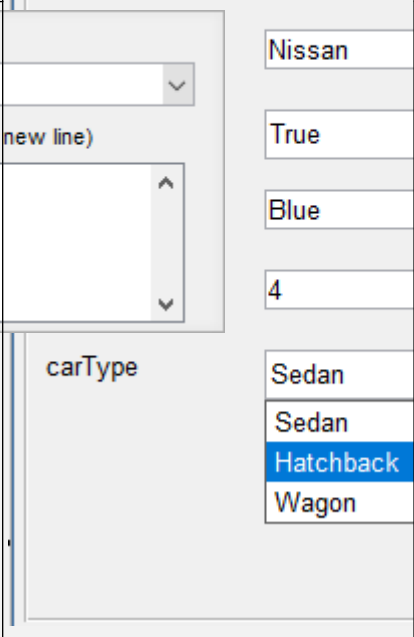
Sublabels can only be used with rectangular or polyline ROI labels and cannot have their own sublabels. For more details on working with sublabels, see “Use Sublabels and Attributes to Label Ground Truth Data” on page 14-96.

Create Attributes

An attribute provides further categorization of an ROI label or sublabel. Attributes specify additional information about a drawable label. For example, in a driving scene, attributes might include the type or color of a vehicle.

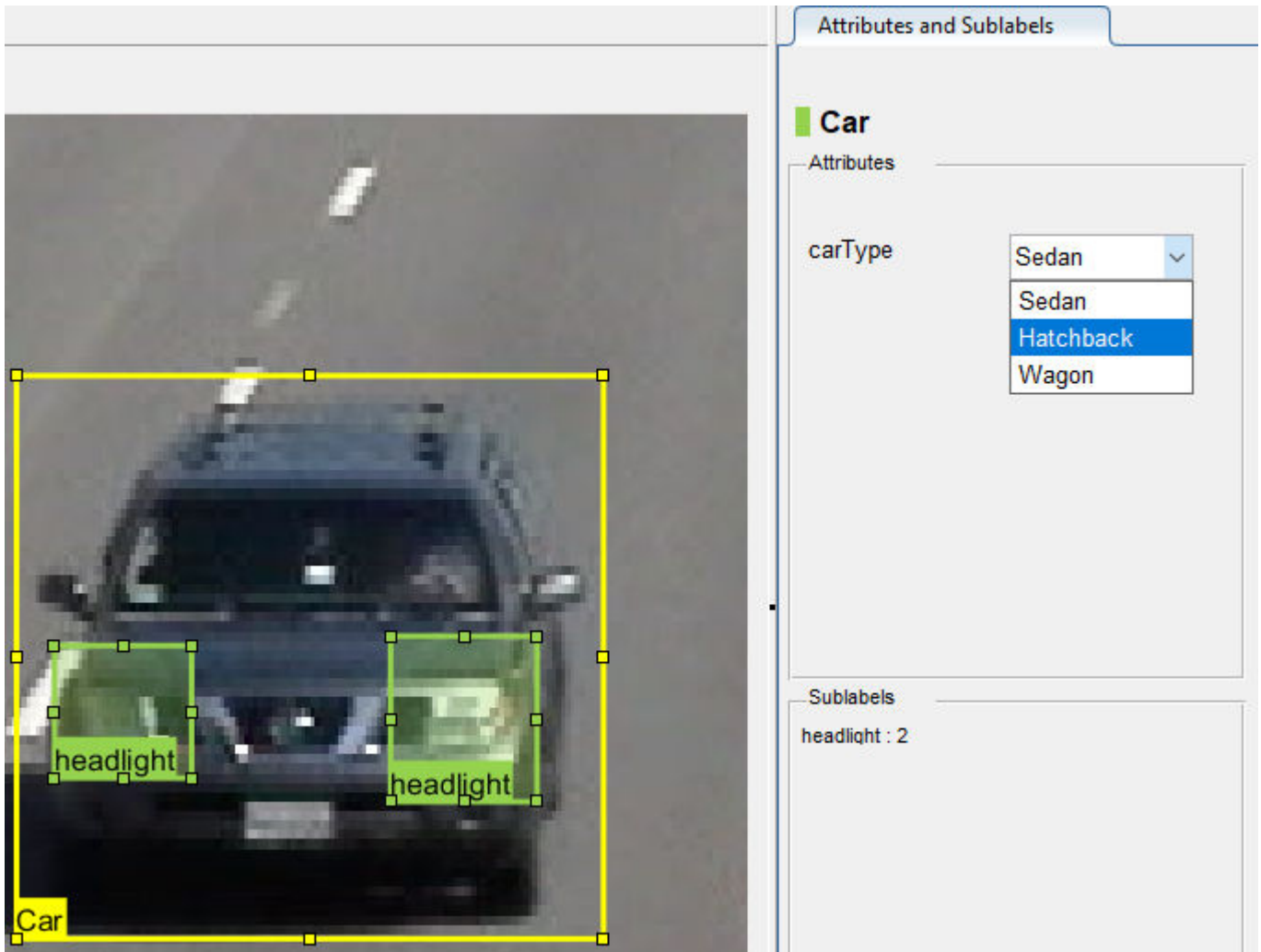
You can define these types of attributes.

Attribute Type	Sample Attribute Definition	Sample Default Values
Numeric Value	<p>Attribute Name</p> <input type="text" value="numDoors"/> <p>Default Scalar Value (Optional)</p> <input type="text" value="4"/>	
String	<p>Attribute Name</p> <input type="text" value="color"/> <p>Default Value (Optional)</p> <input type="text"/>	

Attribute Type	Sample Attribute Definition	Sample Default Values
Logical	Attribute Name <input type="text" value="inMotion"/> Logical Default Value (Optional) <input type="text" value="True"/>	
List	Attribute Name <input type="text" value="carType"/> List List Items (Each item must appear on a new line) <input type="text" value="Sedan"/> <input type="text" value="Hatchback"/> <input type="text" value="Wagon"/>	

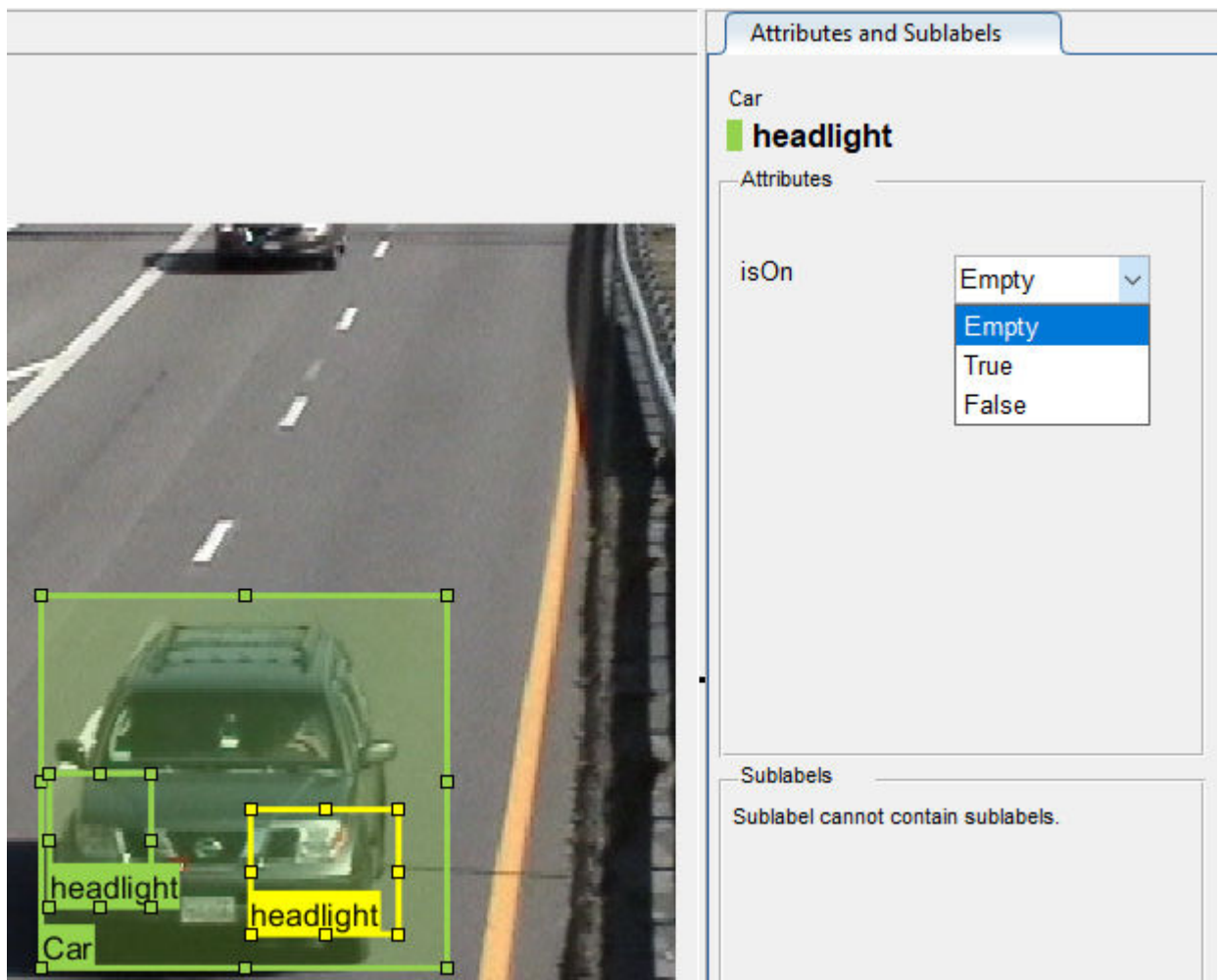
Add an attribute for the vehicle type.

- 1 In the **ROI Labels** pane on the left, select the **Car** label and click **Attribute**.
- 2 In the **Attribute Name** box, type `carType`. Set the attribute type to **List**.
- 3 In the **List Items** section, type different types of cars, such as Sedan, Hatchback, and Wagon, each on its own line. Optionally give the attribute a description, and click **OK**.
- 4 In the first frame of the video, select a **Car** ROI label. In the **Attributes and Sublabels** pane, select the appropriate **carType** attribute value for that vehicle.
- 5 Repeat the previous step to assign a **carType** attribute to the other vehicle.



You can also add attributes to sublabels. Add an attribute for the **headlight** sublabel that tells whether the headlight is on.

- 1 In the **ROI Labels** pane on the left, select the **headlight** sublabel and click **Attribute**.
- 2 In the **Attribute Name** box, type `isOn`. Set the attribute type to **Logical**. Leave the **Default Value** set to **Empty**, optionally write a description, and click **OK**.
- 3 Select a headlight in the video frame. Set the appropriate **isOn** attribute value, or leave the attribute value set to **Empty**.
- 4 Repeat the previous step to set the **isOn** attribute for the other headlights.



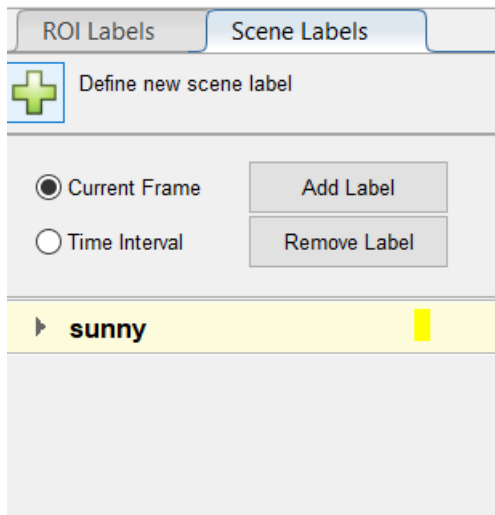
To delete an attribute, right-click an ROI label or sublabel, and select the attribute to delete. Deleting the attribute removes attribute information from all previously created ROI label annotations.

Create Scene Labels

A scene label defines additional information for the entire scene. Use scene labels to describe conditions, such as lighting and weather, or events, such as lane changes.

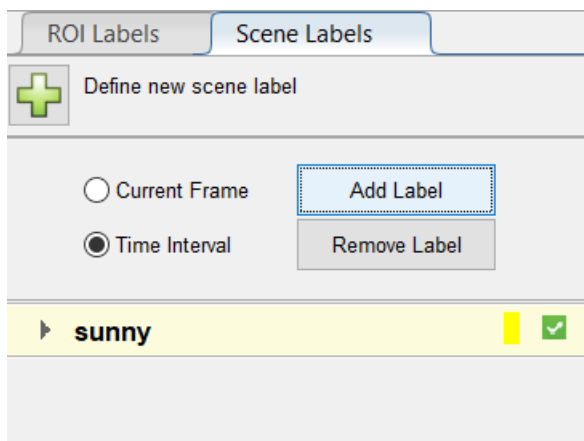
Create a scene label to use in the video.

- 1 Select the **Scene Labels** tab on the left.
- 2 Click the **Define new scene label** button, and create a scene label named sunny. Make sure **Group** is set to None. Click **OK**.



The **Scene Labels** pane shows the scene label definition.

- 3 You can apply the label to just the current frame or to an interval of frames. With the sunny scene label definition still selected in the **Scene Labels** pane, select **Time Interval**.
- 4 Click the **Add Label**. A checkmark appears for the sunny scene label indicating that the label now applies to all frames in the time interval.



- 5 To edit or delete a scene label, right-click on the label and select either **Edit Label** or **Delete Label**.

Label Ground Truth

So far, you have labeled only one frame in the video. To label the remaining frames, choose one of these options.

Label Ground Truth Manually

When you click the right arrow key to advance to the next frame, the ROI labels from the previous frame do not carry over. Only the **sunny** scene label applies to each frame, because this label was applied over the entire time interval.

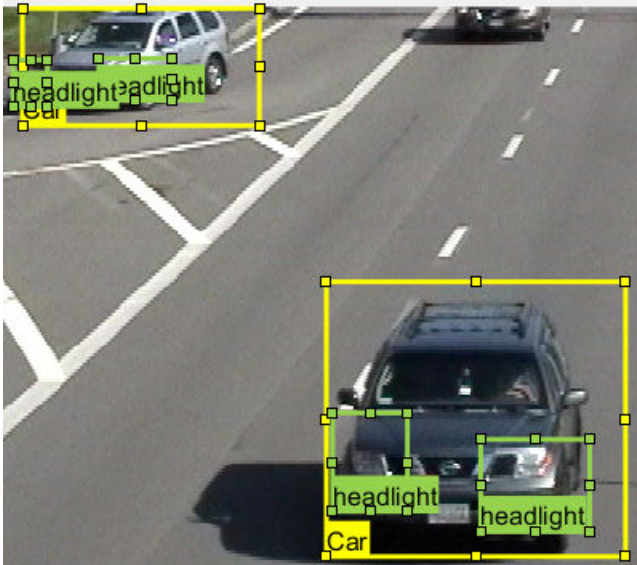
Advance frame by frame and draw the label and sublabel ROIs manually. Also update the attribute information for these ROIs.

Label Ground Truth Using Automation Algorithm

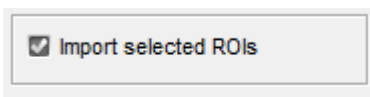
To speed up the labeling process, you can use an automation algorithm within the app. You can either define your own automation algorithm, see “Create Automation Algorithm for Labeling” on page 14-49 and “Temporal Automation Algorithms” on page 14-100, or use a built-in automation algorithm. In this example, you label the ground truth using a built-in point tracking algorithm.

In this example, you automate the labeling of only the **Car** ROI labels. The built-in automation algorithms do not support sublabel and attribute automation.

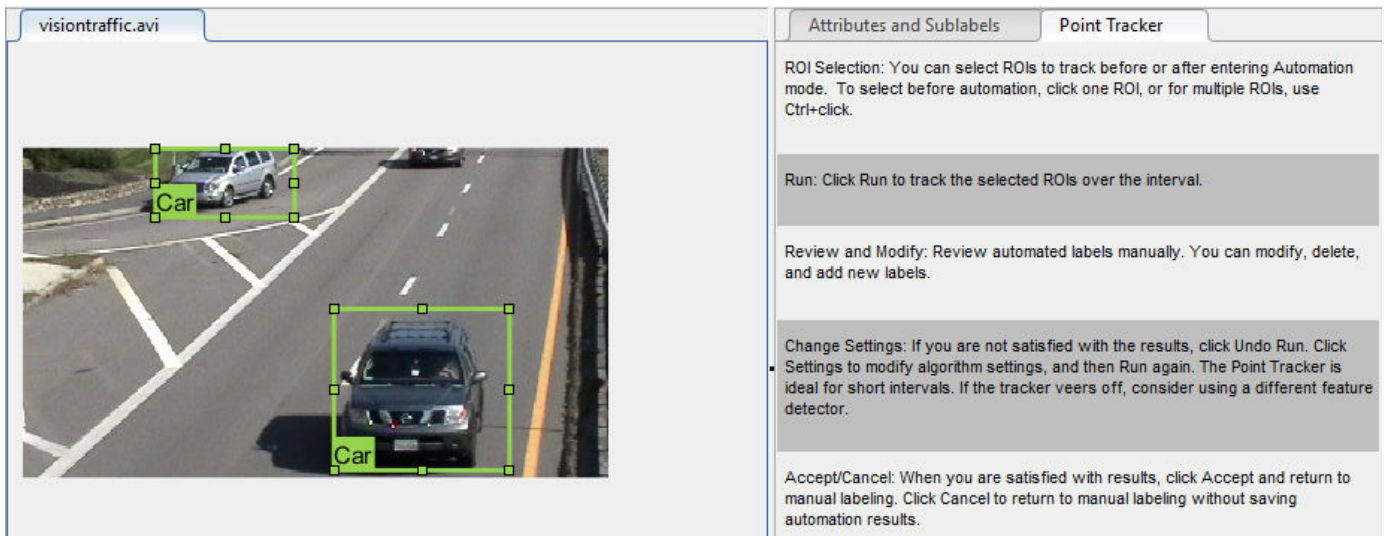
- 1 Select the labels you want to automate. In the first frame of the video, press **Ctrl** and click to select the two **Car** label annotations. The labels are highlighted in yellow.



- 2 From the app toolstrip, select **Select Algorithm > Point Tracker**. This algorithm tracks one or more rectangle ROIs over short intervals using the Kanade-Lucas-Tomasi (KLT) algorithm.
- 3 (optional) Configure the automation settings. For example, make sure that **Import selected ROIs** is selected so that the **Car** labels you selected are imported into the automation session.



- 4 Click **Automate** to open an automation session. The algorithm instructions appear in the right pane, and the selected labels are available to automate.



- 5 Click **Run** to track the selected ROIs over the interval.
- 6 Examine the results of running the algorithm.

The vehicles that enter the scene later are unlabeled. The unlabeled vehicles did not have an initial ROI label, so the algorithm did not track them. Click **Undo Run**. Use the slider to find the frames where each vehicle first appears. Draw **vehicle** ROIs around each vehicle, and then click **Run** again.

- 7 Advance frame by frame and manually move, resize, delete, or add ROIs to improve the results of the automation algorithm.

When you are satisfied with the algorithm results, click **Accept**. Alternatively, to discard labels generated during the session and label manually instead, click **Cancel**. The **Cancel** button cancels only the algorithm session, not the app session.

Optionally, you can now manually label the remaining frames with sublabel and attribute information.

To further evaluate your labels, you can view a visual summary of the labeled ground truth. From the app toolbar, select **View Label Summary**. Use this summary to compare the frames, frequency of labels, and scene conditions. For more details, see “View Summary of Ground Truth Labels” on page 14-102. This summary does not support sublabels or attributes.

Export Labeled Ground Truth

You can export the labeled ground truth to a MAT-file or to a variable in the MATLAB workspace. In both cases, the labeled ground truth is stored as a `groundTruth` object. You can use this object to train a deep-learning-based computer vision algorithm. For more details, see “Training Data for Object Detection and Semantic Segmentation” on page 14-45.

Note If you export pixel data, the pixel label data and ground truth data are saved in separate files but in the same folder. For considerations when working with exported pixel labels, see “How Labeler Apps Store Exported Pixel Labels” on page 14-16.

In this example, you export the labeled ground truth to the MATLAB workspace. From the app toolbar, select **Export Labels > To Workspace**. The exported MATLAB variable is `gTruth`.

Display the properties of the exported `groundTruth` object. The information in your exported object might differ from the information shown here.

```
gTruth
gTruth =
    groundTruth with properties:
        DataSource: [1x1 groundTruthDataSource]
        LabelDefinitions: [2x6 table]
        LabelData: [531x3 timetable]
```

Data Source

`DataSource` is a `groundTruthDataSource` object containing the path to the images or video and timestamps. Display the properties of this object.

```
gTruth.DataSource
ans =
    groundTruthDataSource for a video file with properties
        Source: ...matlab\toolbox\vision\visiondata\visiontraffic.avi
        TimeStamps: [531x1 duration]
```

Label Definitions

`LabelDefinitions` is a table containing information about the label definitions. This table does not contain information about the labels that are drawn on the video frames. To save the label definitions in their own MAT-file, from the app toolstrip, select **Save > Label Definitions**. You can then import these label definitions into another app session by selecting **Import Files**.

Display the label definitions table. Each row contains information about an ROI label definition or a scene label definition. If you exported pixel label data, the `LabelDefinitions` table also includes a `PixelLabelID` column containing the ID numbers for each pixel label definition.

```
gTruth.LabelDefinitions
ans =
    3x6 table

    Name          Type          LabelColor    Group          Description    Hierarchy
    _____  _____  _____  _____  _____  _____
    {'Car' }      Rectangle    {1x3 double} {'Vehicle'}    {0x0 char}    {1x1 struct}
    {'Truck'}     Rectangle    {1x3 double} {'Vehicle'}    {0x0 char}    {0x0 double}
    {'Sunny'}     Scene        {1x3 double} {'Weather'}    {0x0 char}    {0x0 double}
```

Within `LabelDefinitions`, the `Hierarchy` column stores information about the sublabel and attribute definitions of a parent ROI label.

Display the sublabel and attribute information for the Car label.

```
gTruth.LabelDefinitions.Hierarchy{1}
ans =
```

```

struct with fields:
  numDoors: [1x1 struct]
  color: [1x1 struct]
  inMotion: [1x1 struct]
  carType: [1x1 struct]
  headlight: [1x1 struct]
    Type: Rectangle
  Description: ''

```

Display information about the `headlight` sublabel.

```
gTruth.LabelDefinitions.Hierarchy{1}.headlight
```

```
ans =
```

```

struct with fields:
  Type: Rectangle
  Description: ''
  Color: [0.5862 0.8276 0.3103]
  isOn: [1x1 struct]

```

Display information about the `carType` attribute.

```
gTruth.LabelDefinitions.Hierarchy{1}.carType
```

```
ans =
```

```

struct with fields:
  ListItems: {3x1 cell}
  Description: ''

```

Label Data

`LabelData` is a timetable containing information about the ROI labels drawn at each timestamp, across the entire video. The timetable contains one column per label.

Display the first few rows of the timetable. The first few timestamps indicate that no vehicles were detected and that the sunny scene label is `false`. These results are because this portion of the video was not labeled. Only the time interval of 5-10 seconds was labeled.

```
labelData = gTruth.labelData;
head(labelData)
```

```
ans =
```

```
8x3 timetable
```

Time	Car	Truck	sunny
5.005 sec	[1x2 struct]	[1x0 struct]	true
5.0384 sec	[1x2 struct]	[1x0 struct]	true
5.0717 sec	[1x2 struct]	[1x0 struct]	true
5.1051 sec	[1x2 struct]	[1x0 struct]	true
5.1385 sec	[1x2 struct]	[1x0 struct]	true
5.1718 sec	[1x2 struct]	[1x0 struct]	true

```

5.2052 sec    [1x2 struct]    [1x0 struct]    true
5.2386 sec    [1x2 struct]    [1x0 struct]    true

```

Display the first few timetable rows from the 5-10 second interval that contains labels.

```

gTruthInterval = labelData(timerange('00:00:05','00:00:10'),:);
head(gTruthInterval)

```

```
ans =
```

```
8x3 timetable
```

Time	Car	Truck	sunny
5.005 sec	[1x2 struct]	[1x0 struct]	true
5.0384 sec	[1x2 struct]	[1x0 struct]	true
5.0717 sec	[1x2 struct]	[1x0 struct]	true
5.1051 sec	[1x2 struct]	[1x0 struct]	true
5.1385 sec	[1x2 struct]	[1x0 struct]	true
5.1718 sec	[1x2 struct]	[1x0 struct]	true
5.2052 sec	[1x2 struct]	[1x0 struct]	true
5.2386 sec	[1x2 struct]	[1x0 struct]	true

For each Car label, the structure includes the position of the bounding box and information about its sublabels and attributes.

Display the bounding box positions for the vehicles at the start of the time interval. Your bounding box positions might differ from the ones shown here.

```
gTruthInterval(1,:).Car{1}.Position % [x y width height], in pixels
```

```
ans =
```

```
1x4 single row vector
```

```
415.8962    82.4737   130.8474   129.3805
```

```
ans =
```

```
1x4 single row vector
```

```
235.2182    1.0000   117.0611    55.3500
```

Save App Session

From the app toolstrip, select **Save** and save a MAT-file of the app session. The saved session includes the data source, label definitions, and labeled ground truth. It also includes your session preferences, such as the layout of the app. To change layout options, select **Layout**.

At any time during a session, you can select **New Session** to start a new session. You have the option of saving the current session or cancelling.

The app session MAT-file is separate from the ground truth MAT-file that is exported when you select **Export > From File**. To share labeled ground truth data, as a best practice, share the ground truth

MAT-file containing the `groundTruth` object, not the app session MAT-file. For more details, see “Share and Store Labeled Ground Truth Data” on page 14-106.

See Also

Apps **Video Labeler**

Objects

`groundTruth` | `groundTruthDataSource` | `labelDefinitionCreator` |
`vision.labeler.AutomationAlgorithm` | `vision.labeler.mixin.Temporal`

More About

- “Use Custom Image Source Reader for Labeling” on page 14-94
- “Keyboard Shortcuts and Mouse Actions for Video Labeler” on page 14-116
- “Use Sublabels and Attributes to Label Ground Truth Data” on page 14-96
- “Label Pixels for Semantic Segmentation” on page 14-53
- “Create Automation Algorithm for Labeling” on page 14-49
- “View Summary of Ground Truth Labels” on page 14-102
- “Share and Store Labeled Ground Truth Data” on page 14-106
- “Training Data for Object Detection and Semantic Segmentation” on page 14-45

Use Custom Image Source Reader for Labeling

In this section...

“Create Custom Reader Function” on page 14-94

“Import Data Source into Video Labeler App” on page 14-94

“Import Data Source into Ground Truth Labeler App” on page 14-95

The **Video Labeler** and **Ground Truth Labeler** (requires Automated Driving Toolbox) apps enable you to label ground truth data in a video or sequence of images.

You can use a custom reader to import any video or sequence of images that is supported by the `VideoReader` object or `imread` function. First, create a custom reader function. Then, load the custom reader function and corresponding image data source into the **Video Labeler** or **Ground Truth Labeler** app. The **Image Labeler** app does not support custom data source readers.

Create Custom Reader Function

First, specify a custom reader as a function handle. The custom reader must have this syntax.

```
outputImage = readerFcn(sourceName,currentTimestamp)
```

In this example, `readerFcn` is the name of the custom reader function.

The custom reader function loads an image from `sourceName`, which corresponds to the current timestamp specified by `currentTimestamp`. For example, suppose you want to load the image at the third timestamp for a `timestamps` duration vector that runs from 1 to 5 seconds. To specify `currentTimestamp`, at the MATLAB command prompt, enter this code.

```
timestamps = seconds(1:5);
currIdx = 3;
currentTimestamp = timestamps(currIdx);
```

The `outputImage` output from the custom function must be a grayscale or RGB image in any format supported by the `imshow` function. The `currentTimestamp` output is a scalar value that corresponds to the current frame that the function is executing.

Import Data Source into Video Labeler App

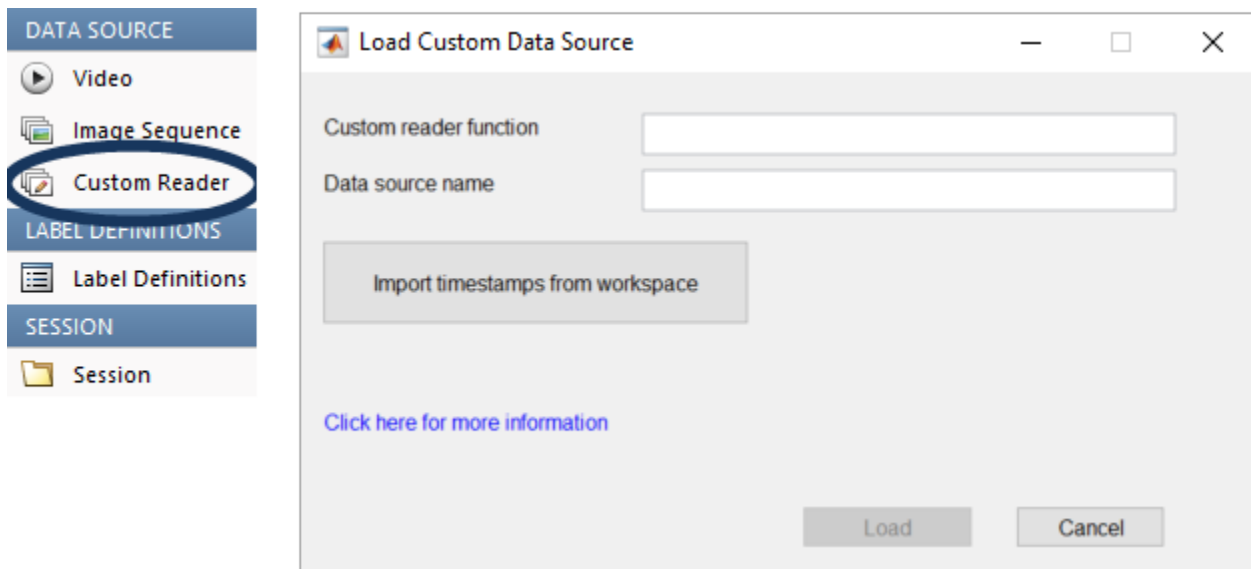
To import a custom data source into the **Video Labeler** app, first create a `groundTruthDataSource` object. This object stores the data source files and timestamps. Specify the name of the data source, the custom reader function handle that reads the data, and the timestamps by using this syntax.

```
gtSource = groundTruthDataSource(sourceName, readerFcn, timestamps)
```

To load this object into the app, at the MATLAB command prompt, enter this code.

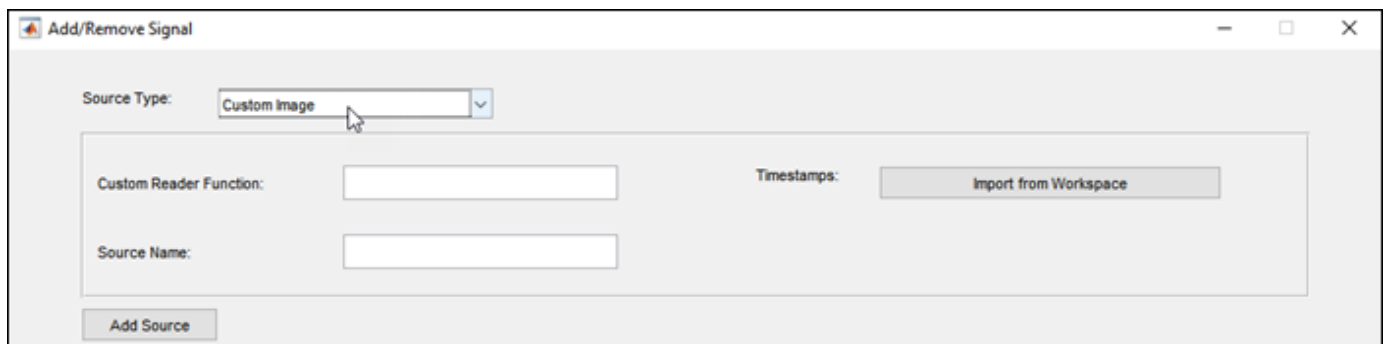
```
videoLabeler(gtSource)
```

Alternatively, on the toolstrip of the **Video Labeler** app, select **Load > Custom Reader**. Then, in the Load Custom Data Source dialog box, specify **Custom reader function** as a function handle and also specify **Data source name**. In addition, you must import corresponding timestamps from the MATLAB workspace.



Import Data Source into Ground Truth Labeler App

To import the custom image data source into the **Ground Truth Labeler** app, on the app toolbar, select **Open > Add Signals**. Then, in the dialog box, set **Source Type** to Custom Image. You can then specify the custom reader function, data source name, and timestamps, and then click **Add Source** to load the image data source.



See Also

Apps

Ground Truth Labeler | Video Labeler

Objects

groundTruth | groundTruthDataSource | groundTruthMultisignal

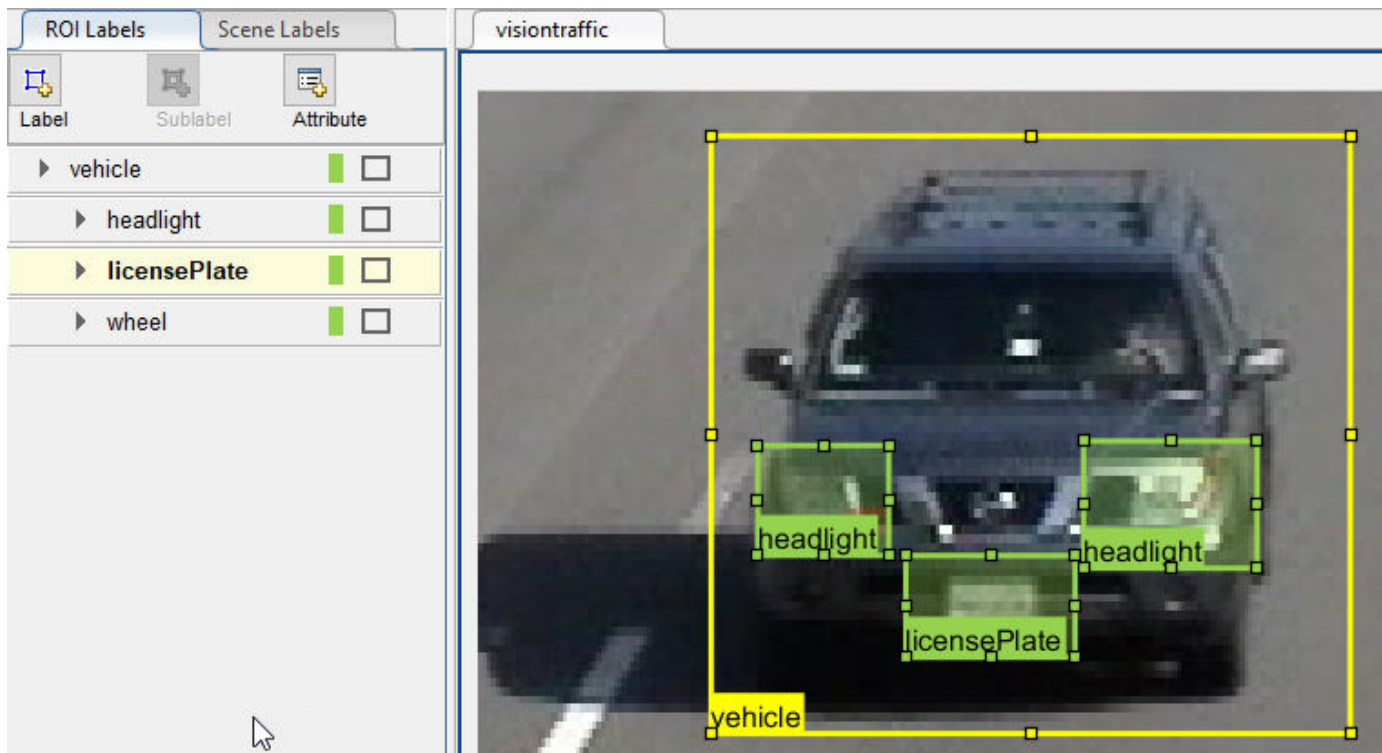
More About

- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Get Started with the Video Labeler” on page 14-78

Use Sublabels and Attributes to Label Ground Truth Data

In the **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** (requires Automated Driving Toolbox) apps, a sublabel is a type of label for drawing regions of interest (ROIs) around objects that belong to a parent label. You can use sublabels to provide a greater level of detail about the ROIs in your labeled ground truth data. For example:

- For a **bird** label, you can define **wing** or **beak** sublabels.
- For a **vehicle** label, you can define **headlight**, **licensePlate**, and **wheel** sublabels.



When to Use Sublabels vs. Attributes

A sublabel can be anything that is drawable and is part of a parent label. An attribute provides information about labels. However, attributes are not drawable and they can be associated with either a label or a sublabel.

Consider the possible sublabel and attribute candidates for the label **vehicle**:

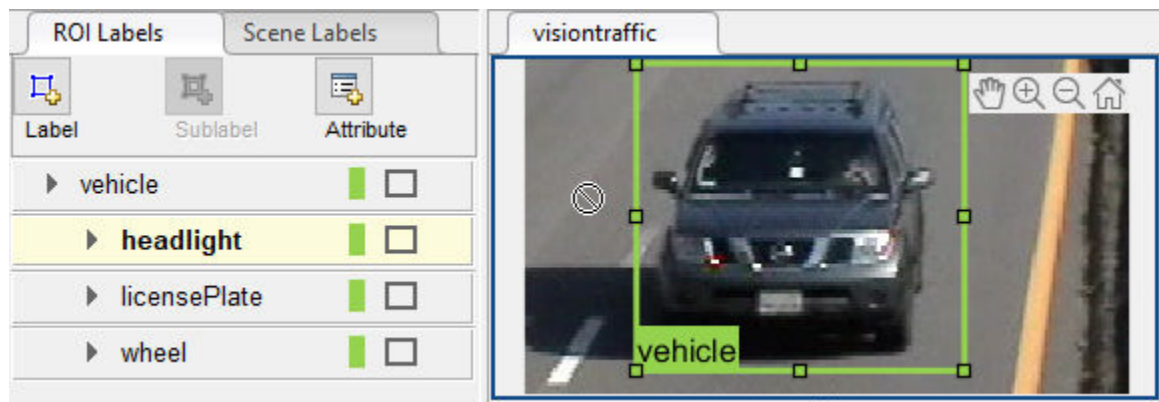
- A **wheel** is a good candidate for a *sublabel*. A wheel is part of a vehicle, and you can draw a label around a wheel.
- **Vehicle color** is a good candidate for an *attribute*. You cannot draw a label around the color of a vehicle.
- **Vehicle type** (car, truck, and so on) is a good candidate for an *attribute*. Although you can draw a label around cars and trucks, they are not part of a vehicle. Instead, you can define a list attribute with types car and truck, or define logical attributes named isCar, isTruck, and so on.

Draw Sublabels

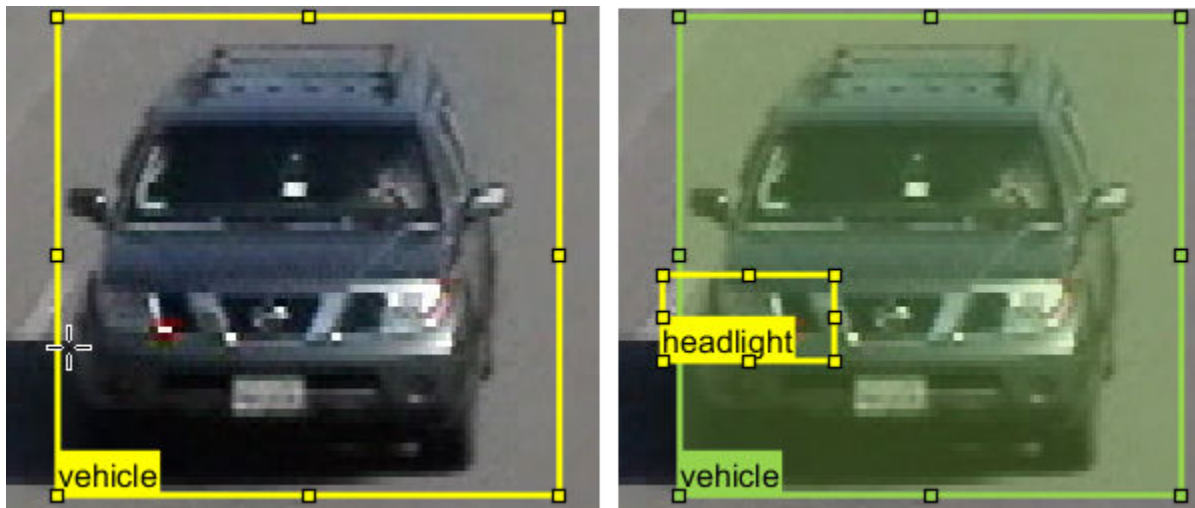
Within each frame, each sublabel that you draw must be associated with a parent label. Therefore, before you can draw a sublabel on a frame, you must:

- 1 From the **ROI Label Definition** pane, select the type of sublabel that you want to draw.
- 2 Within the frame, select a parent ROI label.

For example, to label the headlights of a vehicle, you must first select the **headlight** sublabel definition. On the frame, however, you cannot yet create a sublabel.



After you select a vehicle label on the frame, you can draw a sublabel that is associated with that vehicle. Once you create a sublabel, you cannot add another sublabel to the vehicle unless you select the vehicle label again.



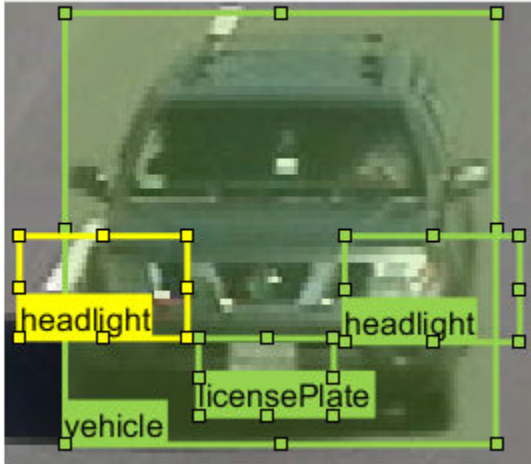
Notice that sublabels do not have to be completely enclosed within the parent label. You can drag sublabels outside the bounds of the parent label and the parent-child relationship remains unchanged.

Copy and Paste Sublabels

When labeling, it is common to copy (**Ctrl+C**) and paste (**Ctrl+V**) labels from one frame into another.

If you copy a sublabel into another frame, the parent label is copied over as well. That way, the parent-child relationship is maintained between frames. Any sublabels that you did not select to copy do not appear in the new frame.

Copy Sublabel

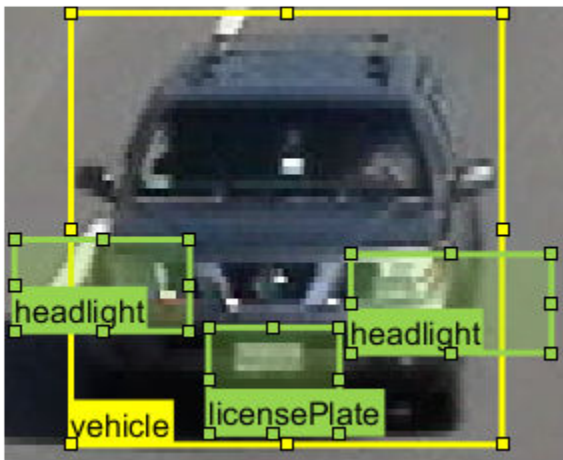


Paste to Next Frame



If you copy a parent label, however, the associated sublabels are not copied over.

Copy Label



Paste to Next Frame



Delete Sublabels

To delete an ROI sublabel from a frame, right-click the sublabel and select the **Delete** option for the sublabel shape.

To delete an ROI sublabel definition, from the **ROI Label Definition** pane, right-click the sublabel and select **Delete**.

Caution If you delete a sublabel, all ROI sublabel annotations currently on the frames are deleted as well. Attribute definitions for that sublabel are deleted as well.

Sublabel Limitations

- Sublabels can be used only with rectangle and polyline labels.
- Sublabels cannot have their own sublabels.
- The built-in automation algorithms do not support sublabel automation.
- When you click **View Label Summary**, the Label Summary window does not display sublabel information.

See Also

Apps

Ground Truth Labeler | Image Labeler | Video Labeler

More About

- “Get Started with the Image Labeler” on page 14-63
- “Get Started with the Video Labeler” on page 14-78
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Label Pixels for Semantic Segmentation” on page 14-53
- “Automate Attributes of Labeled Objects” (Automated Driving Toolbox)

Temporal Automation Algorithms

The labeling apps in Computer Vision Toolbox, Lidar Toolbox, and Automated Driving Toolbox enable you to create and import a custom automation algorithm to automatically label your data. Automation algorithms can be time-independent or time-dependent.

- Time-independent (nontemporal) algorithms can operate independently on each timestamp (or image). For example, a detection algorithm, such as the built-in people detector, is a time-independent algorithm.
- Time-dependent (temporal) algorithms have a dependence on the timestamp of execution. For example, a tracking algorithm, such as the temporal built-in Point Tracker, uses tracking from a previous time stamp to track objects in the current time stamp.

The **Image Labeler** app supports only nontemporal algorithms. The **Video Labeler**, **Lidar Labeler**, and **Ground Truth Labeler** apps support nontemporal and temporal algorithms.

Create Temporal Automation Algorithm

To create a temporal automation algorithm to use with a labeling app, on the app toolstrip, select **Select Algorithm > Add Algorithm > Create New Algorithm**. A class template opens, enabling you to define your algorithm. By default, the class inherits from the `vision.labeler.AutomationAlgorithm` and `vision.labeler.mixin.Temporal` classes, as shown by the class definition of the template:

```
classdef MyCustomAlgorithm < vision.labeler.AutomationAlgorithm && vision.labeler.mixin.Temporal
```

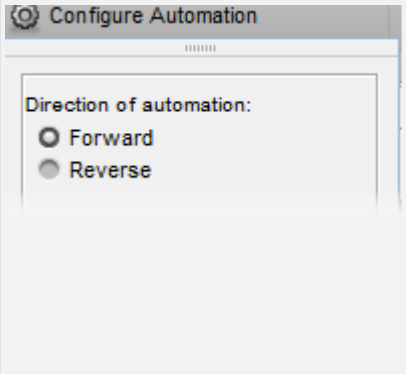
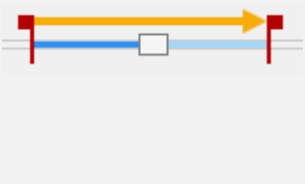
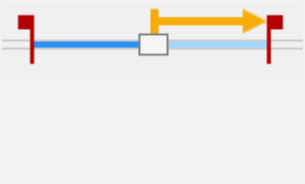
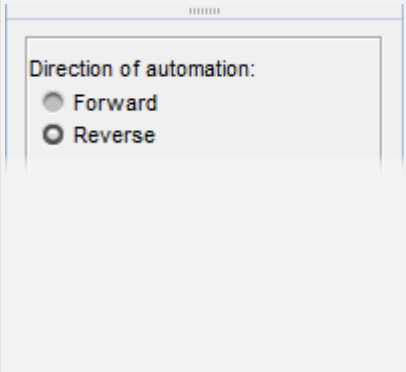
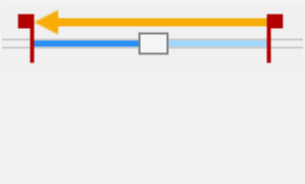
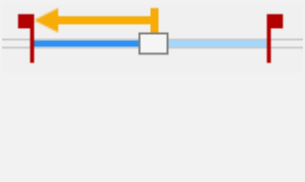
Time-based algorithms must inherit from both of these classes. Inheriting from the temporal mixin class enables you to access properties such as `StartTime`, `CurrentTime` and `EndTime` to design time-based algorithms. For more details on enabling temporal properties, see the `vision.labeler.mixin.Temporal` class reference page. For more details on defining custom automation algorithms in general, see the `vision.labeler.AutomationAlgorithm` class reference page.

After creating your algorithm, follow the instructions in the class template on where to save the algorithm.

Run Temporal Automation Algorithm

To run your temporal algorithm from the labeling, first refresh the algorithm list. On the app toolstrip, select **Select Algorithm > Refresh list**. Then, reopen the **Select Algorithm** list, select your algorithm, and run it on your data as you would any of the built-in automation algorithms.

For temporal algorithms, you can additionally configure the direction of automation. Click **Configure Automation**. By default, automation algorithms apply labels from the start of the time interval to the end. To change the direction and start time of the algorithm, choose one of the options shown in this table.

Direction of automation	Run automation from	Example
	Run automation from: <input checked="" type="radio"/> Start time to End time <input type="radio"/> Current time to End time	
	Run automation from: <input type="radio"/> End time to Start time <input checked="" type="radio"/> Current time to Start time	
	Run automation from: <input checked="" type="radio"/> Start time to End time <input type="radio"/> Current time to End time	
	Run automation from: <input type="radio"/> End time to Start time <input checked="" type="radio"/> Current time to Start time	

See Also

Apps

[Ground Truth Labeler](#) | [Lidar Labeler](#) | [Video Labeler](#)

Functions

[vision.labeler.AutomationAlgorithm](#) | [vision.labeler.mixin.Temporal](#)

Related Examples

- “Get Started with the Video Labeler” on page 14-78
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Get Started with the Lidar Labeler” (Lidar Toolbox)
- “Automate Ground Truth Labeling for Semantic Segmentation” (Automated Driving Toolbox)
- “Automate Ground Truth Labeling of Lane Boundaries” (Automated Driving Toolbox)

View Summary of Ground Truth Labels

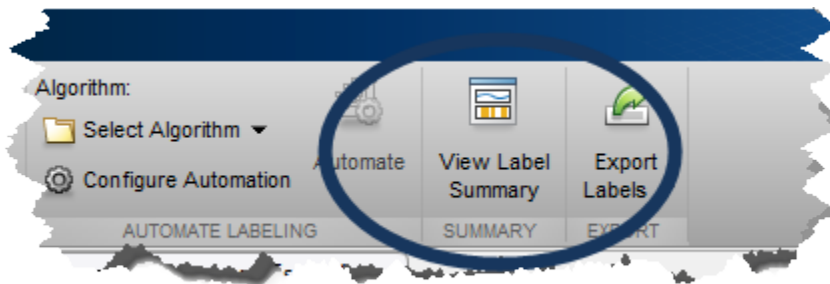
In this section...

“View Label Summary” on page 14-102

“Compare Selected Labels” on page 14-104

You can use the **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** (requires Automated Driving Toolbox) apps to interactively label ground truth data in image collections, videos, image sequences, or lidar point clouds. For details about the supported data sources, see “Choose an App to Label Ground Truth Data” on page 14-75.

You can view and compare the distribution of ROI and scene labels by clicking **View Label Summary** on the app toolstrip.

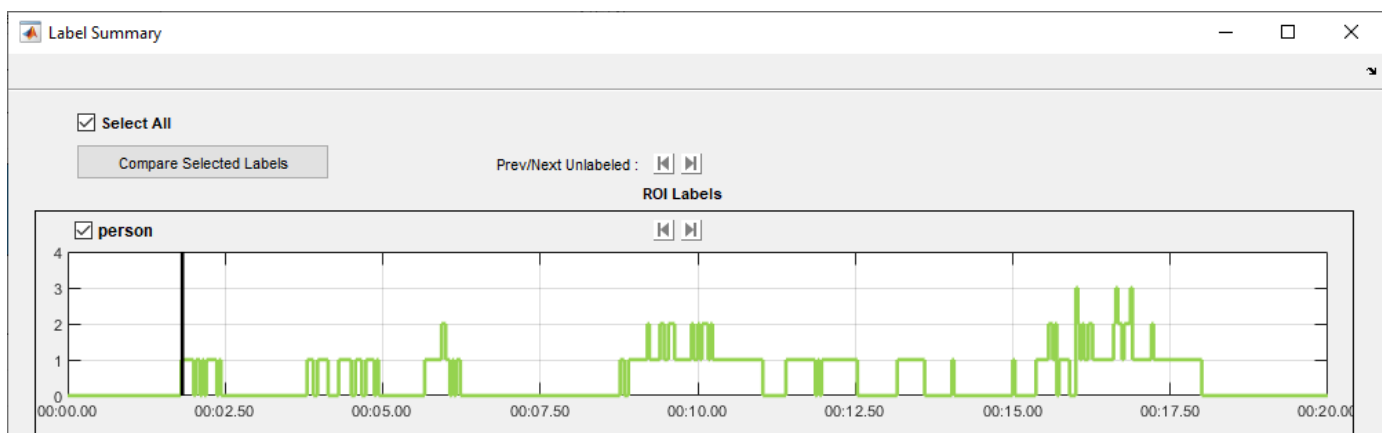


View Label Summary

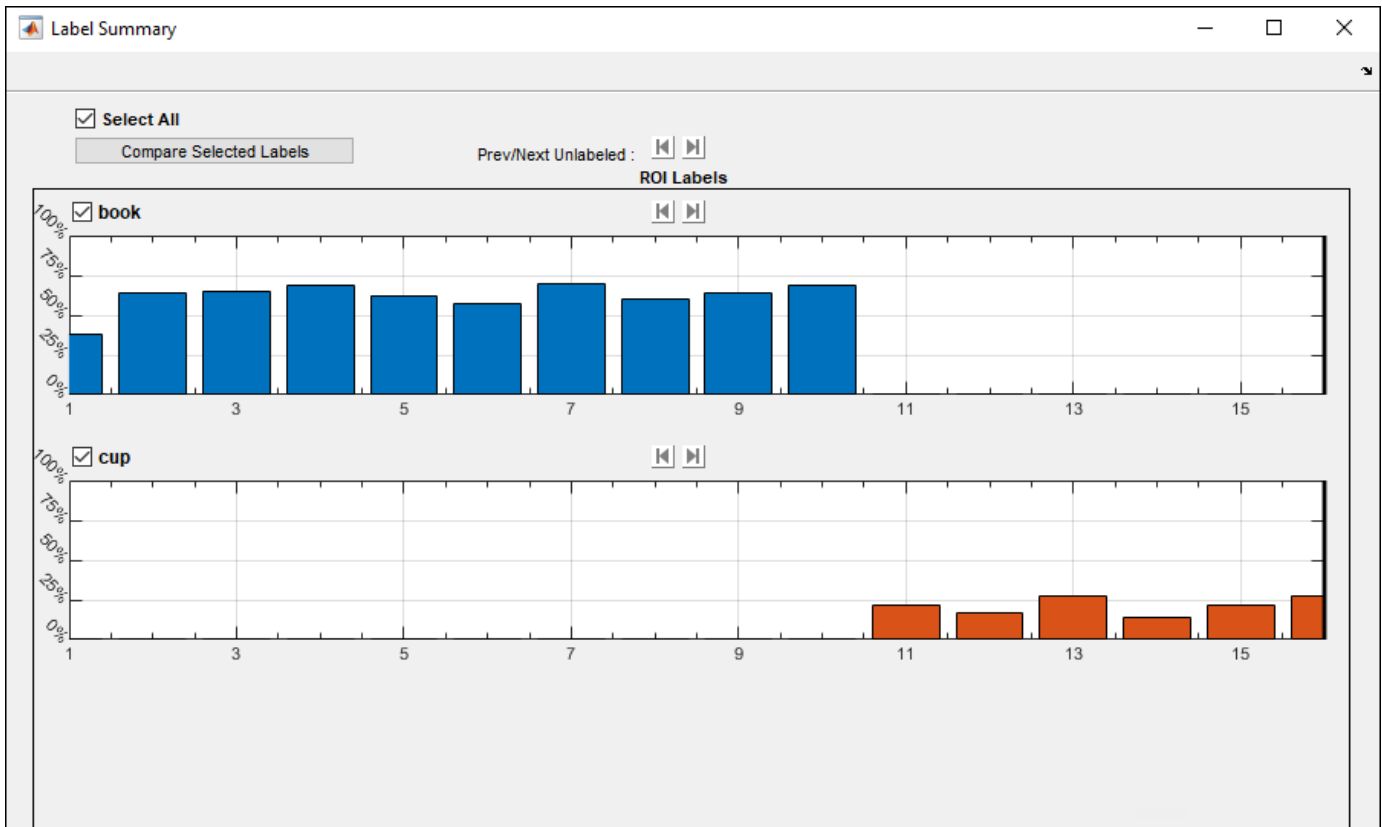
Clicking **View Label Summary** opens dockable distribution graphs for the ROI and scene labels.

The x-axis of the graph displays the timestamps across the duration of the video, image sequence, or lidar signal. Units are in seconds. For image collections (**Image Labeler** app only), the x-axis displays the numeric ID of each image in the collection.

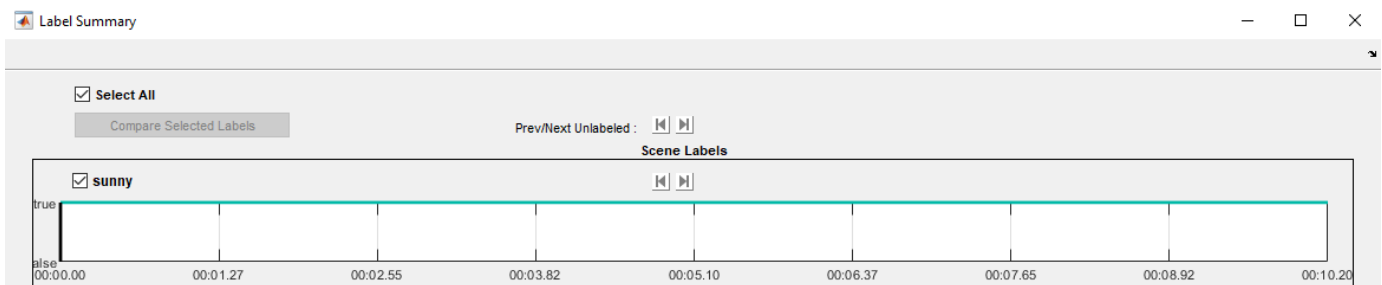
For all ROI labels except pixels, the y-axis displays the number of ROIs at each timestamp or for each image. The visual summary does not include information about sublabels or label attributes.



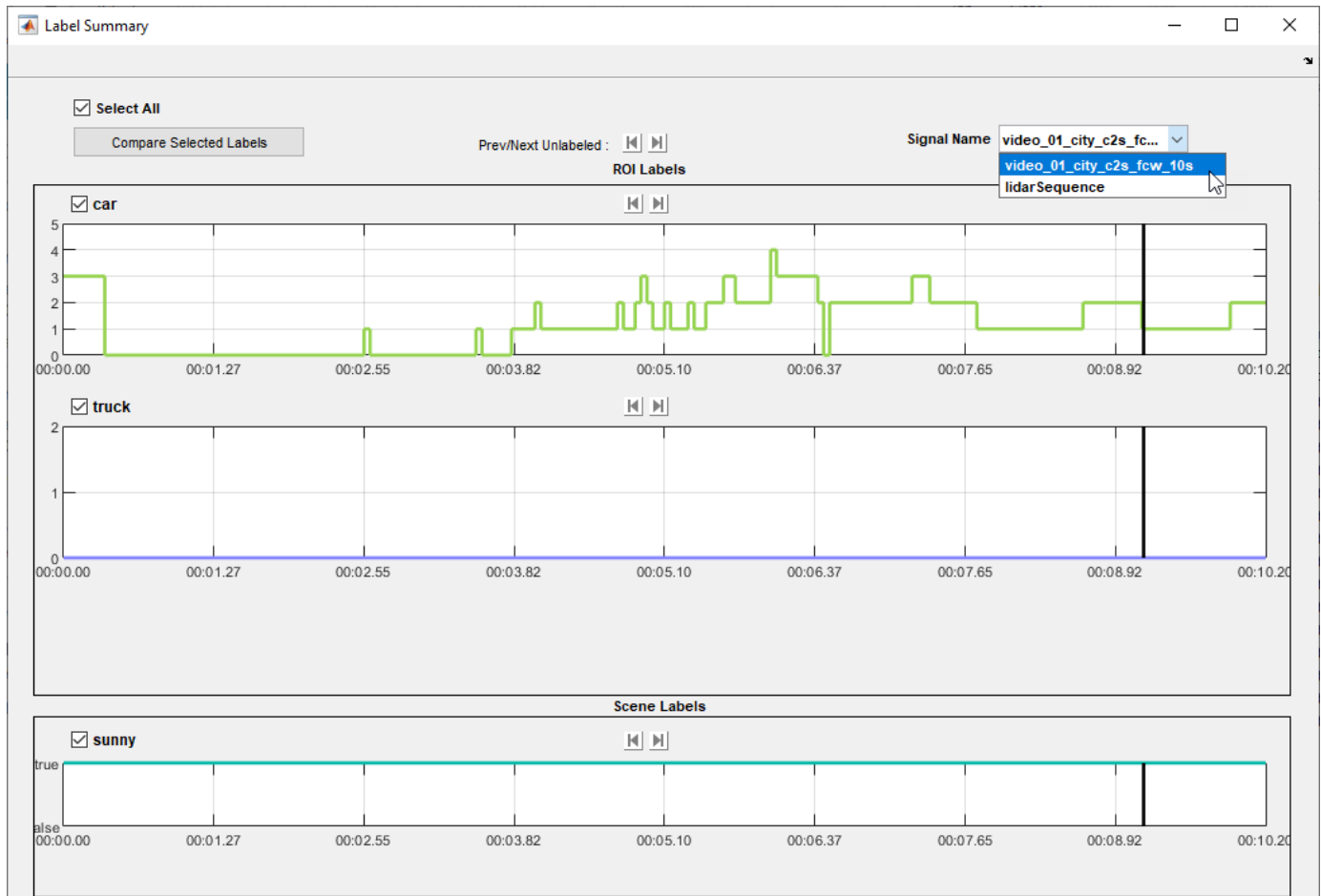
For pixel ROI labels, the y-axis displays the percentage of the frame that is labeled for each pixel label.



For scene labels, the graph displays the presence or absence of a scene label at each timestamp or for each image in a collection.

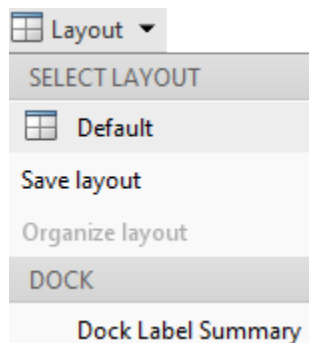


In the **Ground Truth Labeler** app, you can view labels by signal. From **Signal Name**, select a signal to view a summary of the labels for that signal.



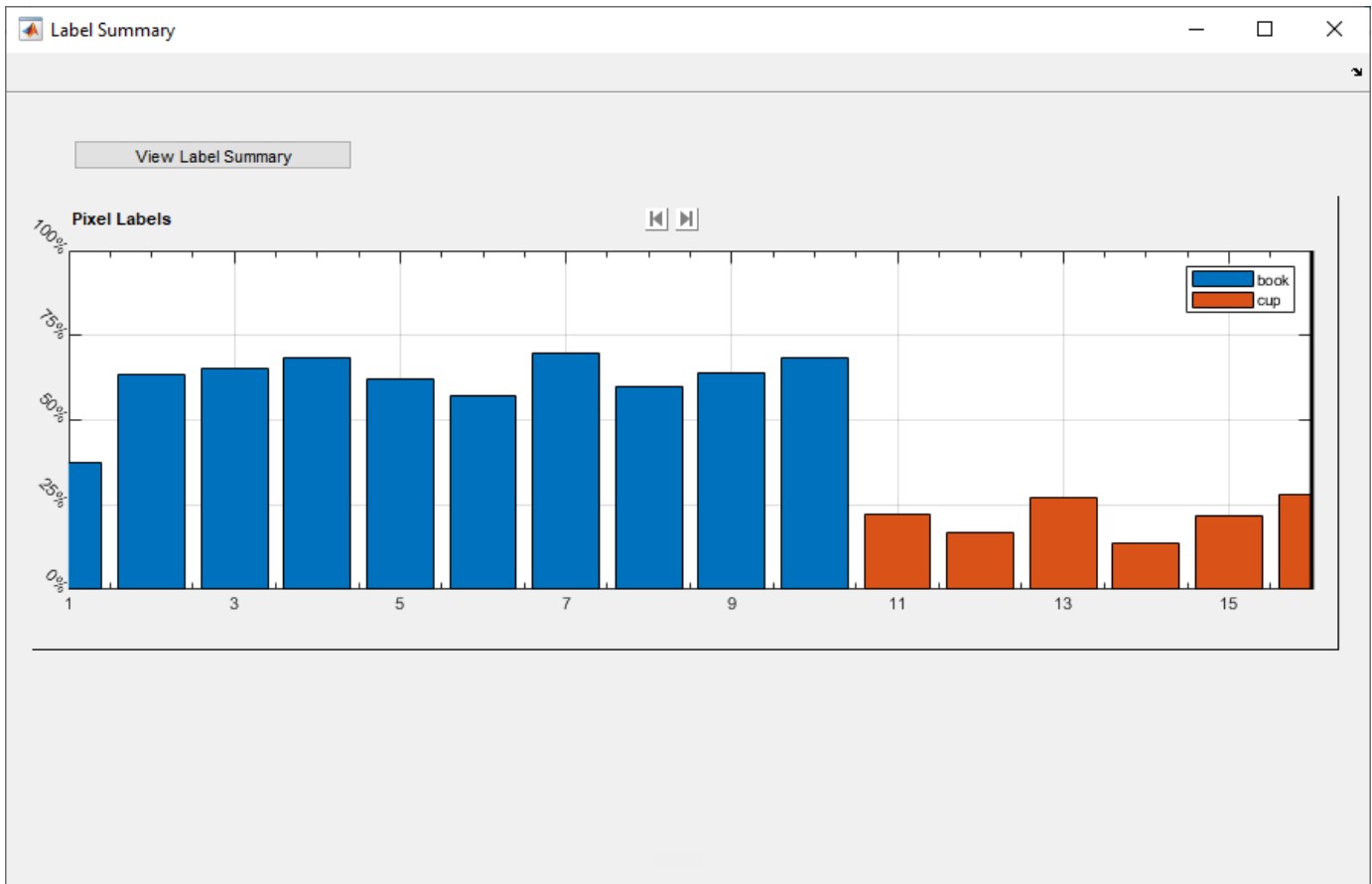
Use the graphs to examine the occurrence of labels over time or in relation to each other. Drag the black vertical line in any graph to move to a different timestamp or image in a collection.

To dock the Label Summary window in your workspace, select **Layout > Dock Label Summary**.



Compare Selected Labels

To selectively compare labels, select specific label check boxes and then click **Compare Selected Labels**. The Label Summary window displays ROI labels selected for comparison on a single graph.



See Also

Apps

[Ground Truth Labeler](#) | [Image Labeler](#) | [Video Labeler](#)

Objects

[groundTruth](#) | [groundTruthMultisignal](#)

More About

- “Choose an App to Label Ground Truth Data” on page 14-75
- “Get Started with the Image Labeler” on page 14-63
- “Get Started with the Video Labeler” on page 14-78
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)

Share and Store Labeled Ground Truth Data

The **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** (requires Automated Driving Toolbox) apps enable you to label images, videos, and other ground truth data sources. You can then export the ground truth labels as a `groundTruth` object or, for the **Ground Truth Labeler** app, a `groundTruthMultisignal` object. The ground truth object contains information about the:

- Data source (or data sources)
- Label definitions
- Drawn ground truth labels

You can share this object with:

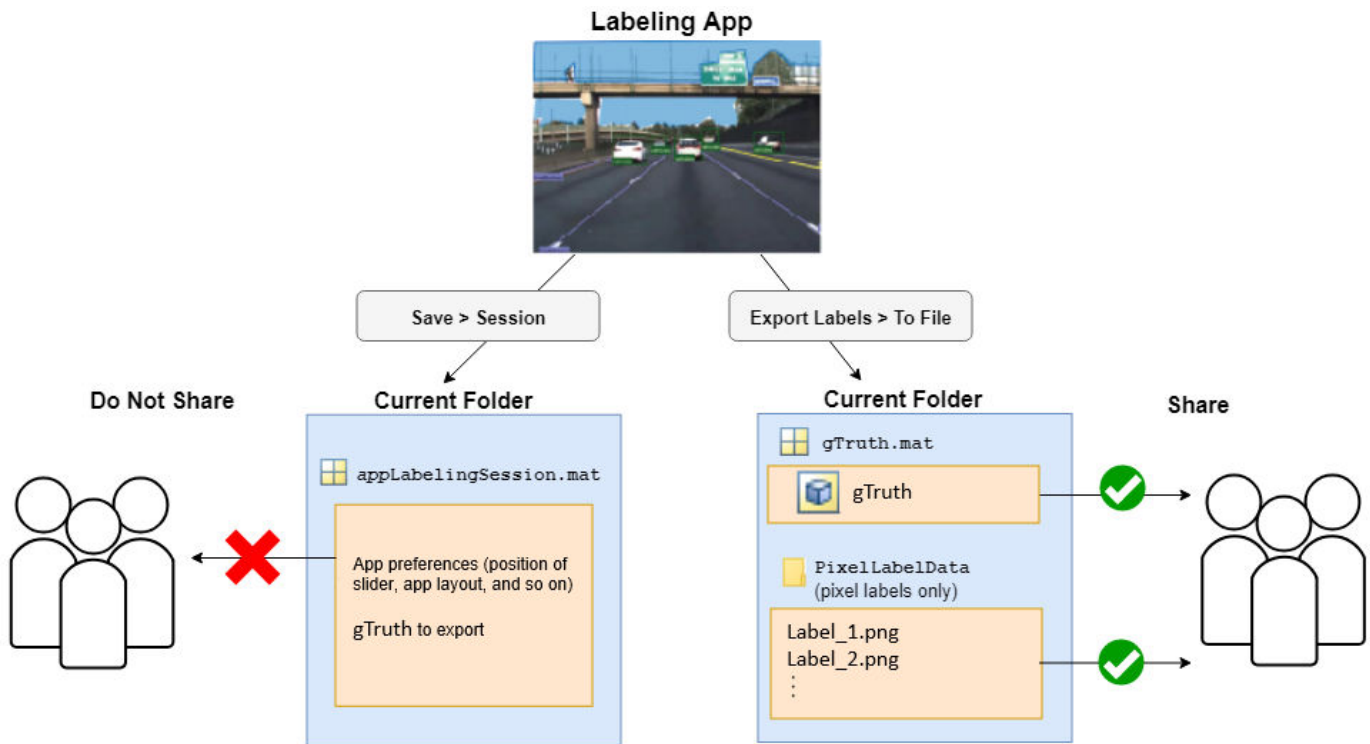
- Other labeling colleagues, who can use it to continue labeling
- Algorithm developers, who can use it to train algorithms, such as an object detector or semantic segmentation network
- Validation engineers, who can use it to validate algorithms

Share Ground Truth

To export and share labeled ground truth data from one of the labeling apps, select **Export Labels > To File**. Then, either share the exported MAT-file directly with individuals on your team or place it in a shared network location.

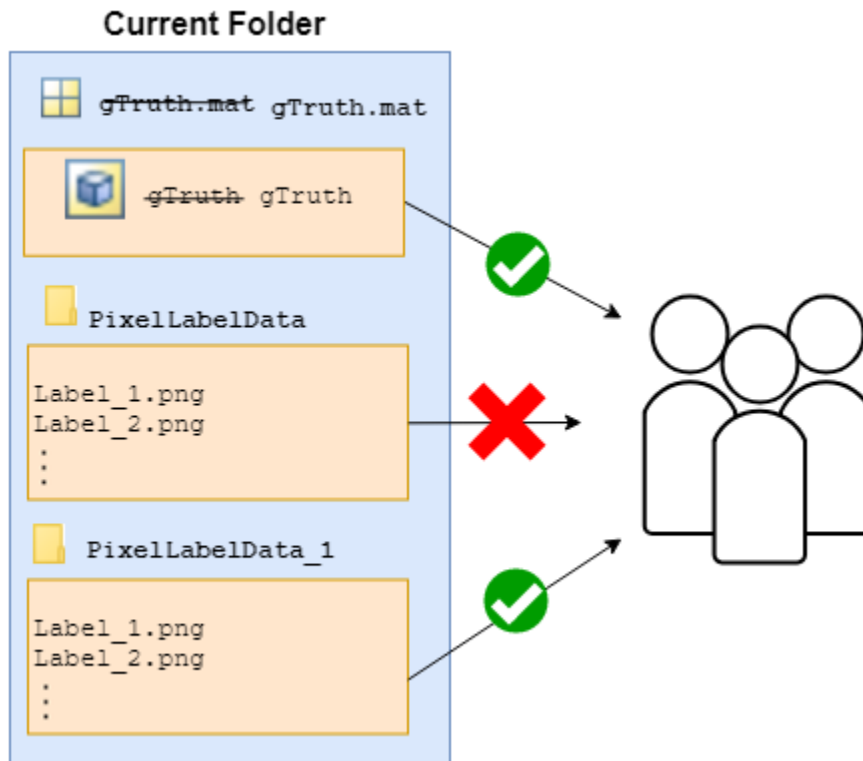
If the exported ground truth data contains pixel labels, the app also generates a `PixelLabelData` folder containing the pixel label data. The label data table stored in the ground truth object references the path to this folder. Share this folder along with the ground truth object.

The labeling apps also enable you to save a MAT-file of the entire app session. Do not share this file. Because the session file contains app preferences that are specific to your local machine, this file might not work on other machines.



If you re-export a ground truth object containing pixel label data, the app generates a new `PixelLabelData` folder. Even if you overwrite the original ground truth object, the app generates a new `PixelLabelData` folder. When re-exporting the ground truth object, the generated folders are named `PixelLabelData_1`, `PixelLabelData_2`, and so on, depending on how many times you re-export the object to the same folder.

When sharing a ground truth object, share the correct `PixelLabelData` folder associated with it. For example, if you overwrite the original ground truth object, share the overwritten object and the newly created `PixelLabelData_1` folder.



In addition to sharing the ground truth object, you must also share the data source (or data sources) and any associated files. These tables show the files to share for each data source in each app.

Image Labeler App Files to Share

Data Source	Files to Share
Image collection	<ul style="list-style-type: none"> groundTruth object MAT-file PixelLabelData folder (pixel labels only) Folders containing image collections (if not in a shared location)

Video Labeler App Files to Share

Data Source	Files to Share
Video	<ul style="list-style-type: none"> groundTruth object MAT-file PixelLabelData folder (pixel labels only) Video source file (if not in a shared location)
Image sequence	<ul style="list-style-type: none"> groundTruth object MAT-file PixelLabelData folder (pixel labels only) Folder containing image sequence (if not in a shared location) Timestamps duration vector (if specified)

Data Source	Files to Share
Custom image data source reader	<ul style="list-style-type: none"> • groundTruth object MAT-file • PixelLabelData folder (pixel labels only) • Data source files (if not in a shared location) • Custom reader function

Ground Truth Labeler App Files to Share

Data Source	Files to Share
Video	<ul style="list-style-type: none"> • groundTruthMultisignal object MAT-file • PixelLabelData folder (pixel labels only) • Video source file (if not in a shared location)
Image sequence	<ul style="list-style-type: none"> • groundTruthMultisignal object MAT-file • PixelLabelData folder (pixel labels only) • Folder containing image sequence (if not in a shared location) • Timestamps duration vector (if specified)
Custom image data source reader	<ul style="list-style-type: none"> • groundTruthMultisignal object MAT-file • PixelLabelData folder (pixel labels only) • Data source files (if not in a shared location) • Custom reader function
Point cloud sequence	<ul style="list-style-type: none"> • groundTruthMultisignal object MAT-file • PixelLabelData folder (pixel labels only) • Folder containing point cloud sequence (if not in a shared location) • Timestamps duration vector (if specified)
Velodyne packet capture (PCAP) file	<ul style="list-style-type: none"> • groundTruthMultisignal object MAT-file • PixelLabelData folder (pixel labels only) • PCAP source file (if not in a shared location) • PCAP calibration file • Timestamps duration vector (if specified)
Rosbag	<ul style="list-style-type: none"> • groundTruthMultisignal object MAT-file • PixelLabelData folder (pixel labels only) • Rosbag file

Move Ground Truth

In the exported ground truth object, the `DataSource` property contains the absolute paths to the data source files. For example, suppose you want to view the paths for a `groundTruth` object, `gTruth`, that was exported from the **Image Labeler** app. At the MATLAB command prompt, enter this code.

```
gTruth.DataSource
```

```
ans =
```

```
groundTruthDataSource for an image collection with properties
```

```
Source: {
    '...\matlab\toolbox\vision\visiondata\imageSets\cups\bigMug.jpg';
    '...\matlab\toolbox\vision\visiondata\imageSets\cups\blueCup.jpg';
    '...\matlab\toolbox\vision\visiondata\imageSets\cups\handMade.jpg';
    ... and 9 more
}
```

If you move the ground truth object to a new location, you might need to change the file paths stored in the data source (or data sources). Even if the data source files are on a shared network, if other people map a different drive letter to their network folder, the file paths can be incorrect.

To update these paths, use the `changeFilePaths` function. Specify the ground truth object as an input argument to this function. If the paths changed but the files names did not, specify a string vector containing the old and new path. The function returns any paths that it is unable to resolve. For example, this code sample shows how to change the drive letter for an image folder.

```
alternativePaths = ["C:\Shared\ImgFolder" "D:\Shared\ImgFolder"];
unresolvedPaths = changeFilePaths(gTruth,alternativePaths);
```

If the file names also changed, specify a cell array of string vectors containing the old and new paths. For example, this code sample shows how to change the drive letter for individual files and how to append a suffix to each file.

```
alternativePaths = ...
    [{"C:\Shared\ImgFolder\Img1.png" "D:\Shared\ImgFolder\Img1_new.png"}, ...
     ["C:\Shared\ImgFolder\Img2.png" "D:\Shared\ImgFolder\Img2_new.png"}, ...
     .
     .
     .
     ["C:\Shared\ImgFolder\ImgN.png" "D:\Shared\ImgFolder\ImgN_new.png"]};
unresolvedPaths = changeFilePaths(gTruth,alternativePaths);
```

If the ground truth object contains pixel label data, you can also use the `changeFilePaths` function to update the path names to the pixel label data stored in the `PixelLabelData` folder.

Store Ground Truth

Store the ground truth object in a location that is on the MATLAB search path. For more details, see “What Is the MATLAB Search Path?”.

For data sources whose contents reside in a single folder, consider storing the ground truth object in the parent folder of the data source. For image collections containing images from different folders, no specific recommendations exist for where to store the object. To label image collections, use the **Image Labeler** app.

See Also

Apps

Ground Truth Labeler | **Image Labeler** | **Video Labeler**

Objects

groundTruth | groundTruthDataSource | groundTruthMultisignal

Functions

changeFilePaths (groundTruth) | changeFilePaths (groundTruthMultisignal)

More About

- “How Labeler Apps Store Exported Pixel Labels” on page 14-16

Keyboard Shortcuts and Mouse Actions for Image Labeler

Note On Macintosh platforms, use the **Command (⌘)** key instead of **Ctrl**.

Label Definitions

Task	Action
In the ROI Label Definition pane, navigate through ROI labels and their groups	Up arrow or down arrow
In the Scene Label Definition pane, navigate through scene labels and their groups	Hold Alt and press the up arrow or down arrow
Reorder labels within a group or move labels between groups	Click and drag labels
Reorder groups	Click and drag groups

Image Browsing and Selection

Browse and select images from the image browser, which is located in the bottom pane of the app.

Task	Action
Browse through images one at a time	Left arrow and right arrow
Browse to the next set of images that is viewable in the image browser	<ul style="list-style-type: none"> PC: Page Up and Page Down Mac: Hold Fn and press the up and down arrows
Go to the first image	<ul style="list-style-type: none"> PC: Home Mac: Hold Fn and press the left arrow
Go to the last image	<ul style="list-style-type: none"> PC: End Mac: Hold Fn and press the right arrow
Select all images from the current image to the first image	<ul style="list-style-type: none"> PC: Shift+Home Mac: Hold Fn+Shift and press the left arrow
Select all images from the current image to the last image	<ul style="list-style-type: none"> PC: Shift+End Mac: Hold Fn+Shift and press the right arrow
Select all images from the current image to a specific image	Hold Shift and click the final image in the range
Select a specific set of images	Hold Ctrl and click the images you want to select

Labeling Window

Perform labeling actions, such as adding, moving, and deleting regions of interest (ROIs). The ROIs can be pixel labels or non-pixel ROI labels that include line, rectangle, cuboid, and projected cuboid.

Task	Action
Undo labeling action	Ctrl+Z
Redo labeling action	Ctrl+Y
Select all non-pixel ROIs	Ctrl+A
Select specific non-pixel ROIs	Hold Ctrl and click the ROIs you want to select
Cut selected non-pixel ROIs	Ctrl+X
Copy selected non-pixel ROIs to clipboard	Ctrl+C
Paste copied non-pixel ROIs <ul style="list-style-type: none"> • If a sublabel was copied, both the sublabel and its parent label are pasted. • If a parent label was copied, only the parent label is pasted, not its sublabels. For more details, see “Use Sublabels and Attributes to Label Ground Truth Data” on page 14-96.	Ctrl+V
Switch between selected non-pixel ROI labels. You can switch between labels only of the same type. For example, if you select a rectangle ROI, you can switch only between other rectangle ROIs.	Tab or Shift+Tab
Move a drawn non-pixel ROI label	Hold Ctrl and press the up, down, left or right arrows
Resize a rectangle ROI uniformly across all dimensions	Ctrl+Plus (+) or Ctrl+Minus (-)
Delete selected non-pixel ROIs	Delete
Copy all pixel ROIs	Ctrl+Shift+C
Cut all pixel ROIs	Ctrl+Shift+X
Paste copied or cut pixel ROIs	Ctrl+Shift+V
Delete all pixel ROIs	Ctrl+Shift+Delete
Fill all or all remaining pixels	Shift+click

Polyline Drawing

Draw ROI line labels on a frame. ROI line labels are polylines, meaning that they are composed of one or more line segments.

Task	Action
Commit a polyline to the frame, excluding the currently active line segment	Press Enter or right-click while drawing the polyline

Task	Action
Commit a polyline to the frame, including the currently active line segment	Double-click while drawing the polyline A new line segment is committed at the point where you double-click.
Delete the previously created line segment in a polyline	Backspace
Cancel drawing and delete the entire polyline	Escape

Polygon Drawing



Draw polygons to label pixels on a frame.

Task	Action
Commit a polygon to the frame, excluding the currently active line segment	Press Enter or right-click while drawing the polygon The polygon closes up by forming a line between the previously committed point and the first point in the polygon.
Commit a polygon to the frame, including the currently active line segment	Double-click while drawing polygon The polygon closes up by forming a line between the point where you double-clicked and the first point in the polygon.
Remove the previously created line segment from a polygon	Backspace
Cancel drawing and delete the entire polygon	Escape

Zooming

Task	Action
Zoom in or out of frame	Move the scroll wheel up (zoom in) or down (zoom out) The scroll wheel works in Zoom In , Zoom Out , and Label mode but not Pan mode.
Zoom in on specific section of frame	From the app toolbar, under Modes , select Zoom In . Then, draw a box around the section of the frame you want to zoom in on.

Zooming and Panning

Task	Action
Zoom in or out of frame	<p>Move the scroll wheel up (zoom in) or down (zoom out)</p> <p>If the frame is in pan mode, then zooming is not supported. To enable zooming, in the upper-right corner of the frame, either click the Pan button  to disable panning or click one of the zoom buttons.</p>
Zoom in on specific section of frame	<p>In the upper-right corner of the frame, click the Zoom In button . Then, draw a box around the section of the frame that you want to zoom in on.</p>
Pan across frame	Press the up, down, left, or right arrows

App Sessions

Task	Action
Save current session	Ctrl+S

See Also

Image Labeler

More About

- “Get Started with the Image Labeler” on page 14-63

Keyboard Shortcuts and Mouse Actions for Video Labeler

Note On Macintosh platforms, use the **Command** (⌘) key instead of **Ctrl**.

Label Definitions

Task	Action
In the ROI Label Definition pane, navigate through ROI labels and their groups	Up arrow or down arrow
In the Scene Label Definition pane, navigate through scene labels and their groups	Hold Alt and press the up arrow or down arrow
Reorder labels within a group or move labels between groups	Click and drag labels
Reorder groups	Click and drag groups

Frame Navigation and Time Interval Settings

Navigate between frames and change the time interval of the signal. These controls are located in the bottom pane of the app.

Task	Action
Go to the next frame	Right arrow
Go to the previous frame	Left arrow
Go to the last frame	<ul style="list-style-type: none"> PC: End Mac: Hold Fn and press the right arrow
Go to the first frame	<ul style="list-style-type: none"> PC: Home Mac: Hold Fn and press the left arrow
Navigate through time interval boxes and frame navigation buttons	Tab
Commit time interval settings	Press Enter within the active time interval box (Start Time , Current , or End Time)

Labeling Window

Perform labeling actions, such as adding, moving, and deleting regions of interest (ROIs). The ROIs can be pixel labels or non-pixel ROI labels that include line, rectangle, cuboid, and projected cuboid.

Task	Action
Undo labeling action	Ctrl+Z
Redo labeling action	Ctrl+Y
Select all non-pixel ROIs	Ctrl+A
Select specific non-pixel ROIs	Hold Ctrl and click the ROIs you want to select

Task	Action
Cut selected non-pixel ROIs	Ctrl+X
Copy selected non-pixel ROIs to clipboard	Ctrl+C
Paste copied non-pixel ROIs <ul style="list-style-type: none"> • If a sublabel was copied, both the sublabel and its parent label are pasted. • If a parent label was copied, only the parent label is pasted, not its sublabels. For more details, see “Use Sublabels and Attributes to Label Ground Truth Data” on page 14-96.	Ctrl+V
Switch between selected non-pixel ROI labels. You can switch between labels only of the same type. For example, if you select a rectangle ROI, you can switch only between other rectangle ROIs.	Tab or Shift+Tab
Move a drawn non-pixel ROI label	Hold Ctrl and press the up, down, left or right arrows
Resize a rectangle ROI uniformly across all dimensions	Ctrl+Plus (+) or Ctrl+Minus (-)
Delete selected non-pixel ROIs	Delete
Copy all pixel ROIs	Ctrl+Shift+C
Cut all pixel ROIs	Ctrl+Shift+X
Paste copied or cut pixel ROIs	Ctrl+Shift+V
Delete all pixel ROIs	Ctrl+Shift+Delete
Fill all or all remaining pixels	Shift+click

Polyline Drawing

Draw ROI line labels on a frame. ROI line labels are polylines, meaning that they are composed of one or more line segments.



Task	Action
Commit a polyline to the frame, excluding the currently active line segment	Press Enter or right-click while drawing the polyline
Commit a polyline to the frame, including the currently active line segment	Double-click while drawing the polyline A new line segment is committed at the point where you double-click.
Delete the previously created line segment in a polyline	Backspace
Cancel drawing and delete the entire polyline	Escape

Polygon Drawing

Draw polygons to label pixels on a frame.

Task	Action
Commit a polygon to the frame, excluding the currently active line segment	Press Enter or right-click while drawing the polygon The polygon closes up by forming a line between the previously committed point and the first point in the polygon.
Commit a polygon to the frame, including the currently active line segment	Double-click while drawing polygon The polygon closes up by forming a line between the point where you double-clicked and the first point in the polygon.
Remove the previously created line segment from a polygon	Backspace
Cancel drawing and delete the entire polygon	Escape

Zooming and Panning

Task	Action
Zoom in or out of frame	Move the scroll wheel up (zoom in) or down (zoom out) If the frame is in pan mode, then zooming is not supported. To enable zooming, in the upper-right corner of the frame, either click the Pan button  to disable panning or click one of the zoom buttons.
Zoom in on specific section of frame	In the upper-right corner of the frame, click the Zoom In button  . Then, draw a box around the section of the frame that you want to zoom in on.
Pan across frame	Press the up, down, left, or right arrows

App Sessions

Task	Action
Save current session	Ctrl+S

See Also
Video Labeler

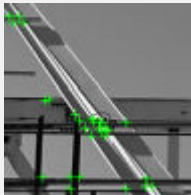

More About

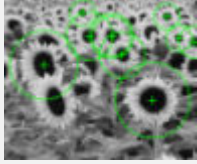

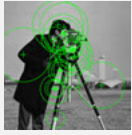

- “Get Started with the Video Labeler” on page 14-78

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 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]	 <p>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.</p>
	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]	
BRISKPoints	detectBRISKFeatures Binary Robust Invariant Scalable Keypoints (BRISK) algorithm [6]	 <p>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</p>

Points Object	Returned By	Type of Feature
SURFPoints	detectSURFFeatures Speeded-up robust features (SURF) algorithm [11]	 <p>Blobs Multiscale detection Object detection and image registration with scale and rotation changes</p>
ORBPoints	detectORBFeatures Oriented FAST and Rotated BRIEF (ORB) method [13]	 <p>Corners Multi-scale detection Point tracking, image registration, handles changes in rotation, corner detection in scenes of human origin, such as streets and indoor scenes</p>
KAZEPoints	detectKAZEFeatures KAZE is not an acronym, but a name derived from the Japanese word <i>kaze</i> , which means wind. The reference is to the flow of air ruled by nonlinear processes on a large scale. [12]	 <p>Multi-scale blob features Reduced blurring of object boundaries</p>
MSERRegions	detectMSERFeatures Maximally stable extremal regions (MSER) algorithm [7] [8] [9] [10]	 <p>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.</p>

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 <code>Orientation</code> property of the <code>validPoints</code> output object to the orientation of the extracted features, in radians.
	FREAK	The function sets the <code>Orientation</code> property of the <code>validPoints</code> output object to the orientation of the extracted features, in radians.
	SURF	<p>The function sets the <code>Orientation</code> property of the <code>validPoints</code> output object to the orientation of the extracted features, in radians.</p> <p>When you use an <code>MSERRegions</code> object with the SURF method, the <code>Centroid</code> property of the object extracts SURF descriptors. The <code>Axes</code> 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 $1/4 * \sqrt{(\text{majorAxes}/2) * (\text{minorAxes}/2)}$ and saturated to 1.6, as required by the <code>SURFPoints</code> object.</p>

Function	Description	
	KAZE	<p>Non-linear pyramid-based features.</p> <p>The function sets the <code>Orientation</code> property of the <code>validPoints</code> output object to the orientation of the extracted features, in radians.</p> <p>When you use an <code>MSERRegions</code> object with the KAZE method, the <code>Location</code> property of the object is used to extract KAZE descriptors.</p> <p>The <code>Axes</code> 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.</p>
	ORB	<p>The function does not set the <code>Orientation</code> property of the <code>validPoints</code> output object to the orientation of the extracted features. By default, the <code>Orientation</code> property of <code>validPoints</code> is set to the <code>Orientation</code> property of the input <code>ORBPoints</code> object.</p>
	Block	<p>Simple square neighborhood.</p> <p>The <code>Block</code> method extracts only the neighborhoods fully contained within the image boundary. Therefore, the output, <code>validPoints</code>, can contain fewer points than the input <code>POINTS</code>.</p>
	Auto	<p>The function selects the <code>Method</code> based on the class of the input points and implements:</p> <ul style="list-style-type: none"> The <code>FREAK</code> method for a <code>cornerPoints</code> input object. The <code>SURF</code> method for a <code>SURFPoints</code> or <code>MSERRegions</code> input object. The <code>FREAK</code> method for a <code>BRISKPoints</code> input object. The <code>ORB</code> method for a <code>ORBPoints</code> input object. <p>For an M-by-2 input matrix of $[x \ y]$ coordinates, the function implements the <code>Block</code> method.</p>
extractHOGFeatures	Extract histogram of oriented gradients (HOG) features	
insertMarker	Insert markers in image or video	
showMatchedFeatures	Display corresponding feature points	

Function	Description
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
- [13] Rublee, E., V. Rabaud, K. Konolige and G. Bradski. "ORB: An efficient alternative to SIFT or SURF." In *Proceedings of the 2011 International Conference on Computer Vision*, 2564-2571. Barcelona, Spain, 2011.

- [14] Rosten, E., and T. Drummond. "Fusing Points and Lines for High Performance Tracking," *Proceedings of the IEEE International Conference on Computer Vision*, Vol. 2 (October 2005): pp. 1508-1511.

See Also

More About

- Local Feature Detection and Extraction on page 14-126

See Also

Related Examples

- "Object Detection in a Cluttered Scene Using Point Feature Matching" on page 3-32

Local Feature Detection and Extraction

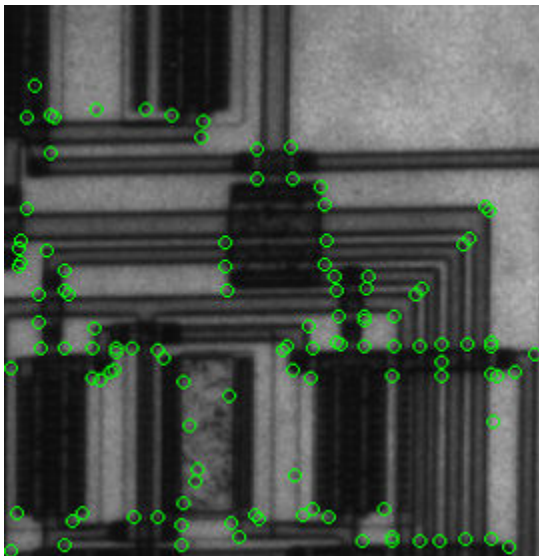
Local features and their descriptors, which are a compact vector representations of a local neighborhood, are the building blocks of many computer vision algorithms. Their applications include image registration, object detection and classification, tracking, and motion estimation. Using local features enables these algorithms to better handle scale changes, rotation, and occlusion. The Computer Vision Toolbox provides the FAST, Harris, ORB, 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, ORB, 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 14.1. 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 14-167.

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” on page 4-27
Object detection	“Object Detection in a Cluttered Scene Using Point Feature Matching” on page 3-32
Object recognition	“Digit Classification Using HOG Features” on page 4-14
Object tracking	“Face Detection and Tracking Using the KLT Algorithm” on page 7-20
Image category recognition	“Image Category Classification Using Bag of Features” on page 3-93
Finding geometry of a stereo system	“Uncalibrated Stereo Image Rectification” on page 1-78
3-D reconstruction	“Structure From Motion From Two Views” on page 1-37, “Structure From Motion From Multiple Views” on page 1-70
Image retrieval	“Image Retrieval Using Customized Bag of Features” on page 3-109

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, ORB 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 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: <ul style="list-style-type: none"> • Matching key points • Classification 	The HOG, SURF, and KAZE descriptors are suitable for classification tasks. In contrast, binary descriptors, such as ORB, 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 SURF, ORB or KAZE feature detector and descriptor. The SURF and the KAZE methods are computationally intensive.
Performance requirements: <ul style="list-style-type: none"> • 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	<code>detectFASTFeatures</code>	No
Minimum eigenvalue algorithm [4]	Corner	<code>detectMinEigenFeatures</code>	No
Corner detector [3]	Corner	<code>detectHarrisFeatures</code>	No
SURF [11]	Blob	<code>detectSURFFeatures</code>	Yes
KAZE [12]	Blob	<code>detectKAZEFeatures</code>	Yes
BRISK [6]	Corner	<code>detectBRISKFeatures</code>	Yes
MSER [8]	Region with uniform intensity	<code>detectMSERFeatures</code>	Yes
ORB [13]	Corner	<code>detectORBFeatures</code>	No

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 Correspondence	Classification
HOG	No	<code>extractHOGFeatures(I, ...)</code>	No	No	No	Yes
LBP	No	<code>extractLBPFeatures(I, ...)</code>	No	Yes	No	Yes
SURF	No	<code>extractFeatures(I,points,'Method','SURF')</code>	Yes	Yes	Yes	Yes
KAZE	No	<code>extractFeatures(I,points,'Method','KAZE')</code>	Yes	Yes	Yes	Yes
FREAK	Yes	<code>extractFeatures(I,points,'Method','FREAK')</code>	Yes	Yes	Yes	No
BRISK	Yes	<code>extractFeatures(I,points,'Method','BRISK')</code>	Yes	Yes	Yes	No
ORB	Yes	<code>extractFeatures(I,points,'Method','ORB')</code>	No	Yes	Yes	No
<ul style="list-style-type: none"> Block Simple pixel neighborhood around a keypoint 	No	<code>extractFeatures(I,points,'Method','Block')</code>	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 `estimateGeometricTransform2D` 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, inlierIdx] = estimateGeometricTransform2D( ...
    matchedDistorted, matchedOriginal, 'similarity');
inlierDistorted = matchedDistorted(inlierIdx, :);
inlierOriginal  = matchedOriginal(inlierIdx, :);
```

Display the matching points

```
figure
showMatchedFeatures(original, distorted, inlierOriginal, inlierDistorted)
```

```
title('Matching points (inliers only)')
legend('ptsOriginal','ptsDistorted')
```



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);
```

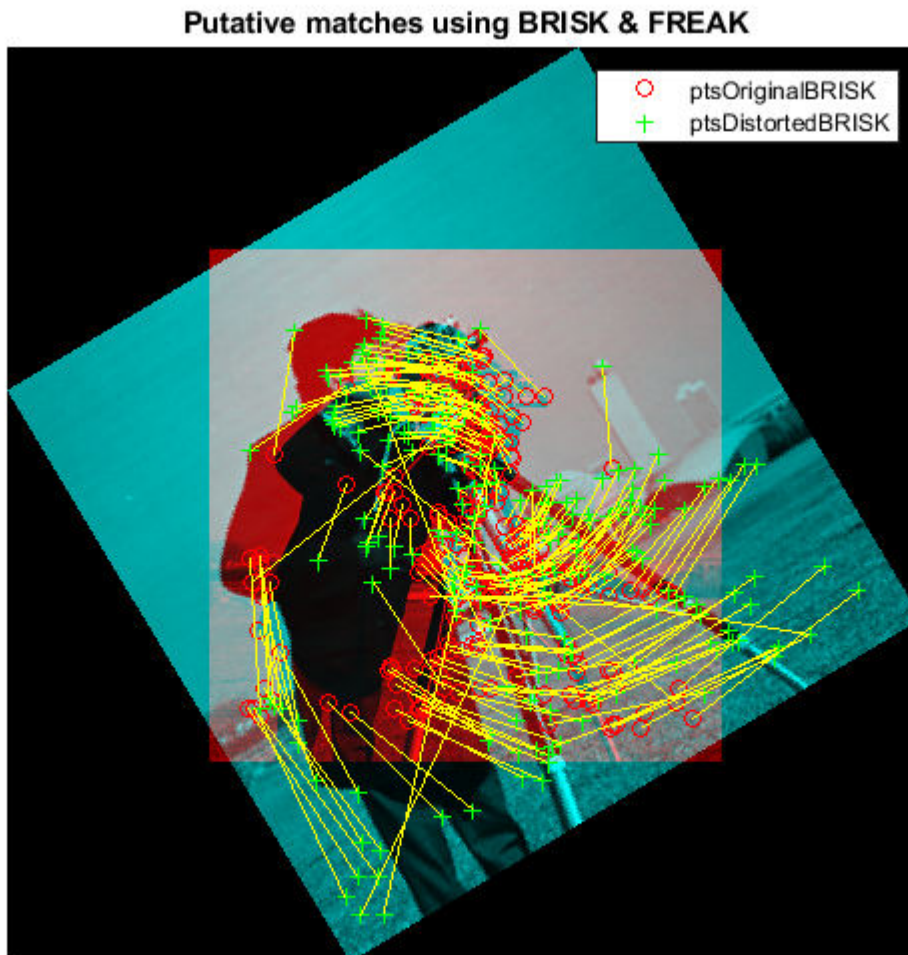
Obtain candidate matched points for BRISK and SURE.

```
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')
```



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.

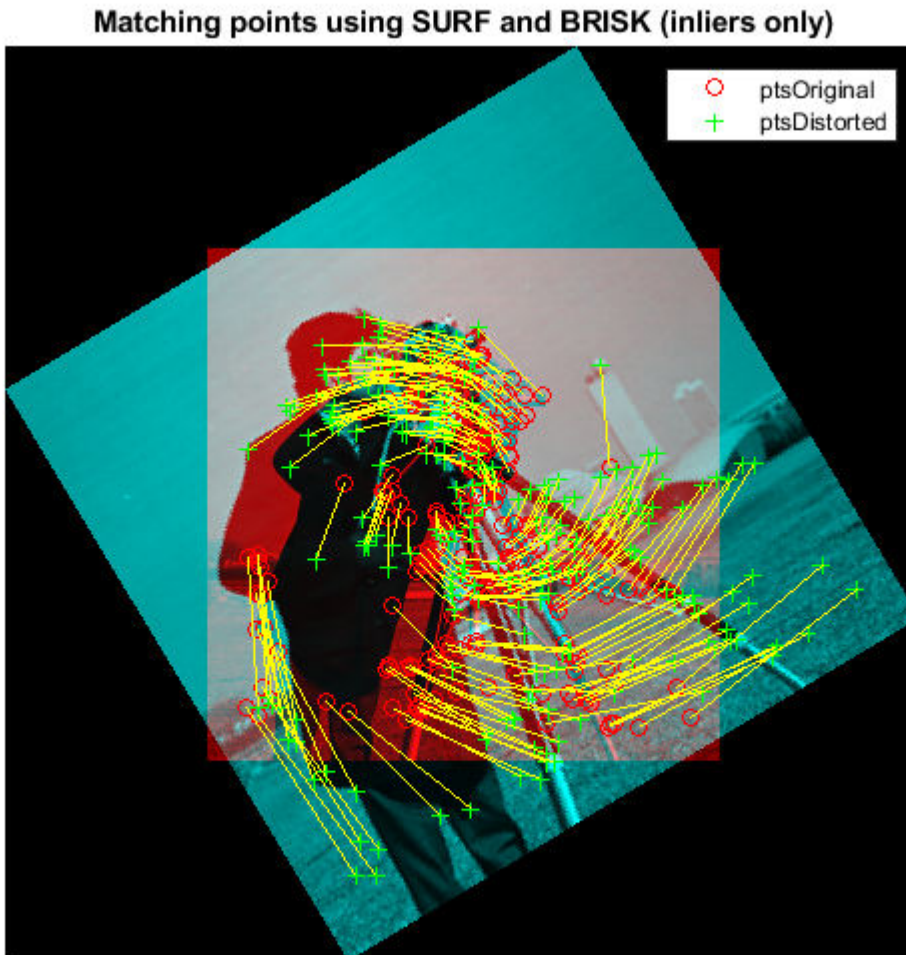
```
[tformTotal,inlierIdx] = ...
    estimateGeometricTransform2D(matchedDistortedXY,...
        matchedOriginalXY,'similarity');
inlierDistortedXY = matchedDistortedXY(inlierIdx, :);
inlierOriginalXY = matchedOriginalXY(inlierIdx, :);
```

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')

```



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
- [13] Rublee, E., V. Rabaud, K. Konolige and G. Bradski. "ORB: An efficient alternative to SIFT or SURF." In *Proceedings of the 2011 International Conference on Computer Vision*, 2564-2571. Barcelona, Spain, 2011.

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" on page 4-2
- "Object Detection in a Cluttered Scene Using Point Feature Matching" on page 3-32

Train a Cascade Object Detector

In this section...

“Why Train a Detector?” on page 14-143

“What Kinds of Objects Can You Detect?” on page 14-143

“How Does the Cascade Classifier Work?” on page 14-143

“Create a Cascade Classifier Using the `trainCascadeObjectDetector`” on page 14-144

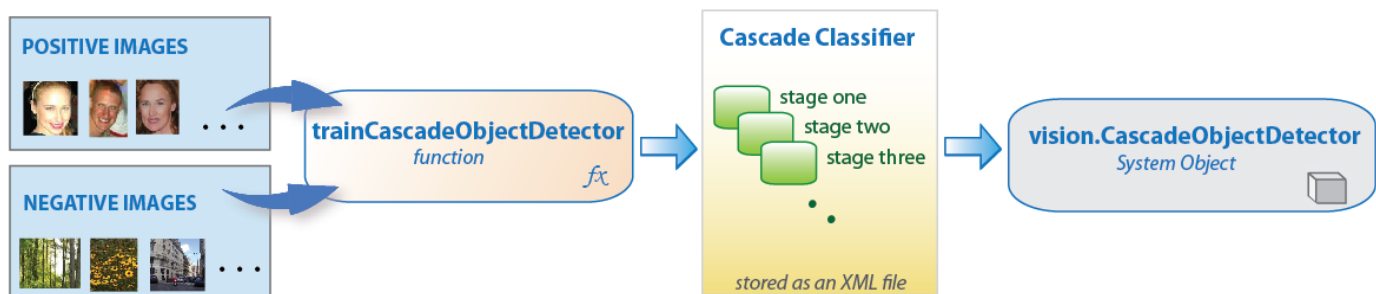
“Troubleshooting” on page 14-147

“Examples” on page 14-149

“Train Stop Sign Detector” on page 14-153

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 Toolbox provides the `trainCascadeObjectDetector` function to train a custom classifier.



What Kinds of Objects Can You Detect?

The Computer Vision 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.

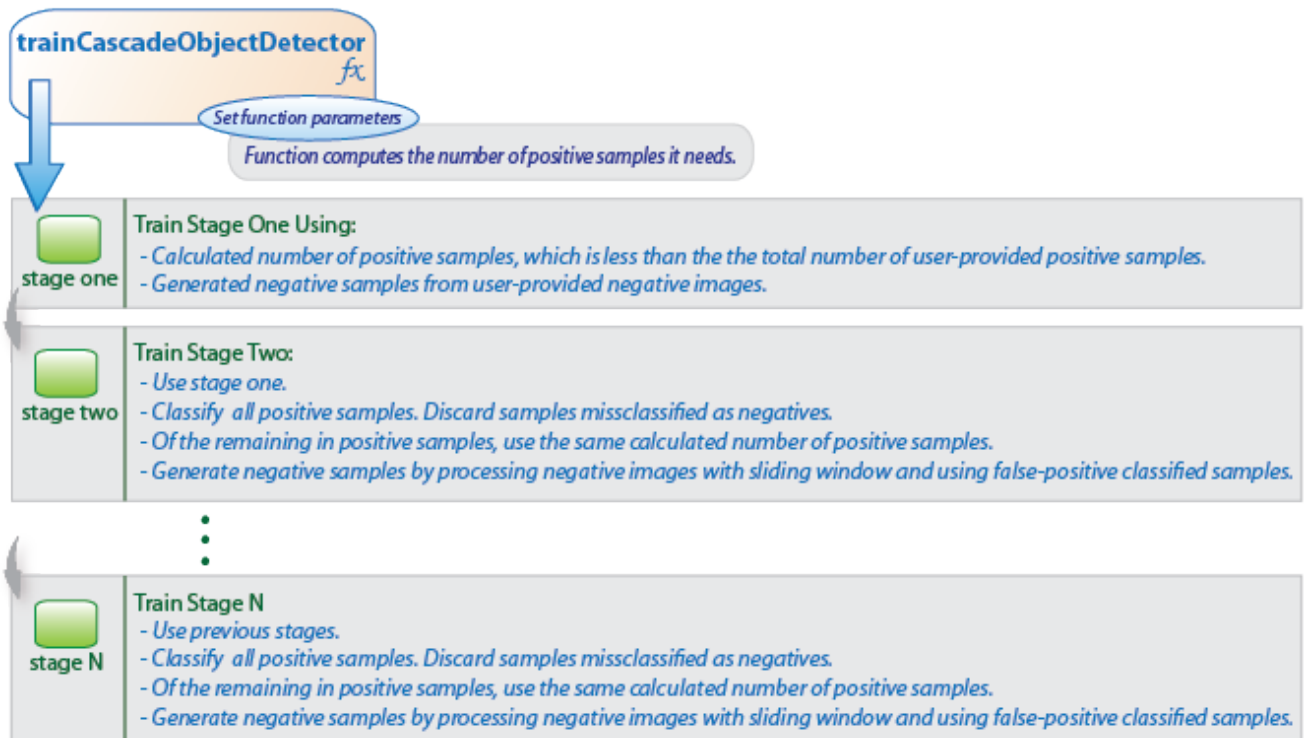
- A true positive occurs when a positive sample is correctly classified.
- A false positive occurs when a negative sample is mistakenly classified as positive.
- A false negative occurs when a positive sample is mistakenly classified as negative.

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.



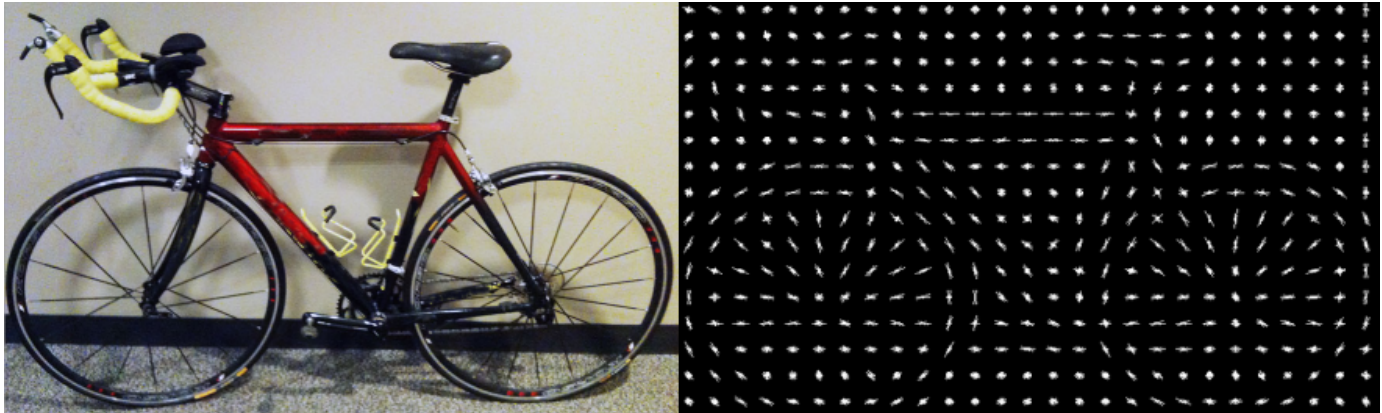
Considerations when Setting Parameters

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

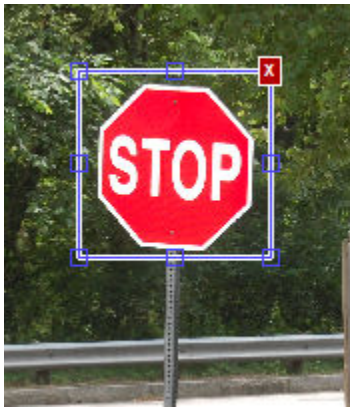
Choose the feature that suits the type of object detection you need. The `trainCascadeObjectDetector` supports three types of features: Haar, local binary patterns (LBP), and histograms of oriented gradients (HOG). Haar and LBP features are often used to detect faces because they work well for representing 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

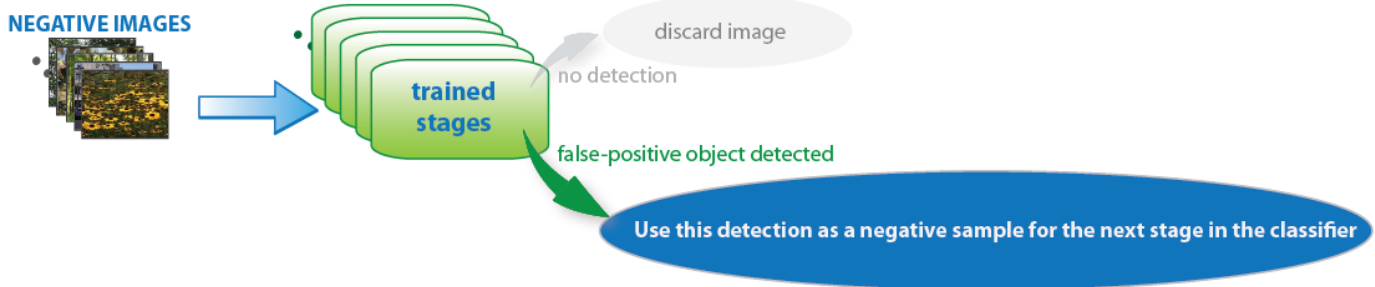
To create positive samples easily, you can use the **Image Labeler** app. The Image Labeler provides an easy way to label positive samples by interactively specifying rectangular regions of interest (ROIs).



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:

$$\text{number of positive samples} = \text{floor}(\text{totalPositiveSamples} / (1 + (\text{NumCascadeStages} - 1) * (1 - \text{TruePositiveRate})))$$

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 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  
[.....] I  
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: 1 seconds
```

```
Training stage 3 of 5
```

```
[.....]
```

```
Used 42 positive and 84 negative samples
```

```
Time to train stage 3: 4 seconds
```

```
Training stage 4 of 5
```

```
[.....]
```

```
Used 42 positive and 84 negative samples
```

```
Time to train stage 4: 10 seconds
```

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

Training complete

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

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

Read the test image.

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

Detect a stop sign.

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

Insert bounding 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);
```

See Also

More About

- “Get Started with the Image Labeler” on page 14-63

External Websites

- [Cascade Trainer](#)

Train Optical Character Recognition for Custom Fonts

In this section...

“Open the OCR Trainer App” on page 14-156


“Train OCR” on page 14-156

“App Controls” on page 14-158

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

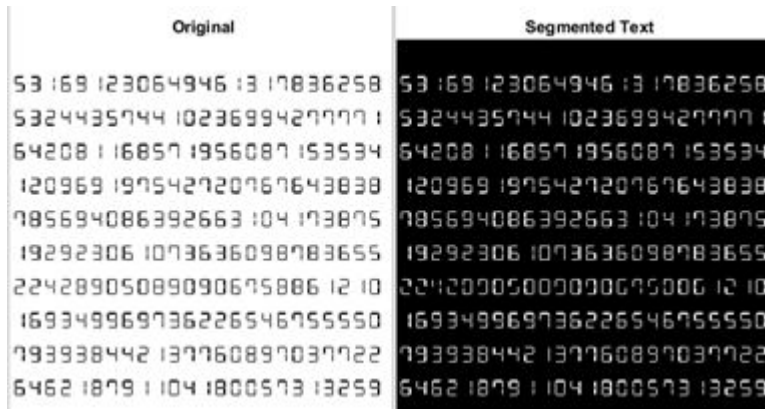
- MATLAB Toolstrip: On the **Apps** tab, under **Image Processing and Computer Vision**, click , the OCR app icon.
- MATLAB command prompt: Enter `ocrTrainer`.

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.”

- 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.



To limit the OCR to a specific character set, select the **Character set** check box and add the characters.

Note Use training images that contain text that you want OCR to recognize. Do not use training images with only a few characters. OCR training works best if training images contain blocks of many words. You can use the `insertText` function to automatically generate training images for a known font.

```
I = zeros(500,500,3,'uint8');

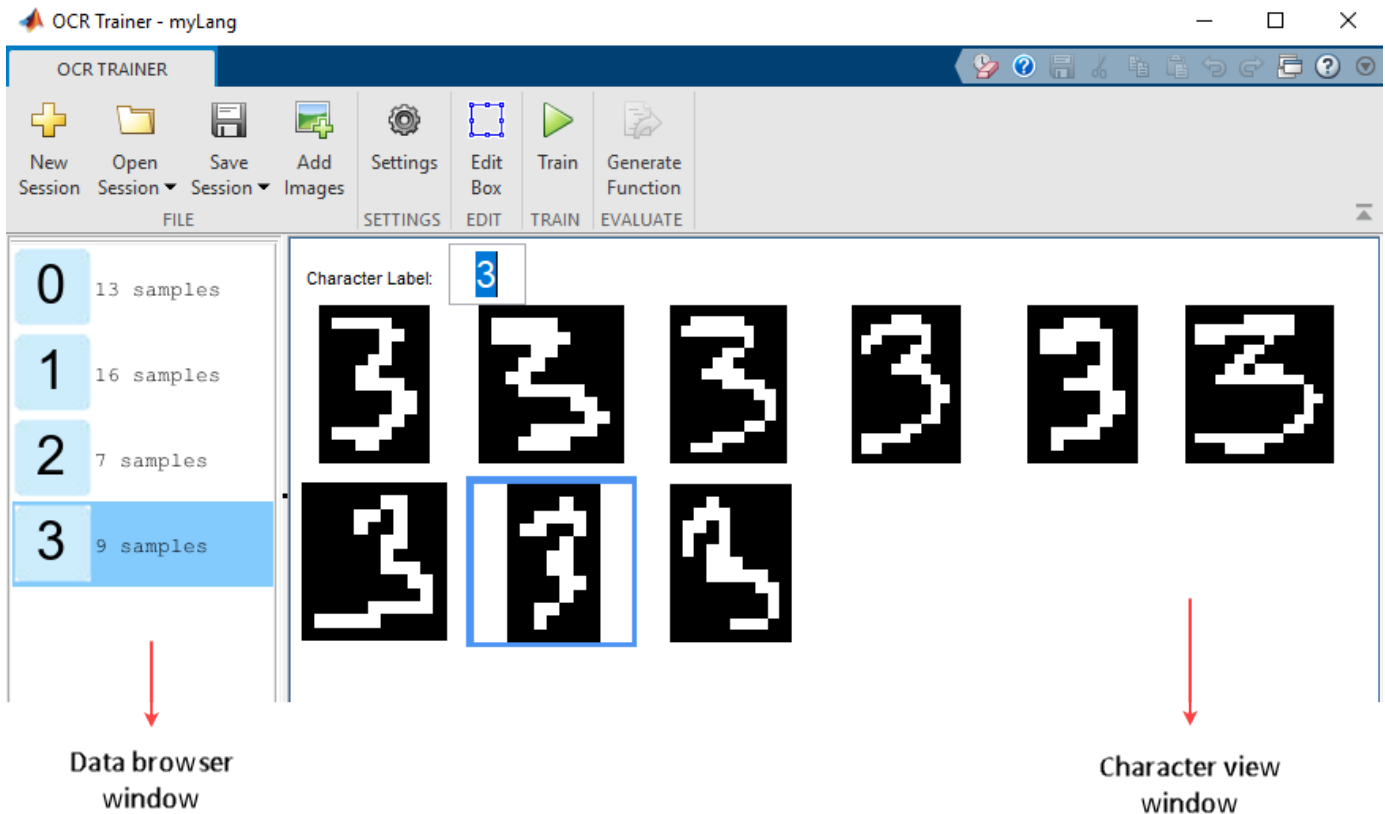
textLines = [
    "some training text"
    "even more stuff to learn"easy
]
lineYLocation = 50;

for i = 1:numel(textLines)
    I = insertText(I,[50 lineYLocation],char(textLines(i)), ...
        'Font','LucidaSansRegular',...
        'FontSize',16,'TextColor','white',...
        'BoxOpacity',0);

    % increment to next line
    lineYLocation = lineYLocation + 20;
end
figure
imshow(I)
```

- 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 in the character view window 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 window 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.

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.

Note Before running the OCR Trainer app, check if your machine has only one Tesseract installation. If there are multiple Tesseract installations, remove the extra installations and restart MATLAB to run the OCR Trainer app. Otherwise, the app returns the error "Not enough input arguments" when you click the Train button.

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 Toolbox Add-on Support Files” on page 10-2

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
<code>img</code>	Input	<ul style="list-style-type: none"> • Binary, grayscale, or truecolor image. • The input image is from the image set that was originally passed into <code>bagOfFeatures</code>.

Argument	Input/Output	Description
features	Output	<ul style="list-style-type: none"> 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 <code>bagOfFeatures</code> 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	<ul style="list-style-type: none"> An M-by-1 vector of feature metrics indicating the strength of each feature vector. Used to apply the 'SelectStrongest' criteria in <code>bagOfFeatures</code> framework. Numeric, real, and nonsparse.
location	Output	<ul style="list-style-type: none"> 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;
```

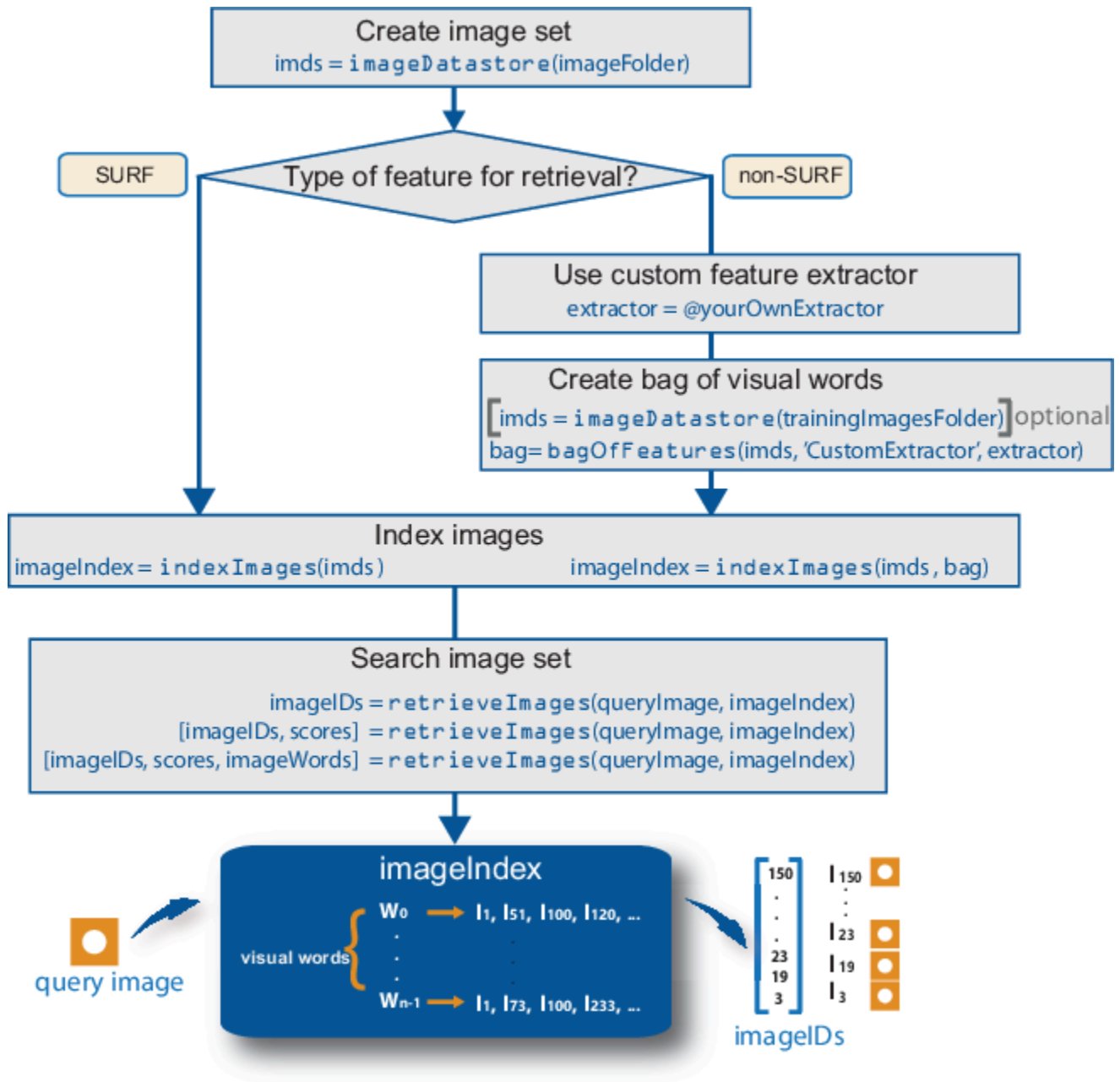
You can optionally return the feature location information. The feature location can be used for spatial or geometric verification image search applications. See the “Geometric Verification Using `estimateGeometricTransform2D` Function” example. The `retrieveImages` and `indexImages` functions are used for content-based image retrieval systems.

```
if nargin > 2
    varargout{1} = multiscaleGridPoints.Location;
end
```

Image Retrieval with Bag of Visual Words

You can use the Computer Vision 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.



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” on page 3-109

Image Classification with Bag of Visual Words

Use the Computer Vision 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

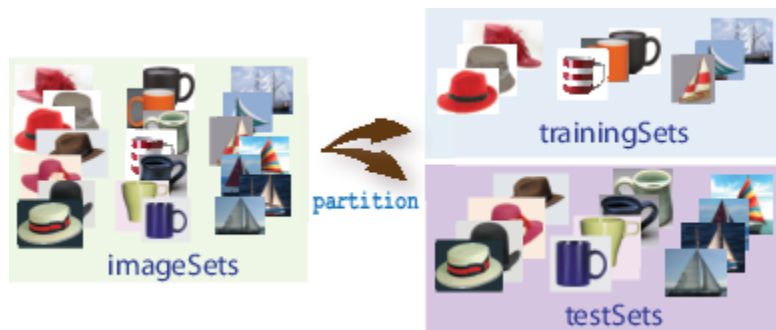
Organize and partition the images into training and test subsets. Use the `imageDatastore` function to store images to use for training an image classifier. Organizing images into categories makes handling large sets of images much easier. You can use the `splitEachLabel` function to split the images 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');
```

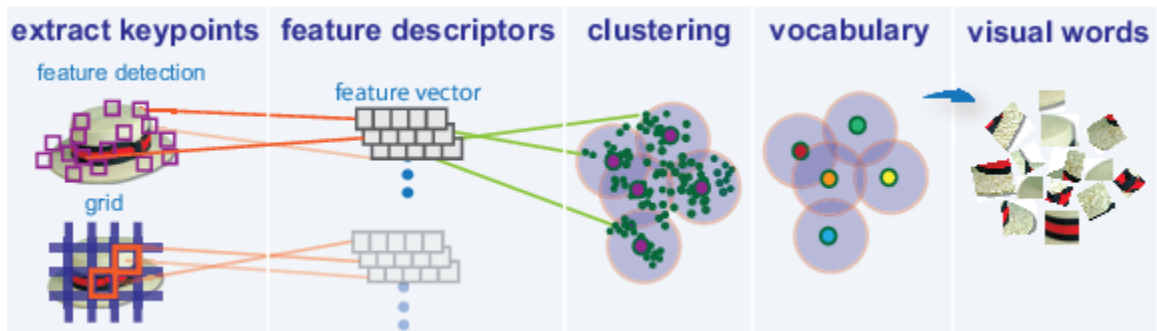


Step 2: Create Bag of Features

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.

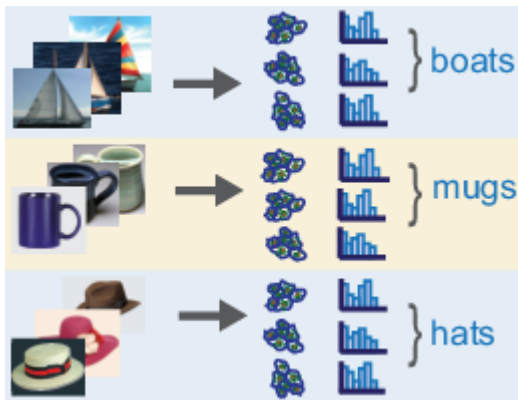
Step 3: Train an Image Classifier With Bag of Visual Words

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 `trainImageCategoryClassifier` 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.

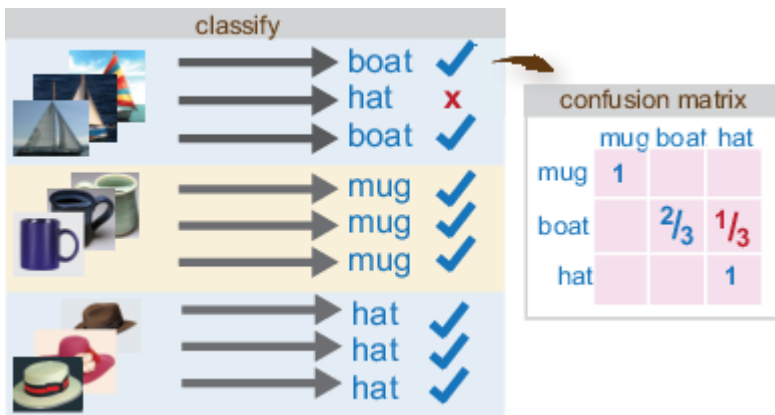
- 1 Use the `bagOfFeatures` `encode` method to encode each image from the training set. This function detects and extracts features from the image and then uses the approximate nearest neighbor algorithm to construct a feature histogram for each image. The function then increments histogram bins based 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.



- 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” on page 3-93
- “Image Retrieval Using Customized Bag of Features” on page 3-109

Motion Estimation and Tracking

- “Multiple Object Tracking” on page 15-2
- “Video Mosaicking” on page 15-5
- “Pattern Matching” on page 15-10
- “Pattern Matching” on page 15-15

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	<code>vision.ForegroundDetector</code> System object™
Faces, eyes, nose, mouth, upper body	Stationary, Moving	<code>vision.CascadeObjectDetector</code> System object
Pedestrians	Stationary, Moving	<code>vision.PeopleDetector</code> System object
Custom object category	Stationary, Moving	<code>trainCascadeObjectDetector</code> function or custom sliding window detector using <code>extractHOGFeatures</code> and <code>selectStrongestBbox</code>

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” on page 7-40 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` function of the `vision.KalmanFilter` object. You can also implement a custom cost function that 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” on page 7-40
- “Use Kalman Filter for Object Tracking” on page 7-50
- “Motion-Based Multiple Object Tracking” on page 7-31

More About

- “Train a Cascade Object Detector” on page 14-143

External Websites

- Detect and Track Multiple Faces

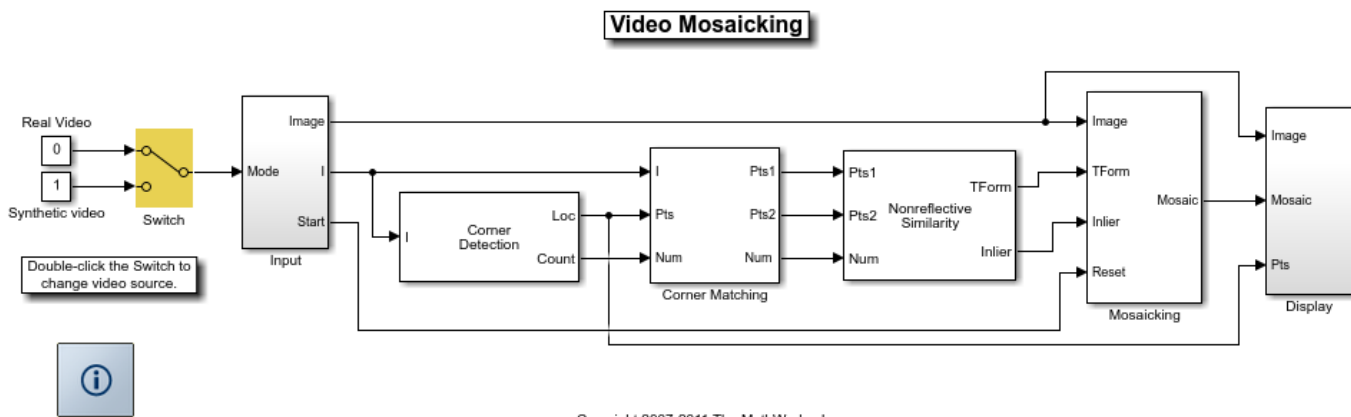
Video Mosaicking

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

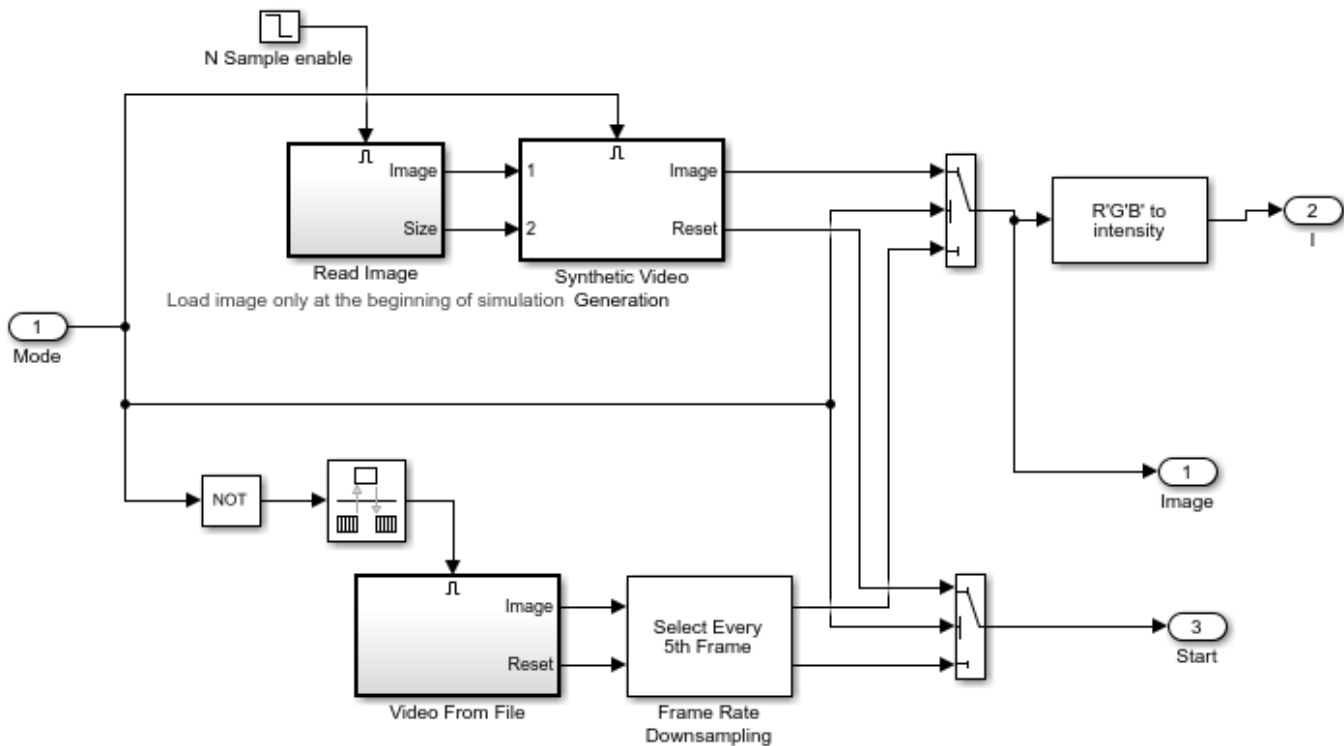
The following figure shows the Video Mosaicking model:



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.

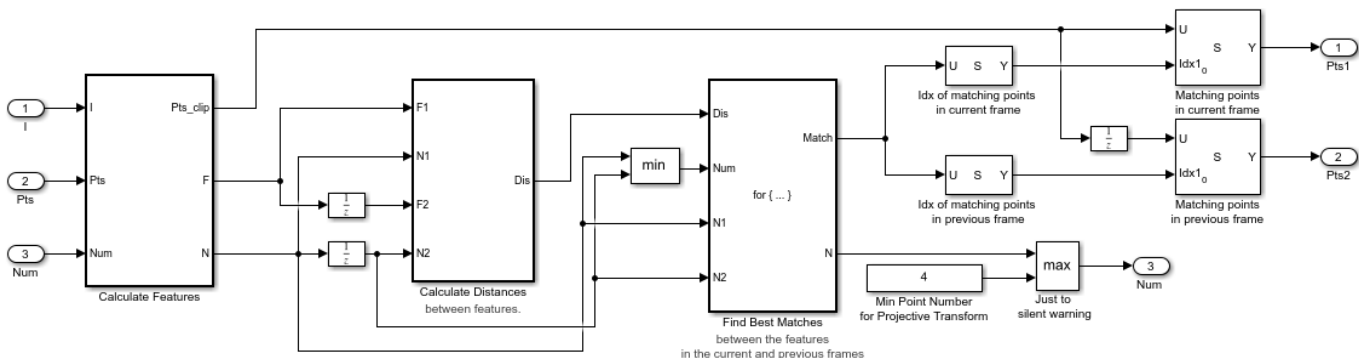


If you choose to use a video sequence from a file, you can reduce computation time by processing only some of the video frames. This is done by setting the downsampling rate in the Frame Rate Downsampling subsystem.

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

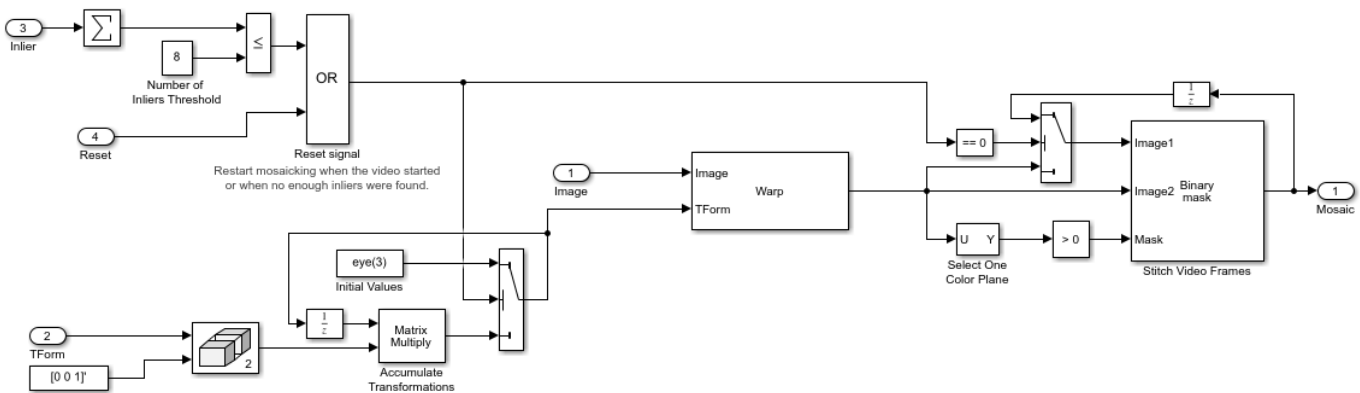
The subsystem finds corner features in the current video frame in one of three methods. The example uses Local intensity comparison (Rosen & Drummond), which is the fastest method. The other methods available are the Harris corner detection (Harris & Stephens) and the Minimum Eigenvalue (Shi & Tomasi).



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.

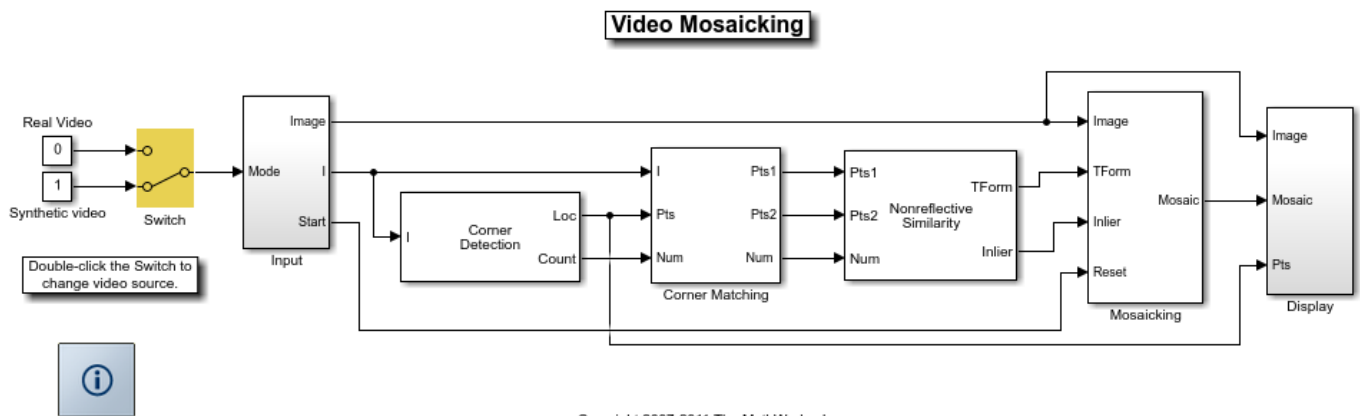
Mosaicking Subsystem

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



Copyright 2007-2011 The MathWorks, Inc.

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 = VideoReader('vipboard.mp4');
```

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);
```

```
% 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.

```
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.

```
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 VideoReader object.

```
while hasFrame(hVideoSrc)
    Im = rgb2gray(im2single(readFrame(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.
videopatternplots('update', hPlot, norm_Corr_value);
step(hROIPattern, Imf);
end

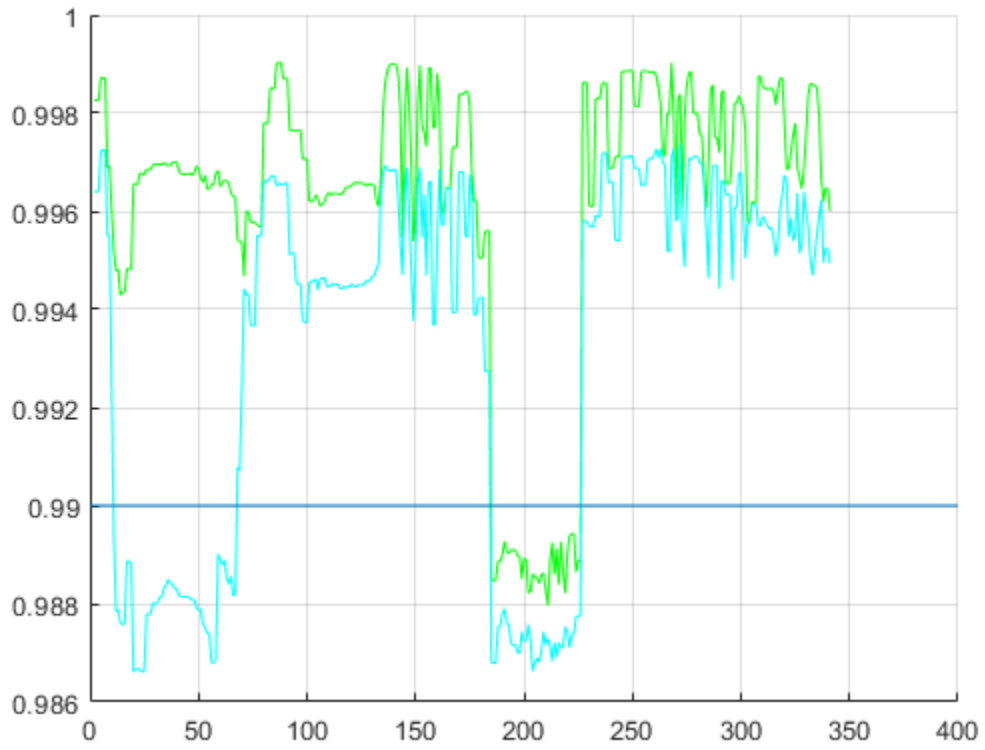
snapnow

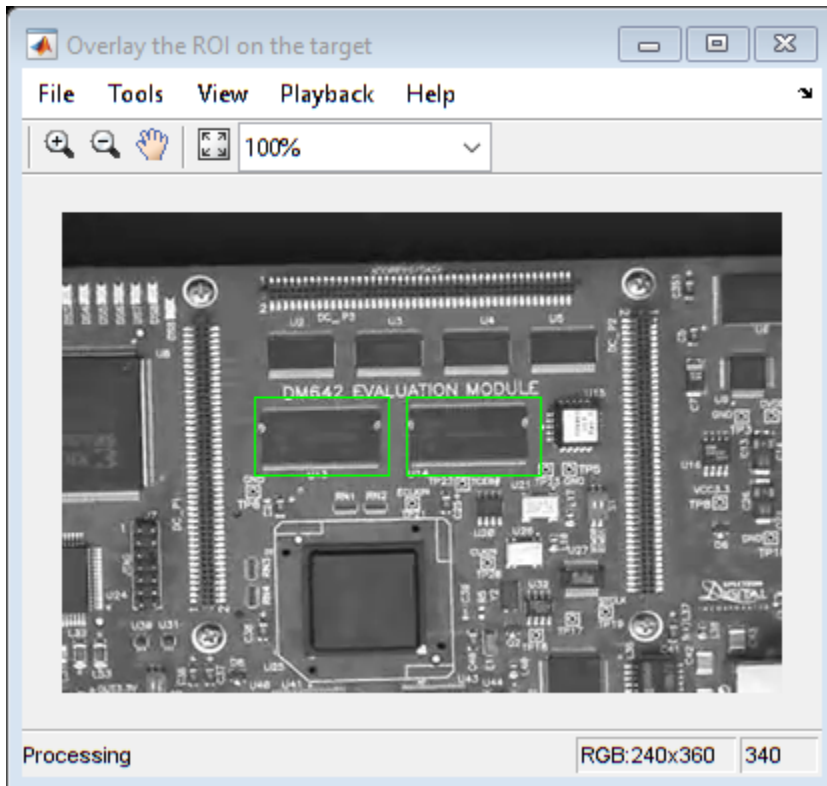
% Function to compute pyramid image at a particular level.
function outI = multilevelPyramid(inI, level)

I = inI;
outI = I;

for i=1:level
    outI = impyramid(I, 'reduce');
    I = outI;
end

end
```





Summary

This example shows use of Computer Vision 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

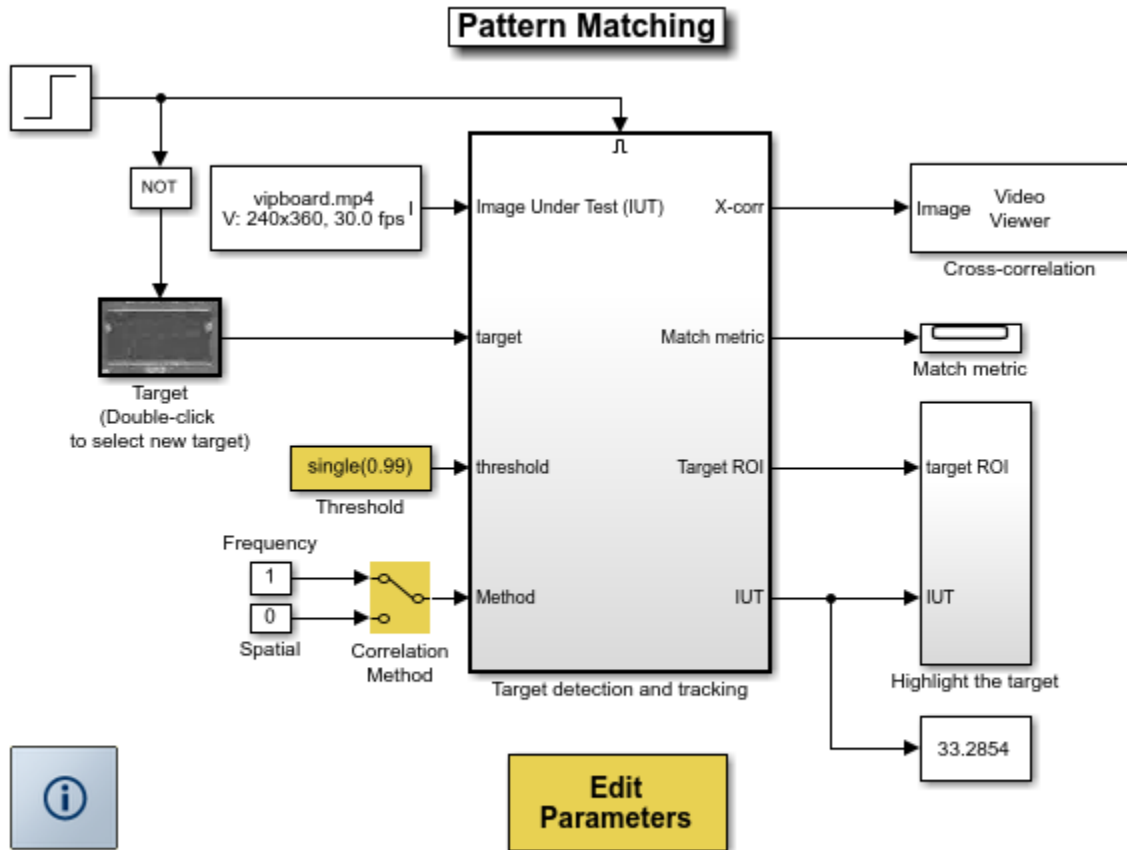
This example shows how to use the 2-D normalized cross-correlation for pattern matching and target tracking.

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

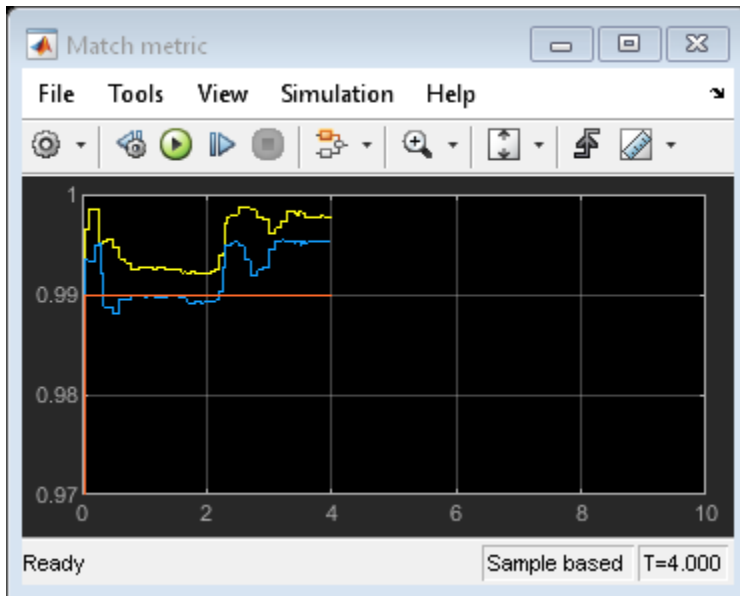
The following figure shows the Pattern Matching model:



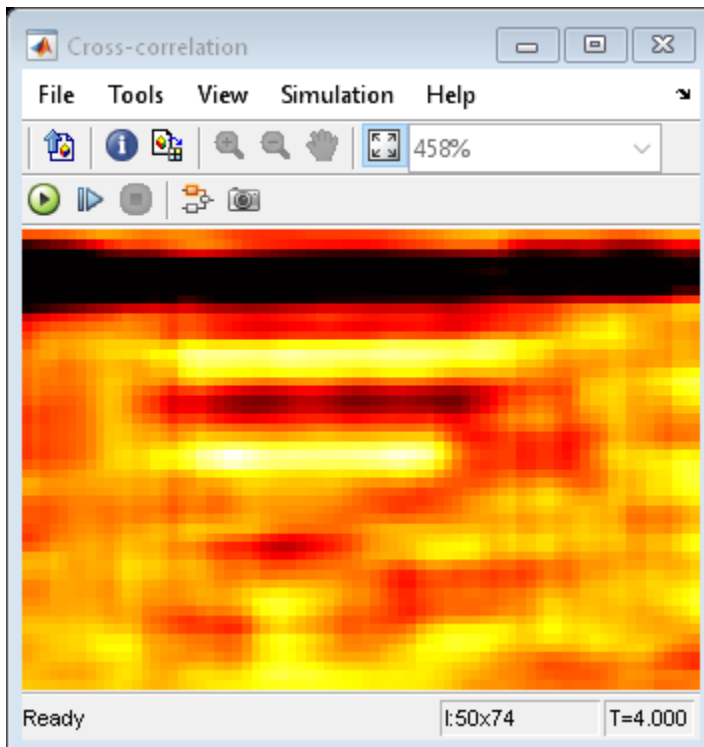
Copyright 2003-2008 The MathWorks, Inc.

Pattern Matching Results

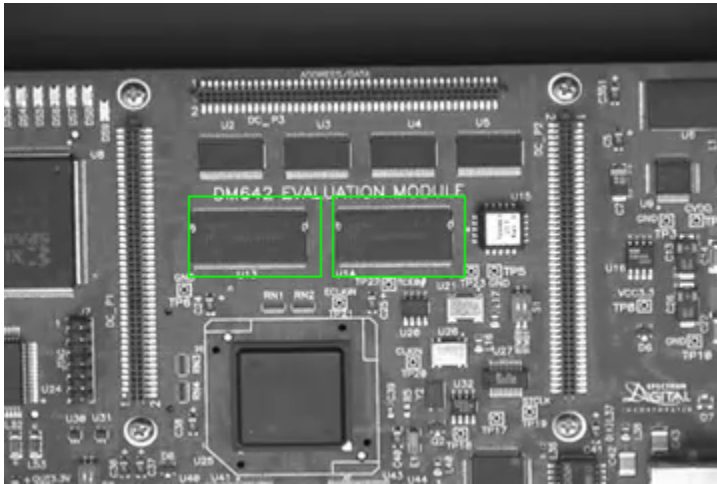
The Match metric window shows the variation of the target match metrics. The model determines that the target template is present in a video frame when the match metric exceeds a threshold (cyan line).



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.



Geometric Transformations

Nearest Neighbor, Bilinear, and Bicubic Interpolation Methods

In this section...

“Nearest Neighbor Interpolation” on page 16-2

“Bilinear Interpolation” on page 16-3

“Bicubic Interpolation” on page 16-3

Nearest Neighbor Interpolation

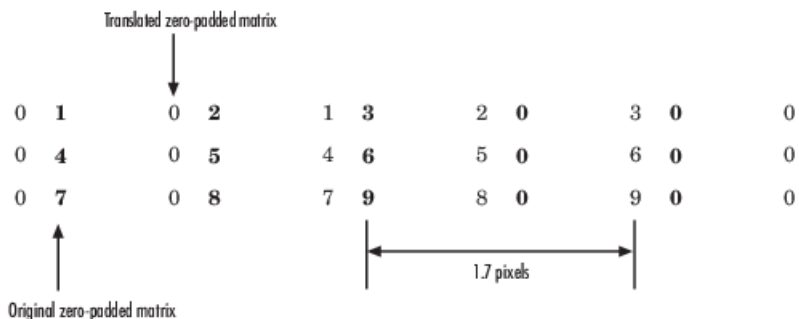
For nearest neighbor interpolation, the block uses the value of nearby translated pixel values for the output pixel values.

For example, suppose this matrix,

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

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

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



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

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

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

Bilinear Interpolation

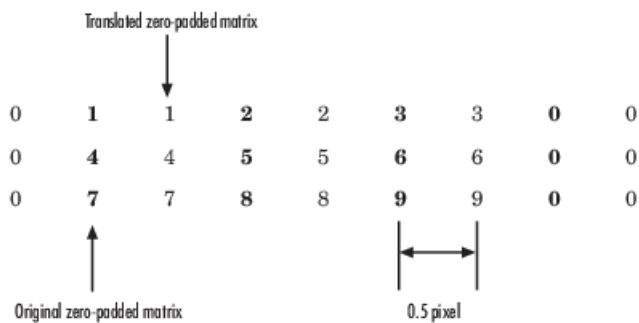
For bilinear interpolation, the block uses the weighted average of two translated pixel values for each output pixel value.

For example, suppose this matrix,

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

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

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



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

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

Bicubic Interpolation

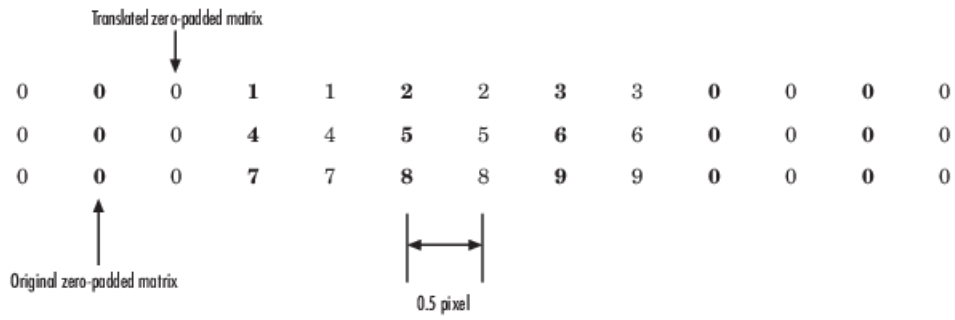
For bicubic interpolation, the block uses the weighted average of four translated pixel values for each output pixel value.

For example, suppose this matrix,

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

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

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



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

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

Filters, Transforms, and Enhancements

- “Adjust the Contrast of Intensity Images” on page 17-2
- “Adjust the Contrast of Color Images” on page 17-6
- “Remove Salt and Pepper Noise from Images” on page 17-10
- “Sharpen an Image” on page 17-14

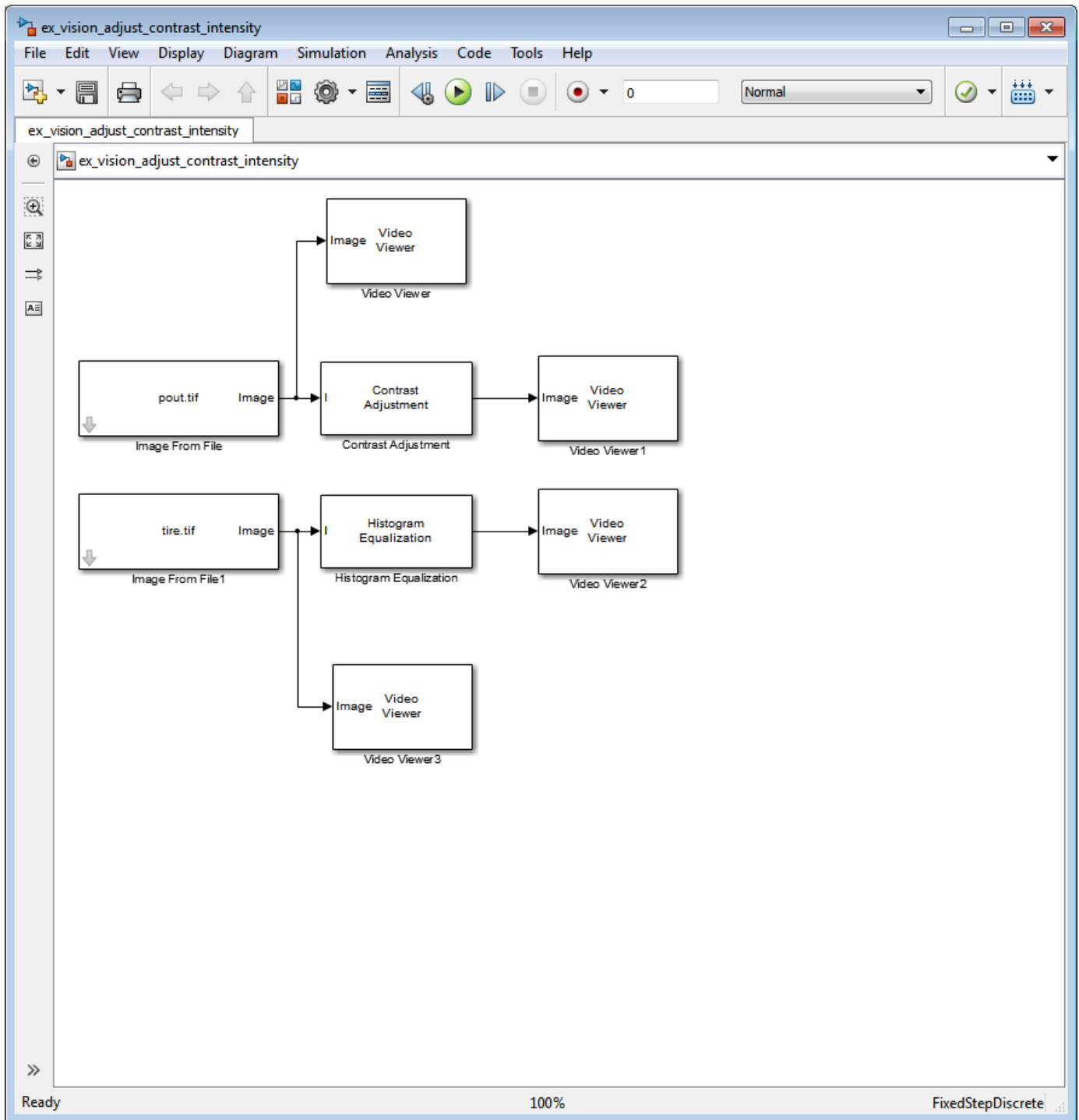
Adjust the Contrast of Intensity Images

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

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

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

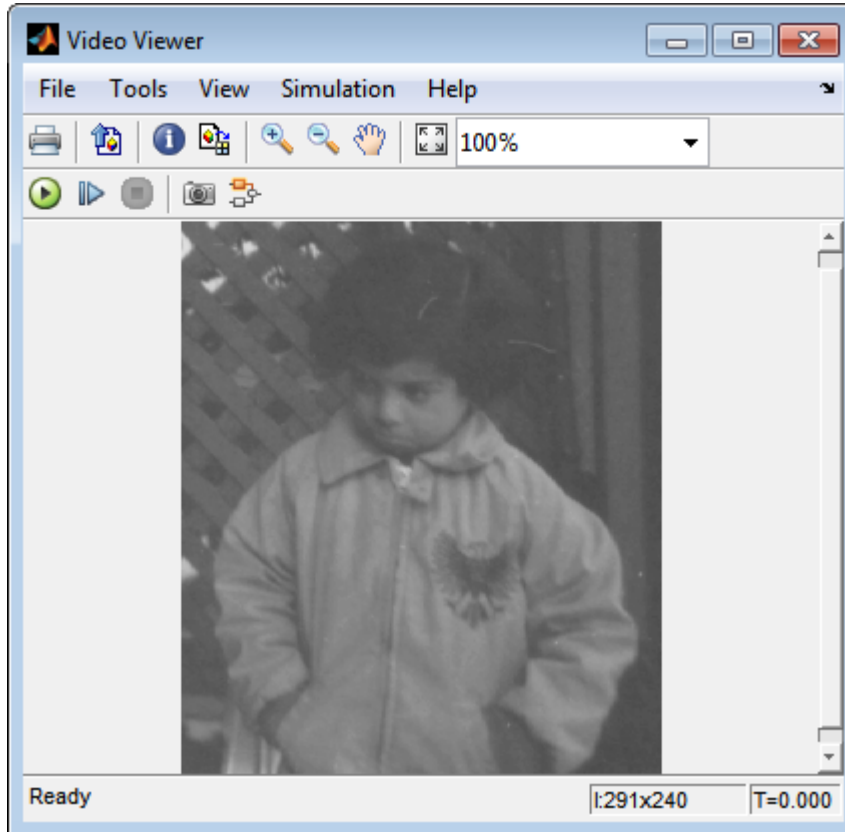
- 2 Place the blocks listed in the table above into your new model.
- 3 Use the Image From File block to import the first image into the Simulink model. Set the **File name** parameter to `pout.tif`.
- 4 Use the Image From File1 block to import the second image into the Simulink model. Set the **File 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.

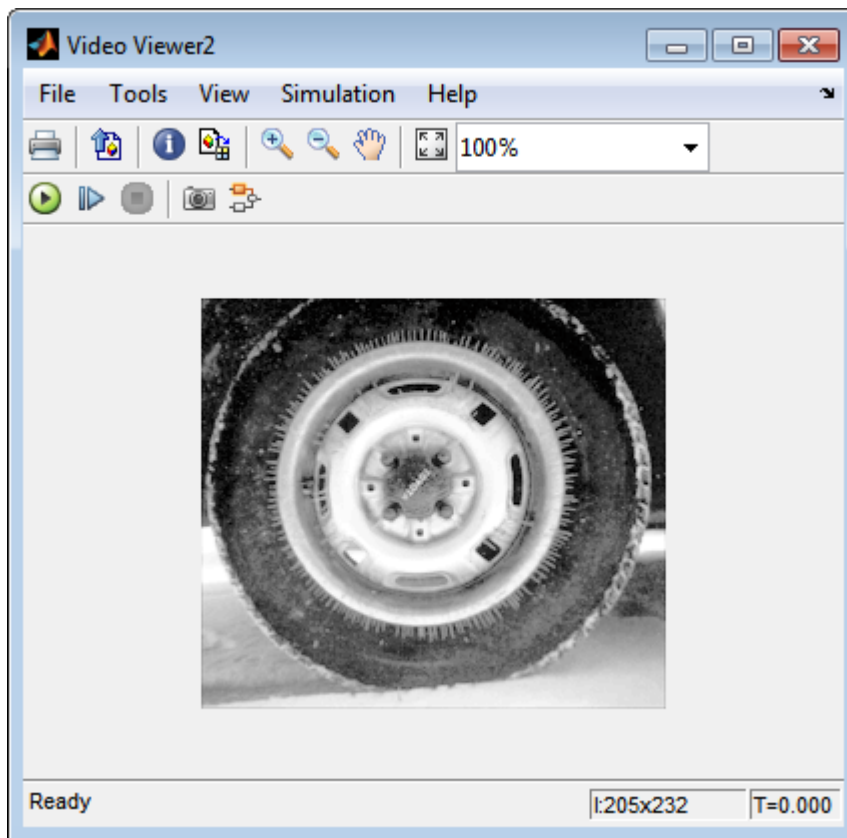


- 9 Set the configuration parameters. Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings > Model Settings**. Set the parameters as follows:
 - **Solver** pane, **Stop time** = 0
 - **Solver** pane, **Type** = Fixed-step
 - **Solver** pane, **Solver** = 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.

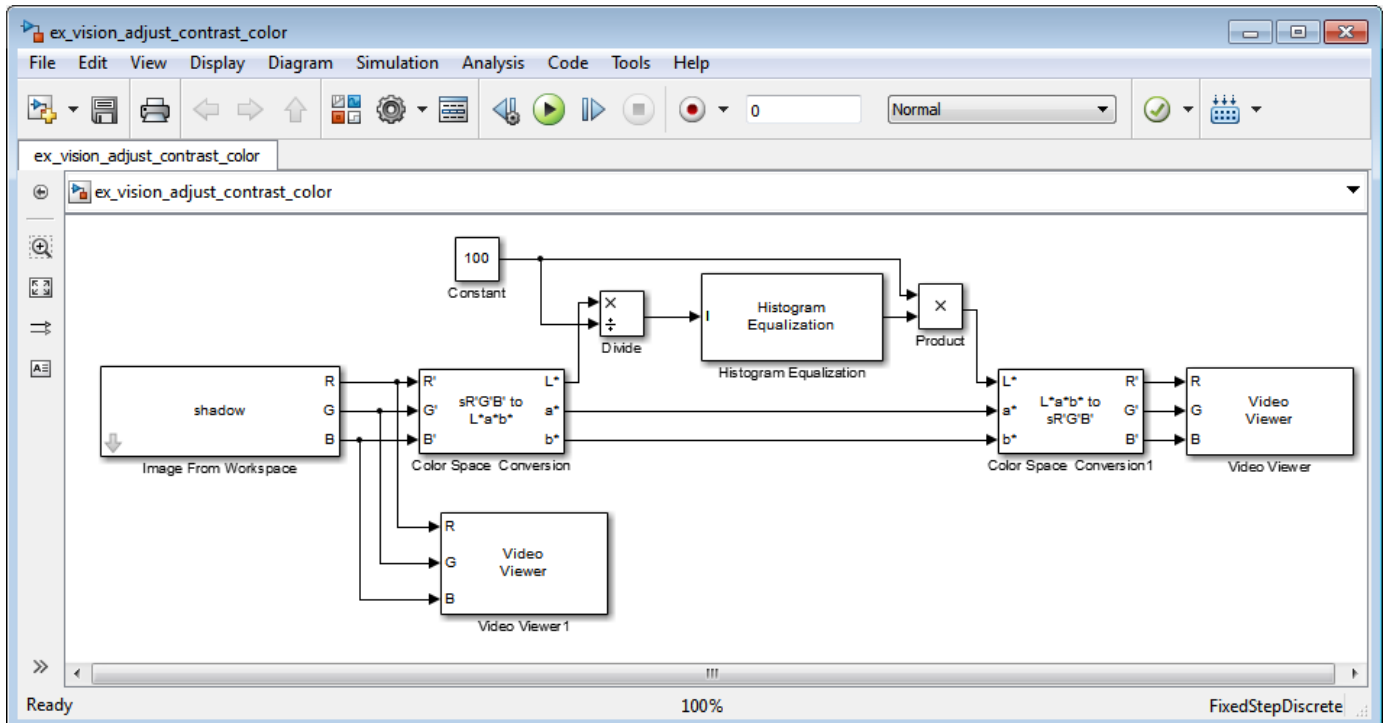
Block	Library	Quantity
Image From Workspace	Computer Vision Toolbox > Sources	1
Color Space Conversion	Computer Vision Toolbox > Conversions	2
Histogram Equalization	Computer Vision Toolbox > Analysis & Enhancement	1
Video Viewer	Computer Vision Toolbox > Sinks	2
Constant	Simulink > Sources	1
Divide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

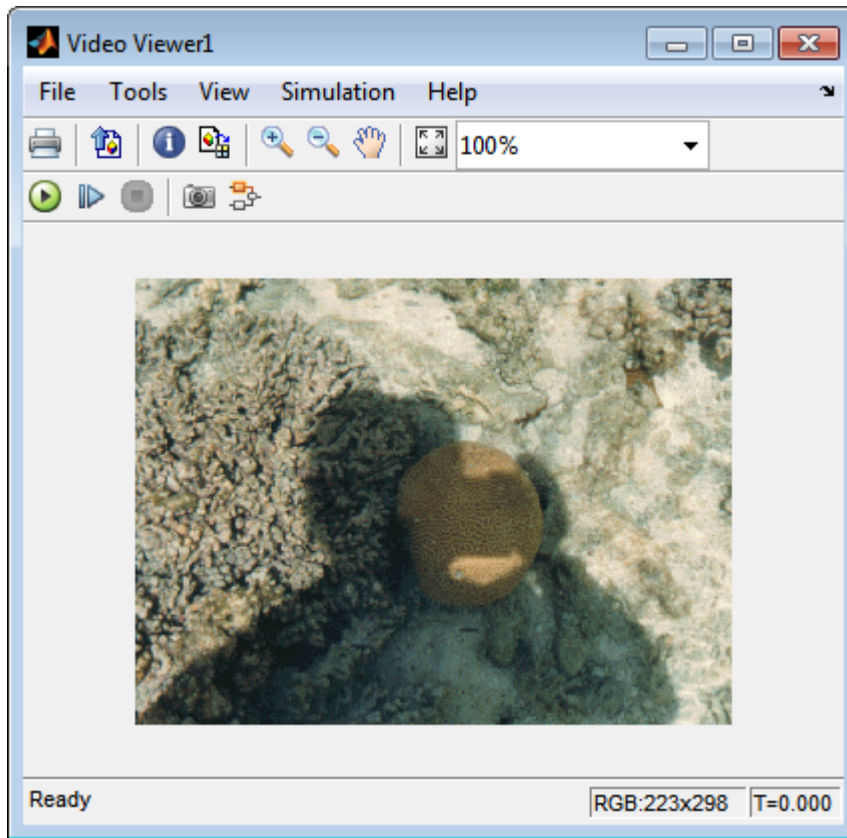
- 6 Use the Constant block to define a normalization factor. Set the **Constant value** parameter to 100.
- 7 Use the Divide block to normalize the L^* values to be between 0 and 1. Accept the default parameters.
- 8 Use the Histogram Equalization block to modify the contrast in the image. This block 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.

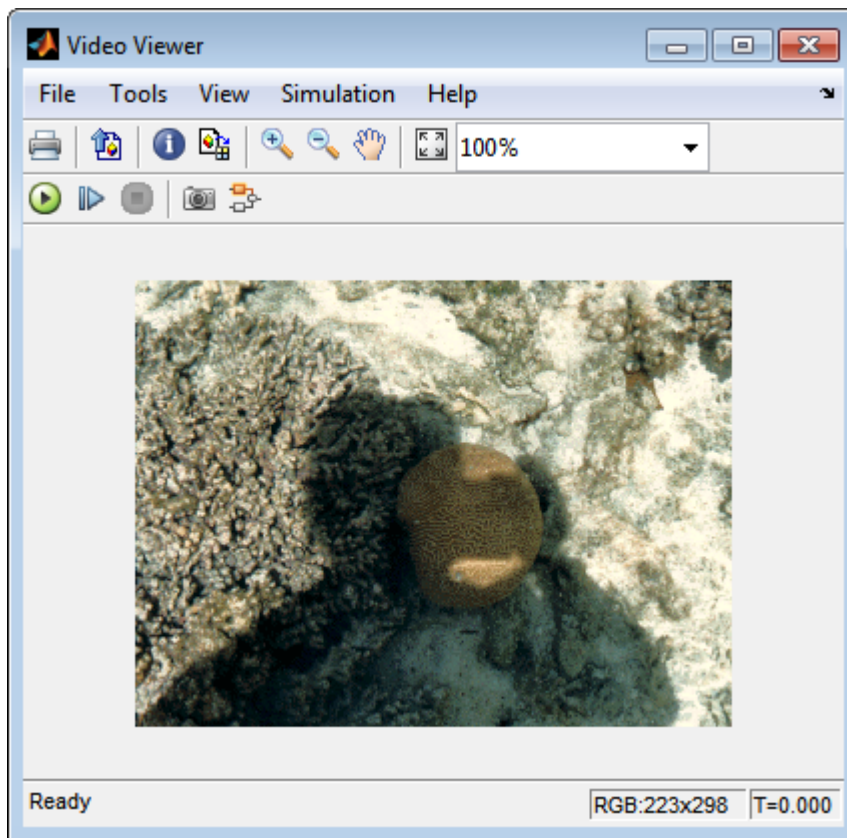


- 13 Set the configuration parameters. Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings > Model Settings**. Set the parameters as follows:
 - **Solver** pane, **Stop time** = 0
 - **Solver** pane, **Type** = Fixed-step
 - **Solver** pane, **Solver** = 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

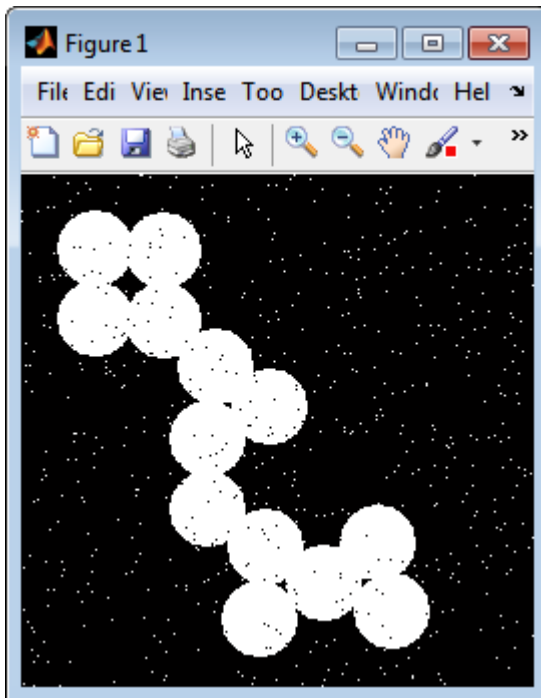
- 1 Define an intensity image in the MATLAB workspace and add noise to it by typing the following at the MATLAB command prompt:

```
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.

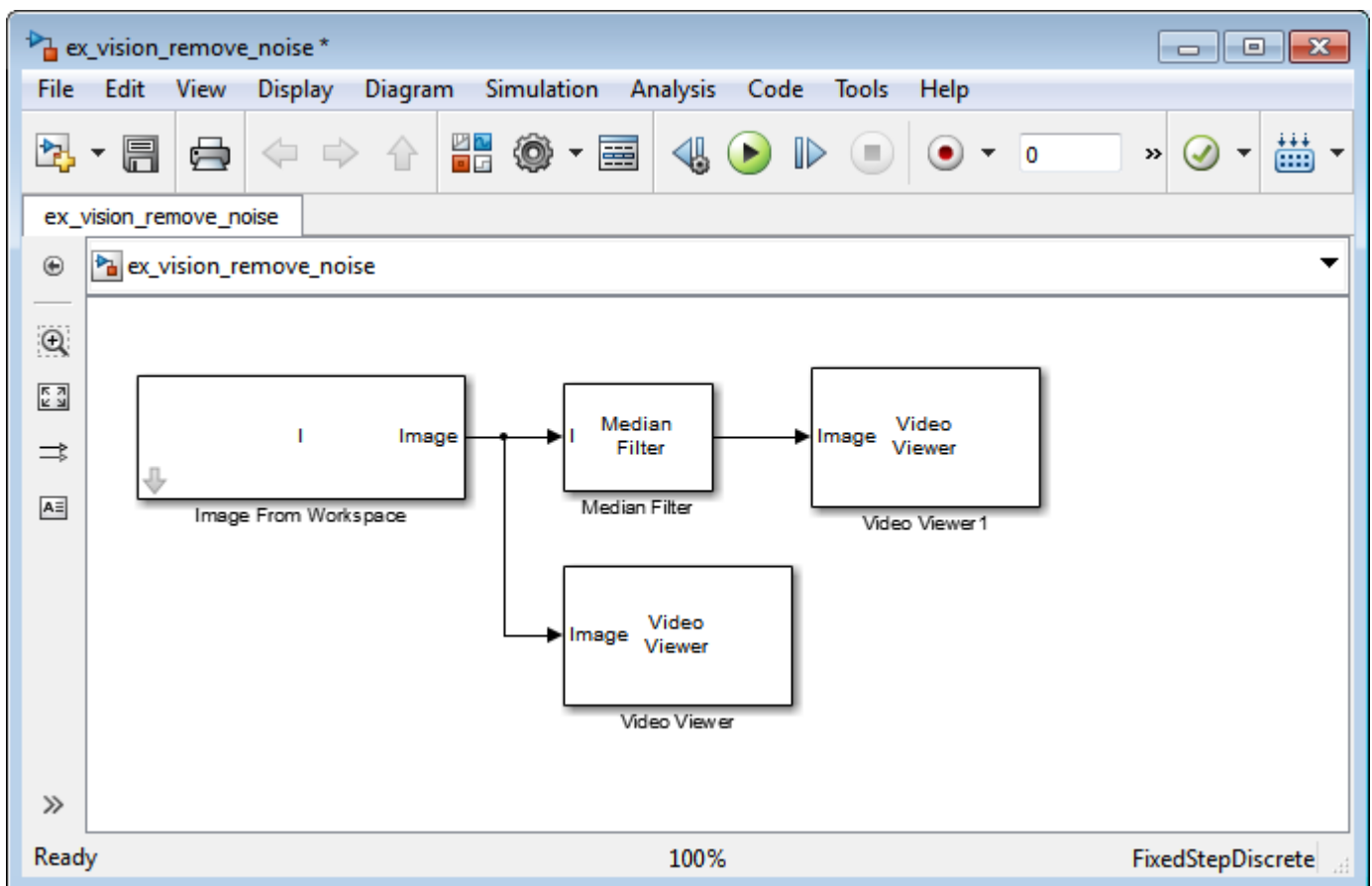
Block	Library	Quantity
Image From Workspace	Computer Vision Toolbox > Sources	1
Median Filter	Computer Vision Toolbox > Filtering	1

Block	Library	Quantity
Video Viewer	Computer Vision 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.

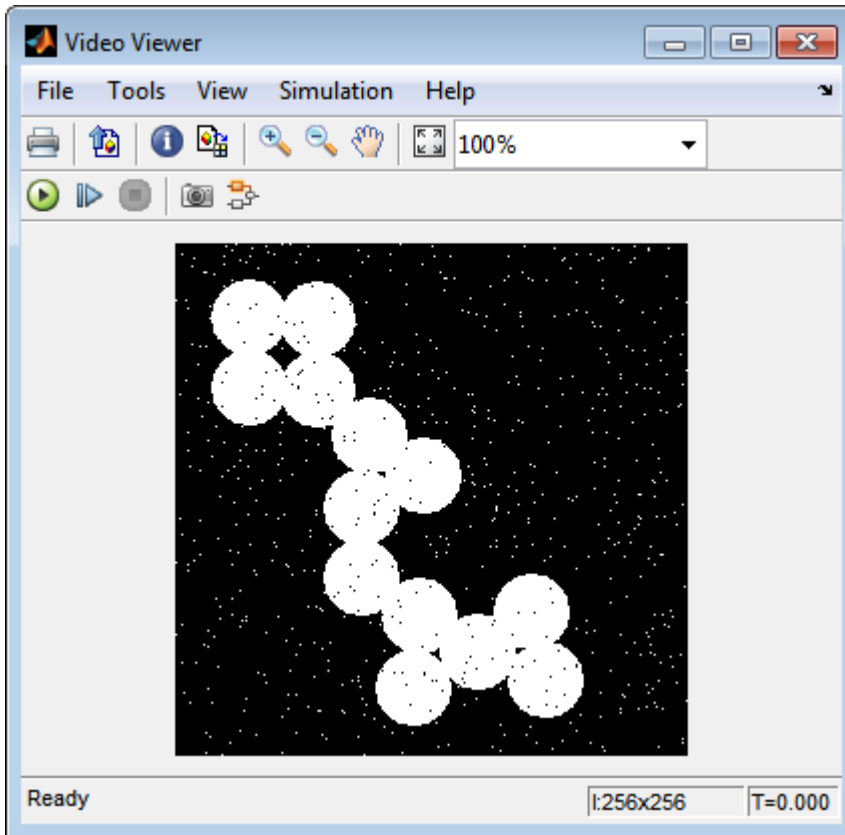
The Median Filter block replaces the central value of the 3-by-3 neighborhood with the median value of the neighborhood. This process removes the noise in the image.

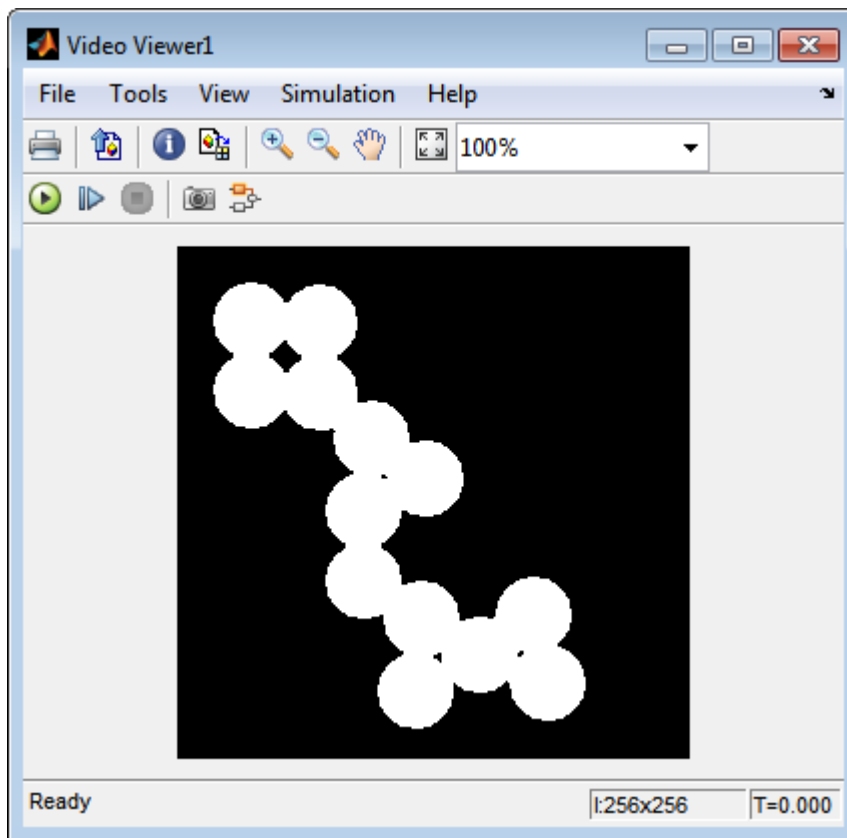
- 6 Use the Video Viewer blocks to display the original noisy image, and the modified image. Images are represented by 8-bit unsigned integers. Therefore, a value of 0 corresponds to black and a value of 255 corresponds to white. Accept the default parameters.
- 7 Connect the blocks as shown in the following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



- 8 Set the configuration parameters. Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings > Model Settings**. 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 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

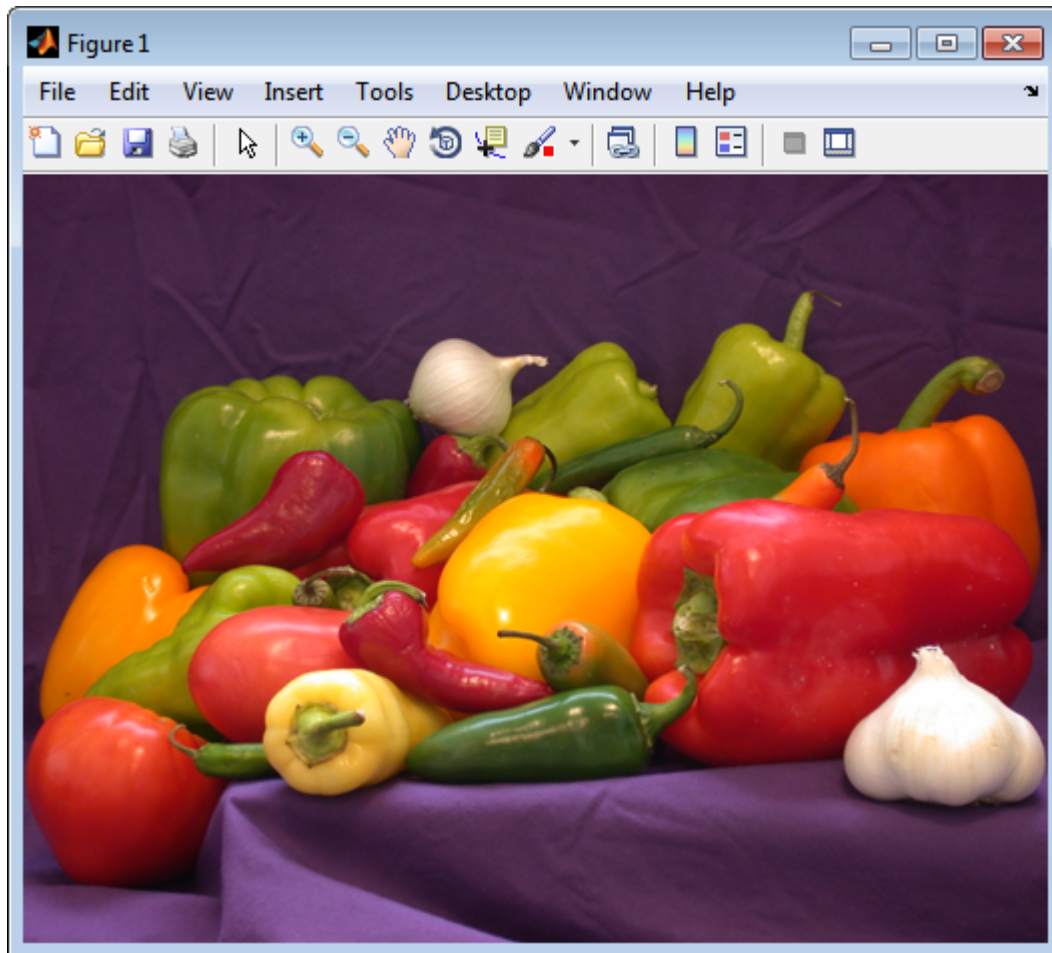
```
I = im2double(imread('peppers.png'));
```

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 Toolbox > Sources	1
Color Space Conversion	Computer Vision Toolbox > Conversions	2
2-D FIR Filter	Computer Vision Toolbox > Filtering	1
Video Viewer	Computer Vision 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

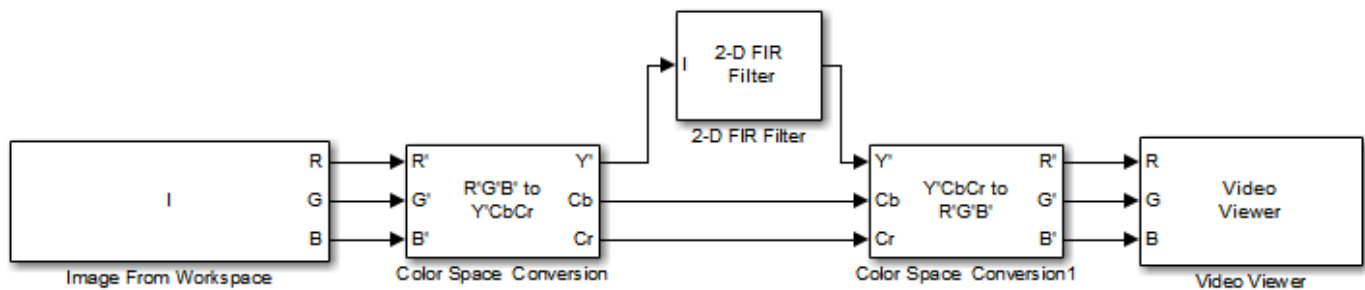
The block outputs the R', G', and B' planes of the I array at the output ports.

- 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

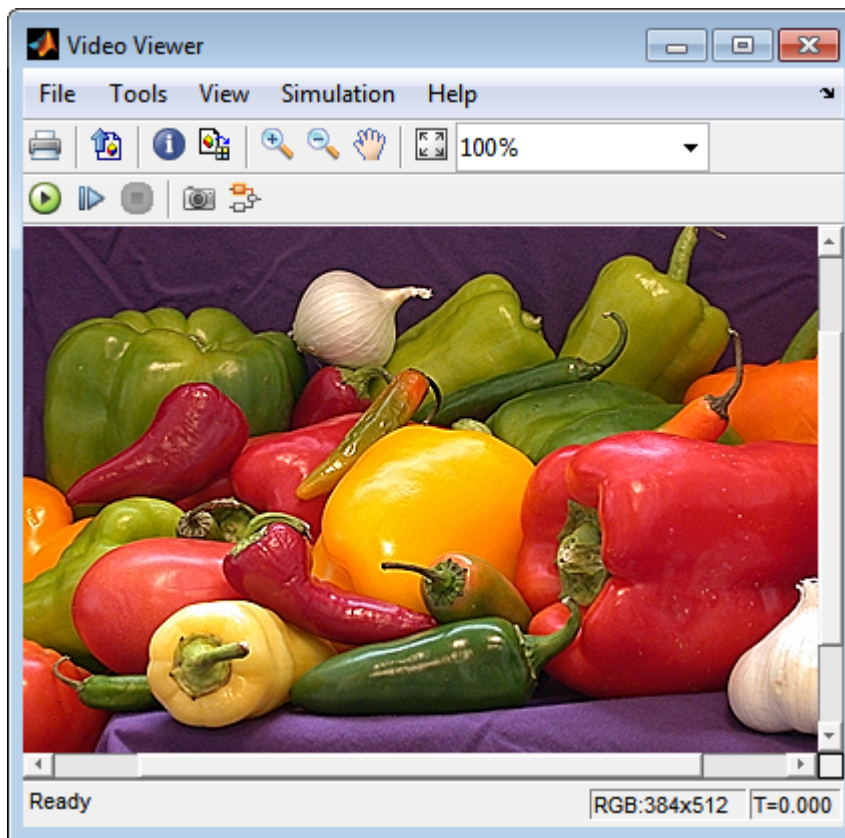
The `fspecial('unsharp')` command creates two-dimensional highpass filter coefficients suitable for correlation. This highpass filter sharpens the image by removing the low frequency noise in it.

- 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
- 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**.
- Connect the blocks as shown in the following figure.



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

- “Correct Nonuniform Illumination” on page 18-2
- “Count Objects in an Image” on page 18-8

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.

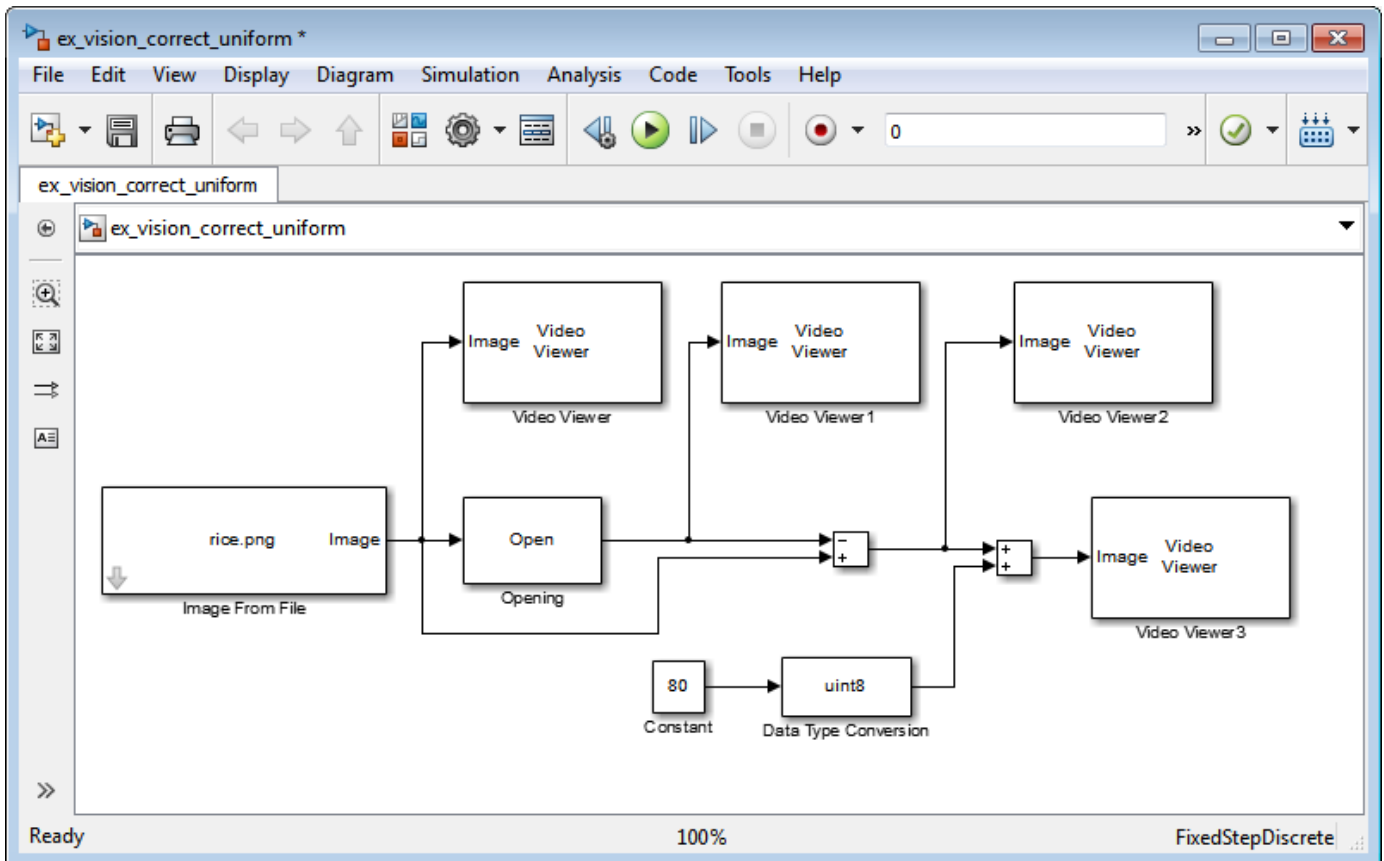
Block	Library	Quantity
Image From File	Computer Vision Toolbox > Sources	1
Opening	Computer Vision Toolbox > Morphological Operations	1
Video Viewer	Computer Vision Toolbox > Sinks	4
Constant	Simulink > Sources	1
Sum	Simulink > Math Operations	2
Data Type Conversion	Simulink > Signal Attributes	1

- 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.

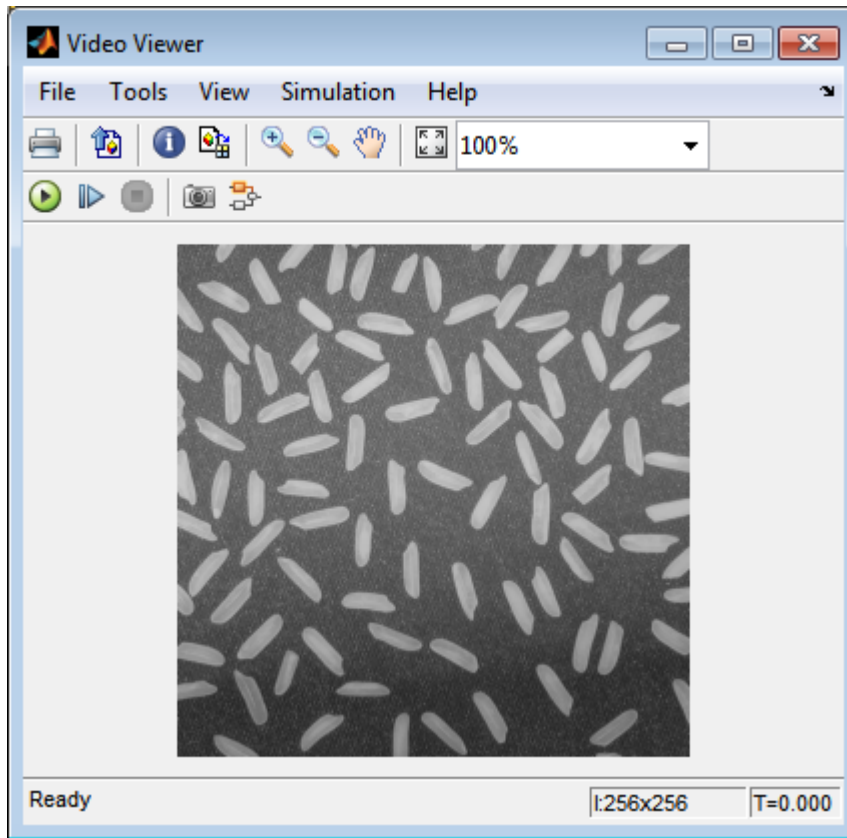
- 5 Use the Video Viewer1 block to view the background estimated by the Opening block. Accept the default parameters.
- 6 Use the first Add block to subtract the estimated background from the original image. Set the block parameters as follows:
 - **Icon shape** = `rectangular`
 - **List of signs** = `--+`
- 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.

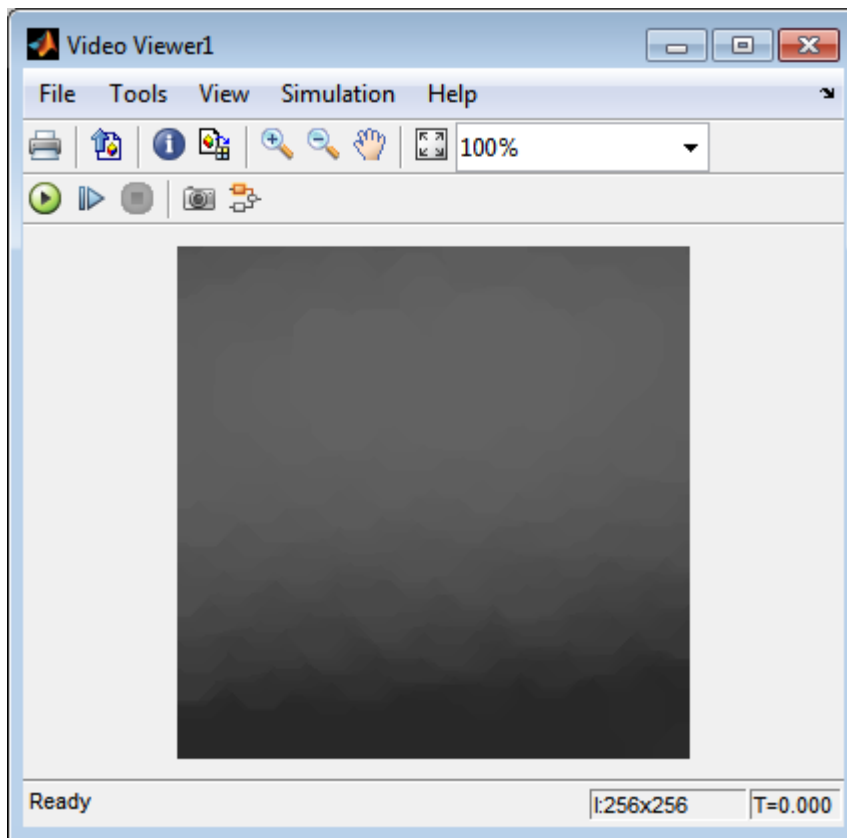


- 13 Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings > Model Settings**. Set the **Solver** parameters as follows:
 - **Solver** pane, **Stop time** = 0
 - **Solver** pane, **Type** = Fixed-step
 - **Solver** pane, **Solver** = discrete (no continuous states)
- 14 Run the model.

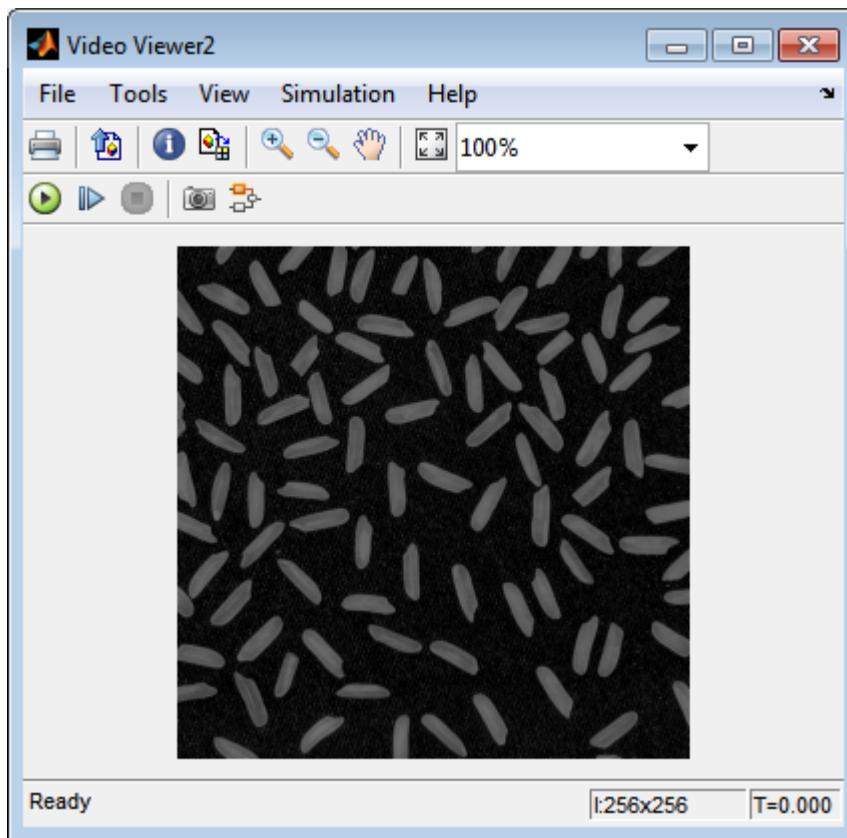
The original image appears in the Video Viewer window.



The estimated background appears in the Video Viewer1 window.

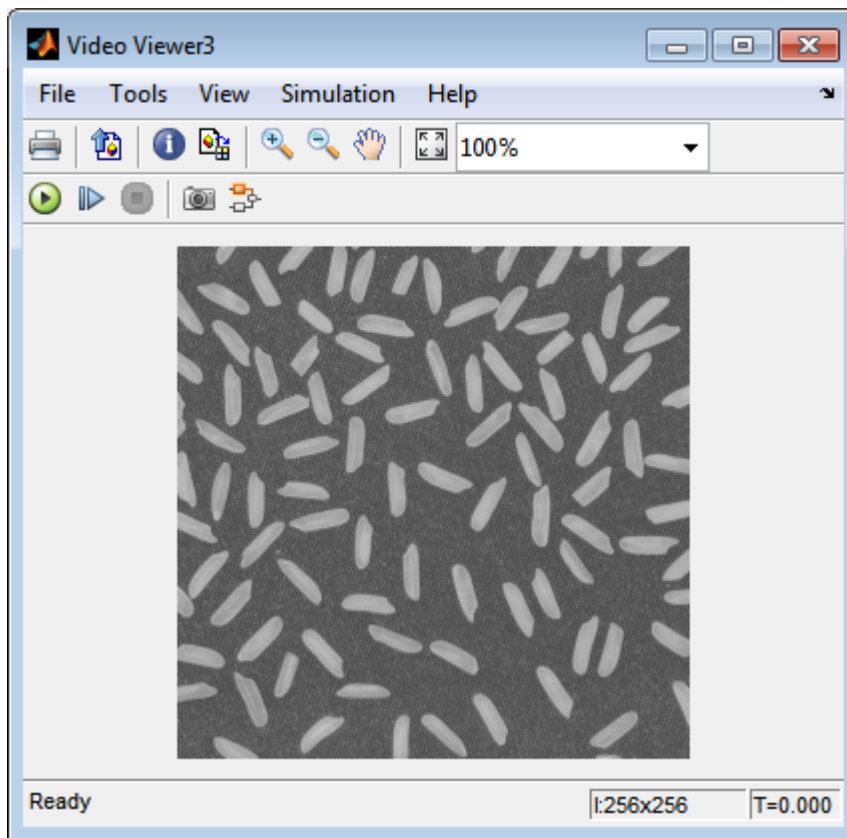


The image without the estimated background appears in the Video Viewer2 window.



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 Toolbox > Sources	1
Opening	Computer Vision Toolbox > Morphological Operations	1
Label	Computer Vision Toolbox > Morphological Operations	1
Video Viewer	Computer Vision 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 `<`.

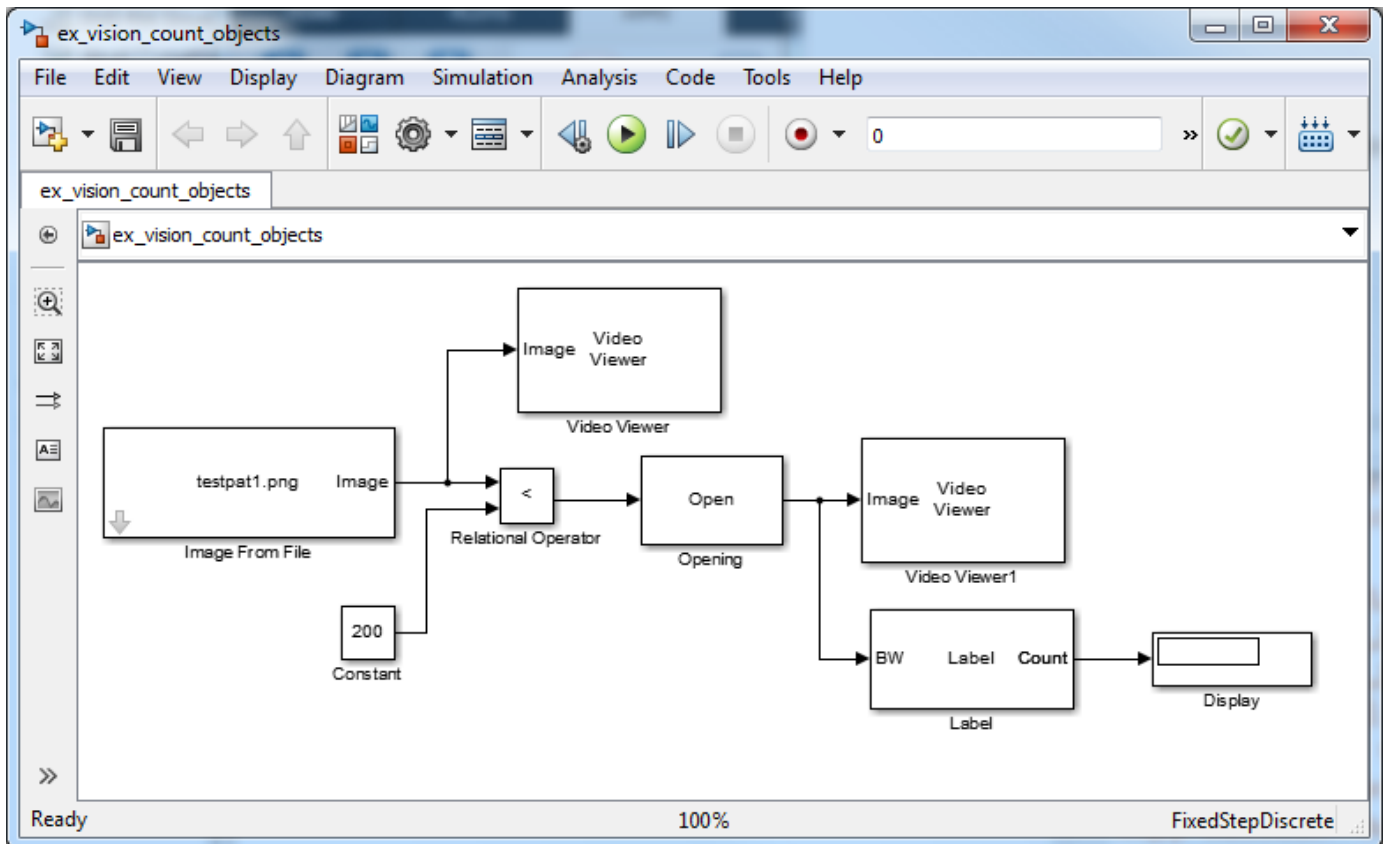
If the input to the Relational Operator block is less than 200, its output is 1; otherwise, its output is 0. You must threshold your intensity image because the Label block expects binary input. Also, the objects it counts must be white.

- 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.

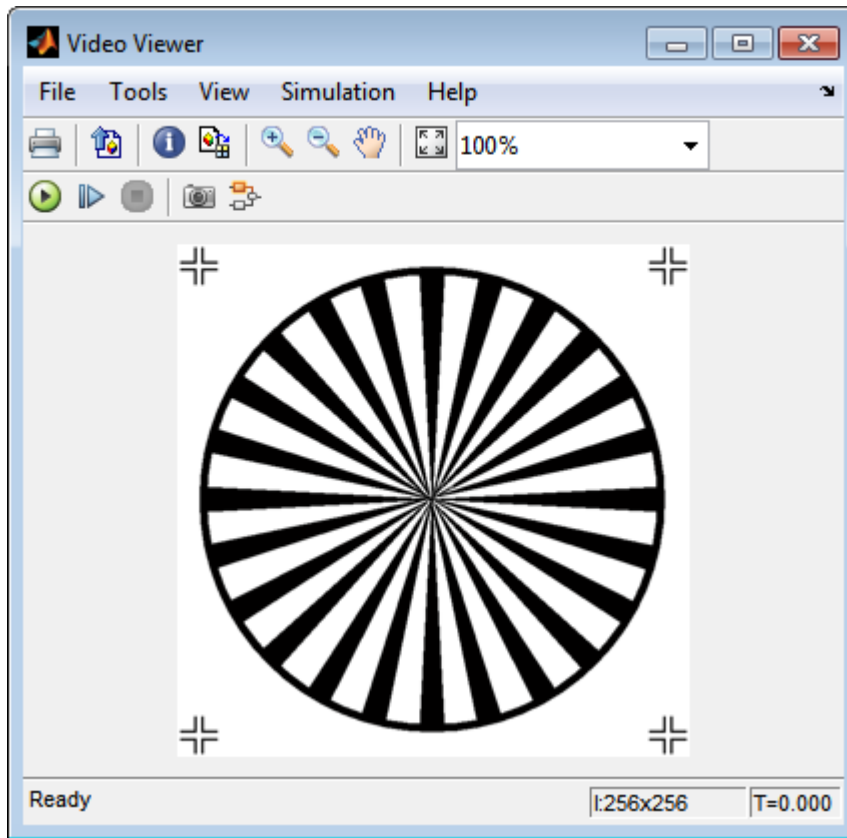
- 7 Use the Video Viewer1 block to view the opened image. Accept the default parameters.
- 8 Use the Label block to count the number of spokes in the input image. Set the **Output** parameter to `Number of labels`.

- 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.

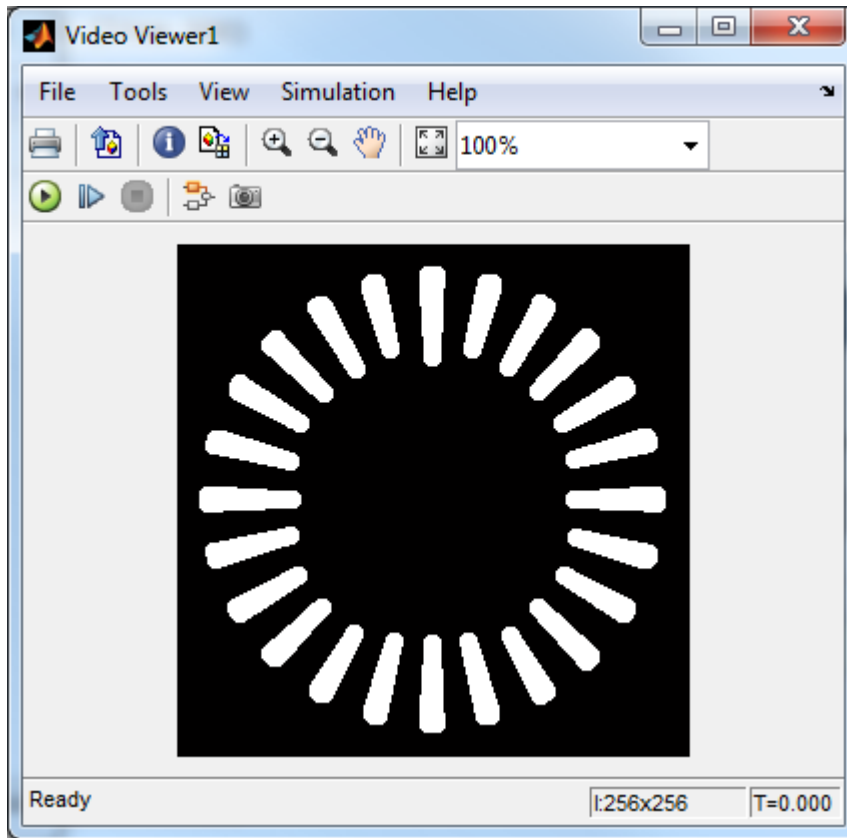


- 11 Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings > Model Settings**. Set the **Solver** parameters as follows:
 - **Solver** pane, **Stop time** = 0
 - **Solver** pane, **Type** = Fixed-step
 - **Solver** pane, **Solver** = discrete (no continuous states)
- 12 Run the model.

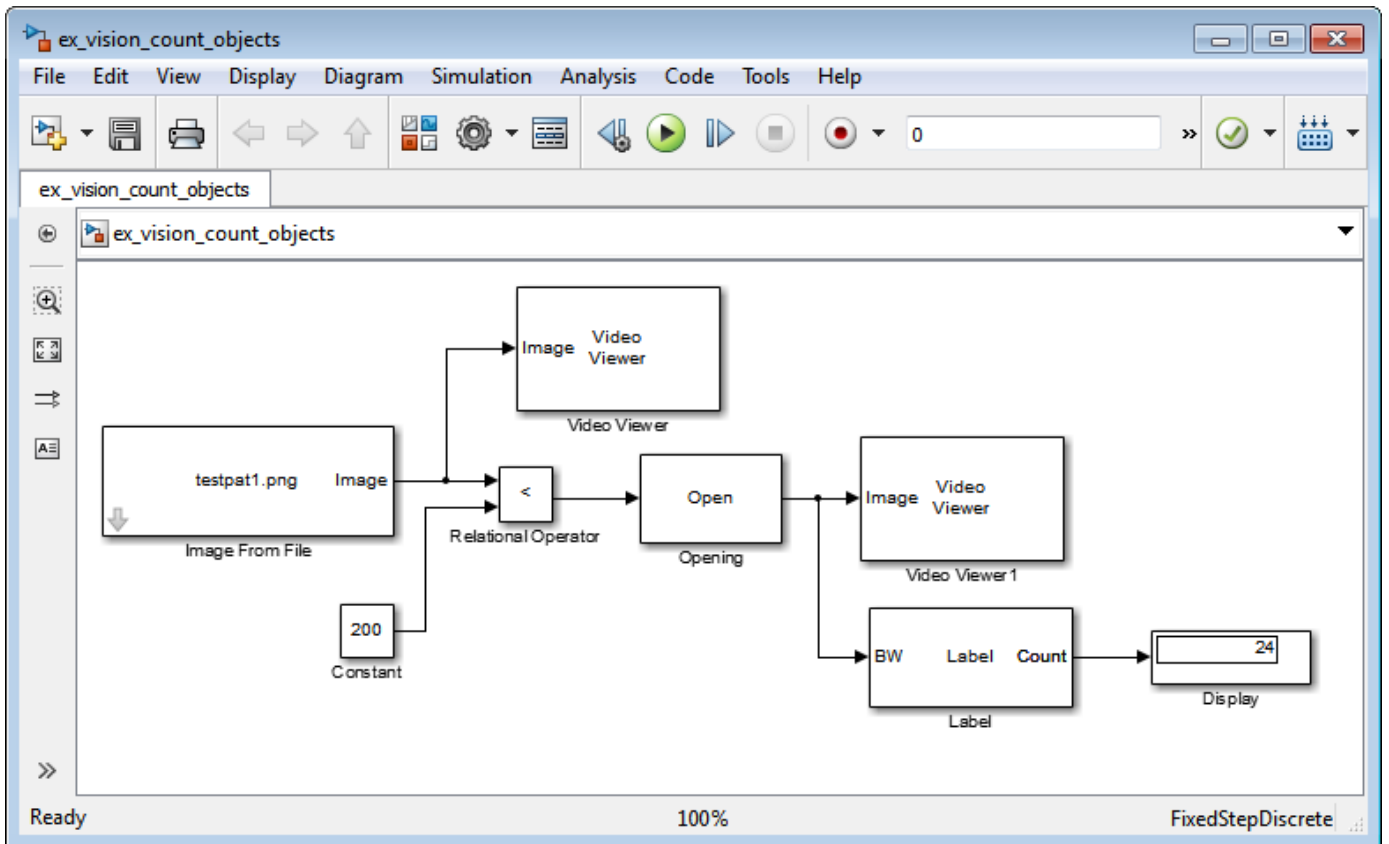
The original image appears in the Video Viewer1 window. To view the image at its true size, right-click the window and select **Set Display To True Size**.



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 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.

Fixed-Point Design

- “Fixed-Point Signal Processing” on page 19-2
- “Fixed-Point Concepts and Terminology” on page 19-4
- “Arithmetic Operations” on page 19-8
- “Fixed-Point Support for MATLAB System Objects” on page 19-15
- “Specify Fixed-Point Attributes for Blocks” on page 19-17

Fixed-Point Signal Processing

In this section...
“Fixed-Point Features” on page 19-2
“Benefits of Fixed-Point Hardware” on page 19-2
“Benefits of Fixed-Point Design with System Toolboxes Software” on page 19-2

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 19-4

“Scaling” on page 19-5

“Precision and Range” on page 19-6

Fixed-Point Data Types

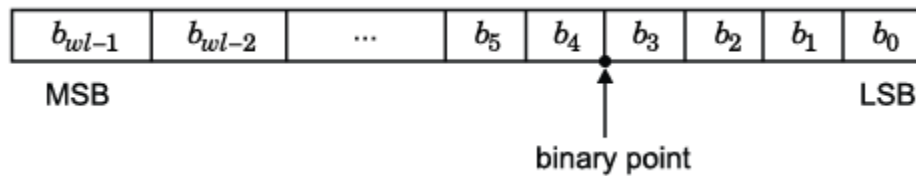
In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). The way hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

Binary numbers are represented as either floating-point or fixed-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 the signedness of a number which can be signed or unsigned. Signed numbers and data types can represent both positive and negative values, whereas unsigned numbers and data types can only represent values that are greater than or equal to zero.

The position of the binary point is the means by which fixed-point values are scaled and interpreted.

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



where

- b_i is the i^{th} binary digit.
- wl is the number of bits in a binary word, also known as word length.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB). In signed binary numbers, this bit is the sign bit which indicates whether the number is positive or negative.
- b_0 is the location of the least significant, or lowest, bit (LSB). This bit in the binary word can represent the smallest value. The weight of the LSB is given by:

$$weight_{LSB} = 2^{-fractionlength}$$

where, *fractionlength* is the number of bits to the right of the binary point.

- Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits. Number of bits to the left of the binary point is known as the integer length. The binary point in this example is shown four places to the left of the LSB. 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 -- Representation of signed fixed-point or floating-point numbers. In the sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.
- One's complement
- Two's complement -- Two's complement is the most common representation of signed fixed-point numbers. See "Two's Complement" on page 19-8 for more information.

Unsigned fixed-point numbers can only represent numbers greater than or equal to zero.

Scaling

In [Slope Bias] representation, fixed-point numbers can be encoded according to the scheme

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

where the slope can be expressed as

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

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

In the trivial case, slope = 1 and bias = 0. Scaling is always trivial for pure integers, such as int8, and also for the true floating-point types single and double.

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

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

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.

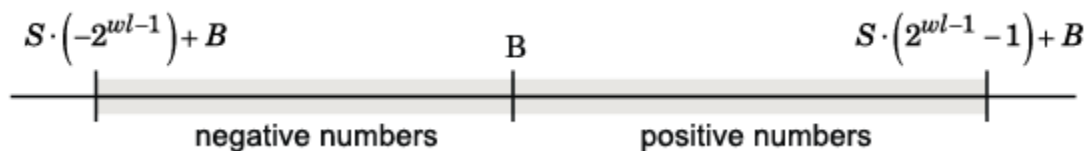
All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

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

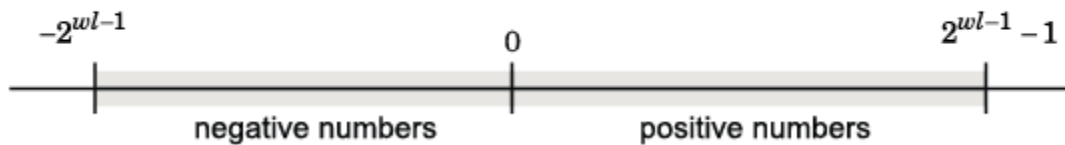
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl , scaling S , and bias B is illustrated below:



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:



The full range is the broadest range for a data type. For floating-point types, the full range is $-\infty$ to ∞ . For integer types, the full range is the range from the smallest to largest integer value (finite) the type can represent. For example, from -128 to 127 for a signed 8-bit integer.

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. Guard bits are extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow. 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 19-8 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. The term resolution is sometimes used as a synonym for this definition.

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. The truncation operation results in dropping of one or more least significant bits from a number.
- **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

In this section...

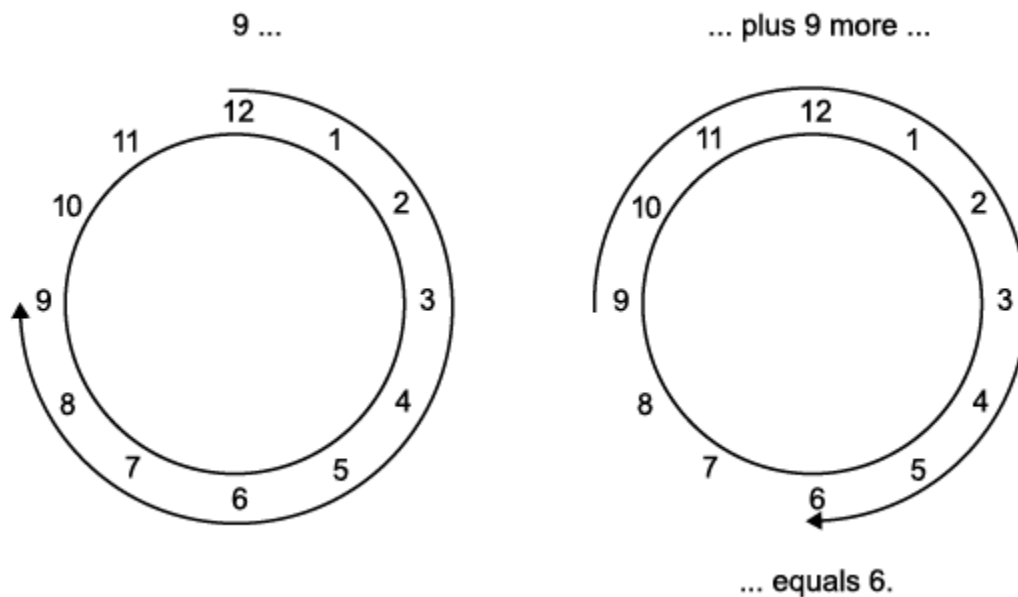
“Modulo Arithmetic” on page 19-8
 “Two's Complement” on page 19-8
 “Addition and Subtraction” on page 19-9
 “Multiplication” on page 19-10
 “Casts” on page 19-12

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 common representation of signed fixed-point numbers. 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. That is, all 0's are flipped to 1's and all 1's are flipped to 0's.
- 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:

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

Addition and Subtraction

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

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

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

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

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

Carry bit is discarded.

Most fixed-point DSP System Toolbox blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting is necessary during the addition to line up the binary points. See "Casts" on page 19-12 for more information.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):

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

The extra 1 is the result of necessary sign extension.

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

Multiplication Data Types

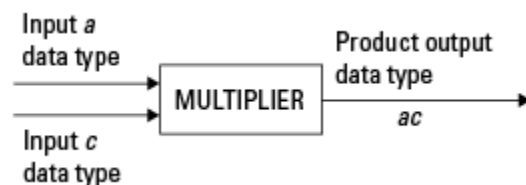
The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages to determine whether a particular block accepts complex fixed-point inputs.

In most cases, you can set the data types used during multiplication in the block mask. For details, see “Casts” on page 19-12.

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.

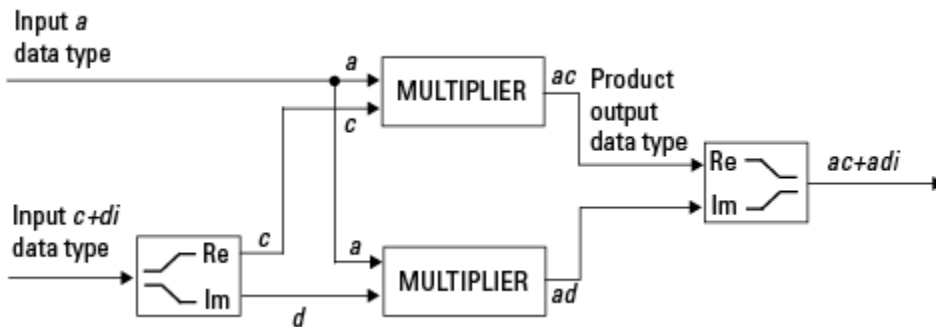
Real-Real Multiplication

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.



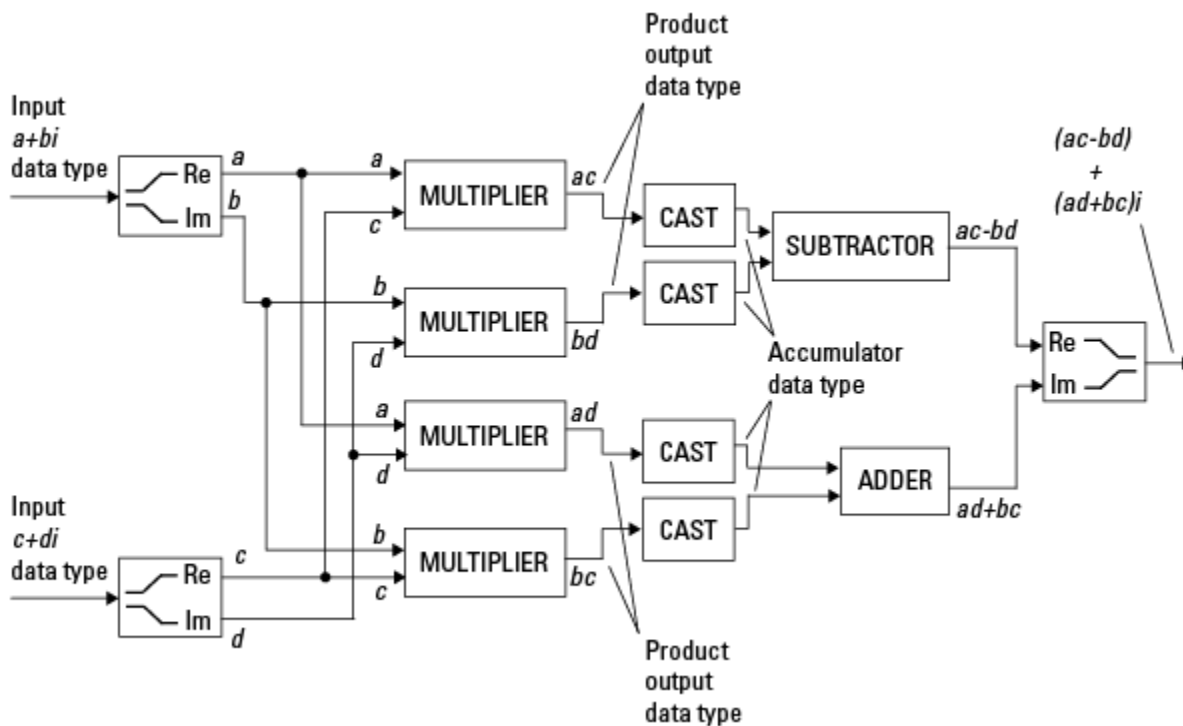
Real-Complex Multiplication

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. Sign extension is the addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number. Padding is extending the least significant bit of a binary word with one or more zeros.

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.

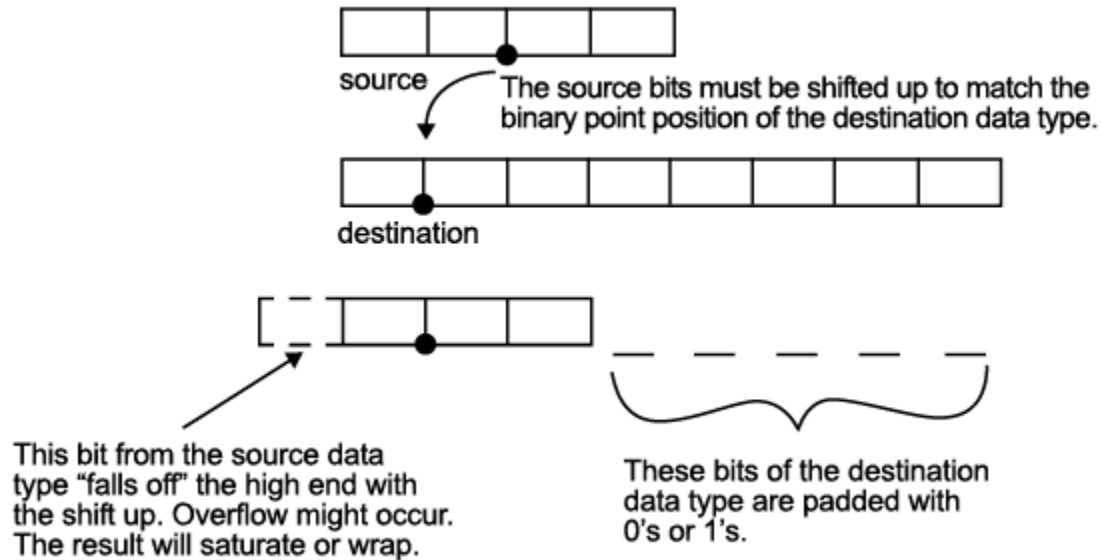
Note that although you cannot mix fixed-point and floating-point signals on the input and output ports of blocks, you can have fixed-point signals with different word and fraction lengths on the ports of blocks that support fixed-point signals.

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:



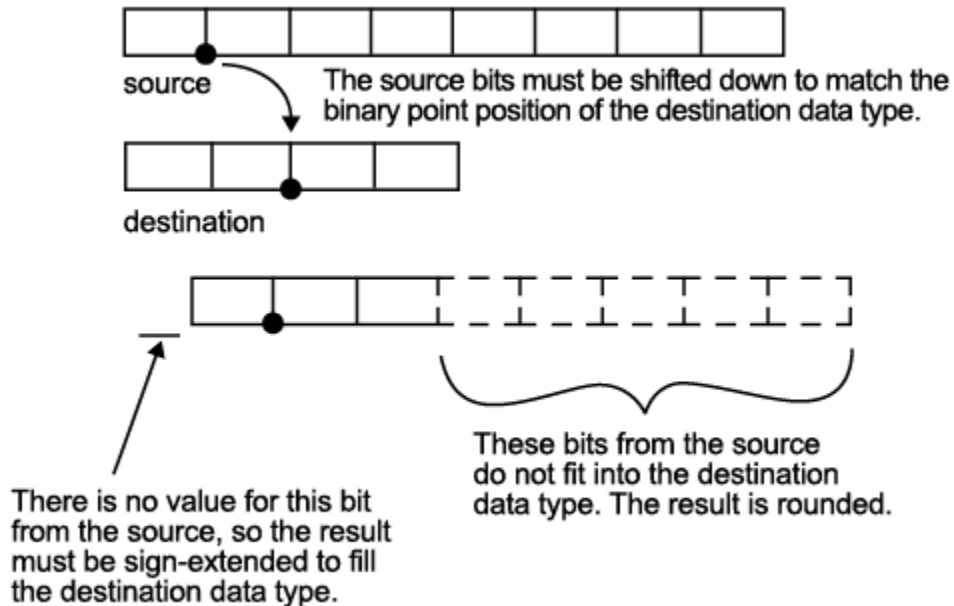
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either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

In this section...

“Getting Information About Fixed-Point System Objects” on page 19-15

“Setting System Object Fixed-Point Properties” on page 19-15

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 Toolbox objects support fixed-point data processing.

Fixed-Point Data Processing Support

```
vision.AlphaBlender
vision.BlobAnalysis
vision.BlockMatcher
vision.DCT
vision.Maximum
vision.Mean
vision.Median
vision.Minimum
```

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” on page 19-17
 “Specify System-Level Settings” on page 19-19
 “Inherit via Internal Rule” on page 19-19
 “Specify Data Types for Fixed-Point Blocks” on page 19-26

Fixed-Point Block Parameters

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 toolbox blocks. The following figure shows a typical **Data Types** pane.

Fixed-point operational parameters

Rounding mode: 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:	<input type="text" value="Inherit: Same word length as i"/>	N/A	N/A
Product output:	<input type="text" value="Inherit: Inherit via internal rule"/>	N/A	N/A
Accumulator:	<input type="text" value="Inherit: Inherit via internal rule"/>	N/A	N/A
Output:	<input type="text" value="Inherit: Inherit via internal rule"/>	<input type="text" value=""/>	<input type="text" value=""/>

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

All 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	<p>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.</p> <p>See “Rounding Modes” on page 19-7 for more information on the available options.</p>
Saturate on integer overflow	<p>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.</p> <p>For details on saturate and wrap, see “Overflow Handling” on page 19-6 for fixed-point operations.</p>
Intermediate Product	<p>Specifies the data type and scaling of the intermediate product for fixed-point blocks. Blocks that feed multiplication results back to the input of the multiplier use the intermediate product data type.</p> <p>See the reference page of a specific block to learn about the intermediate product data type for that block.</p>
Product Output	<p>Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results.</p> <p>See the reference page of a specific block to learn about the product output data type for that block. For or complex-complex multiplication, the multiplication result is in the accumulator data type. See “Multiplication Data Types” on page 19-10 for more information on complex fixed-point multiplication in toolbox software.</p>
Accumulator	<p>Specifies the data type and scaling of the accumulator (sum) for fixed-point blocks that must hold summation results for further calculation. Most such blocks cast to the accumulator data type before performing the add operations (summation).</p> <p>See the reference page of a specific block for details on the accumulator data type of that block.</p>
Output	Specifies the output data type and scaling for blocks.

Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool available on the **Data Types** pane of some fixed-point 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 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 “Specify Signal Ranges” (Simulink).

Specify System-Level Settings

You can monitor and control fixed-point settings for toolbox blocks at a system or subsystem level with the Fixed-Point Tool. For more information, see **Fixed-Point Tool**.

Logging

The Fixed-Point Tool logs overflows, saturations, and simulation minimums and maximums for fixed-point 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 toolbox fixed-point data types.

Data type override

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 toolboxes source and byte-shuffling blocks, and for some arithmetic blocks such as Difference and Normalization.

Scaled double is a double data type that retains fixed-point scaling information. Using the data type override, you can convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information, you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.

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 toolbox software:

- “Internal Rule for Accumulator Data Types” on page 19-20
- “Internal Rule for Product Data Types” on page 19-20
- “Internal Rule for Output Data Types” on page 19-20
- “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 19-20
- “Internal Rule Examples” on page 19-21

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:

$$WL_{idealaccumulator} = WL_{inputtoaccumulator} + \text{floor}(\log_2(N - 1)) + 1$$

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$

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 19-20 for more information.

Internal Rule for Product Data Types

The internal rule for product data types first calculates the ideal, full-precision result:

$$WL_{idealproduct} = WL_{input1} + WL_{input2}$$

$$FL_{idealproduct} = FL_{input1} + FL_{input2}$$

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 19-20 for more information.

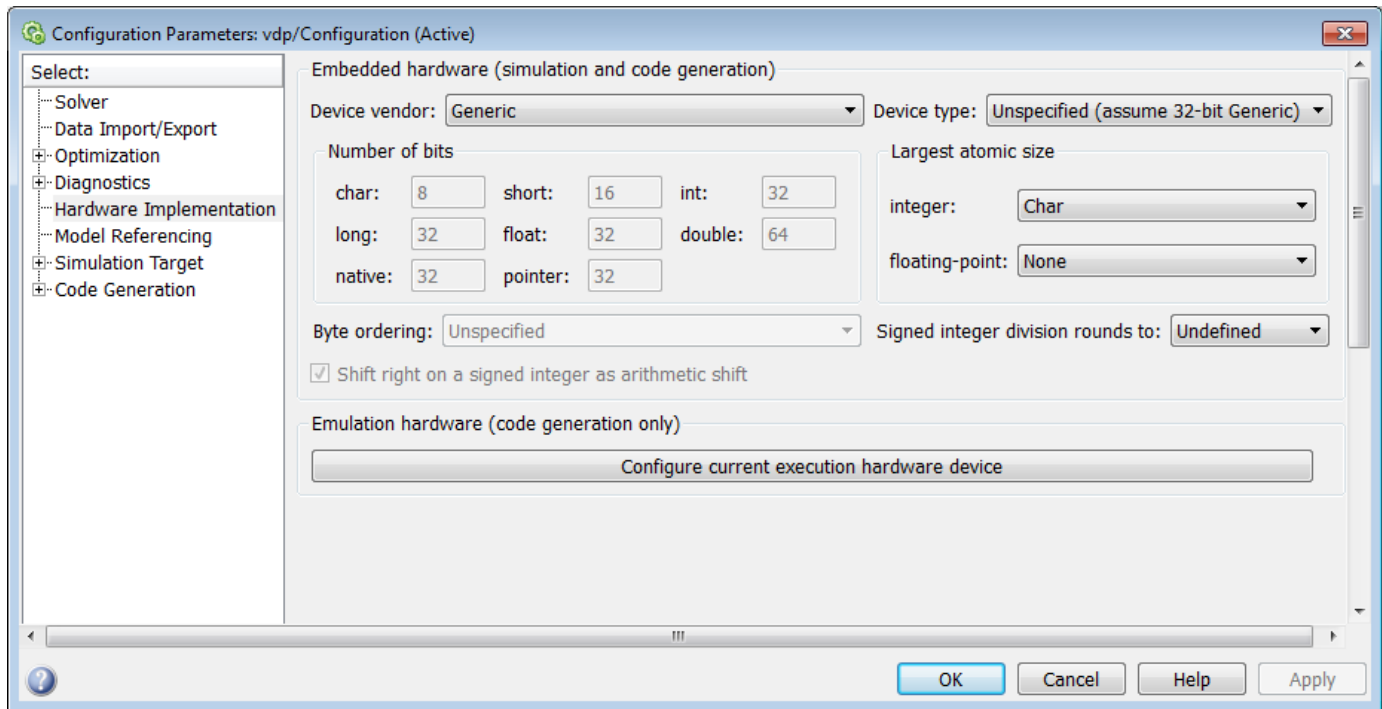
Internal Rule for Output Data Types

A few 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 19-20.

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. To open this dialog box, click **Modeling > Model Settings** in the Simulink toolstrip.



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.

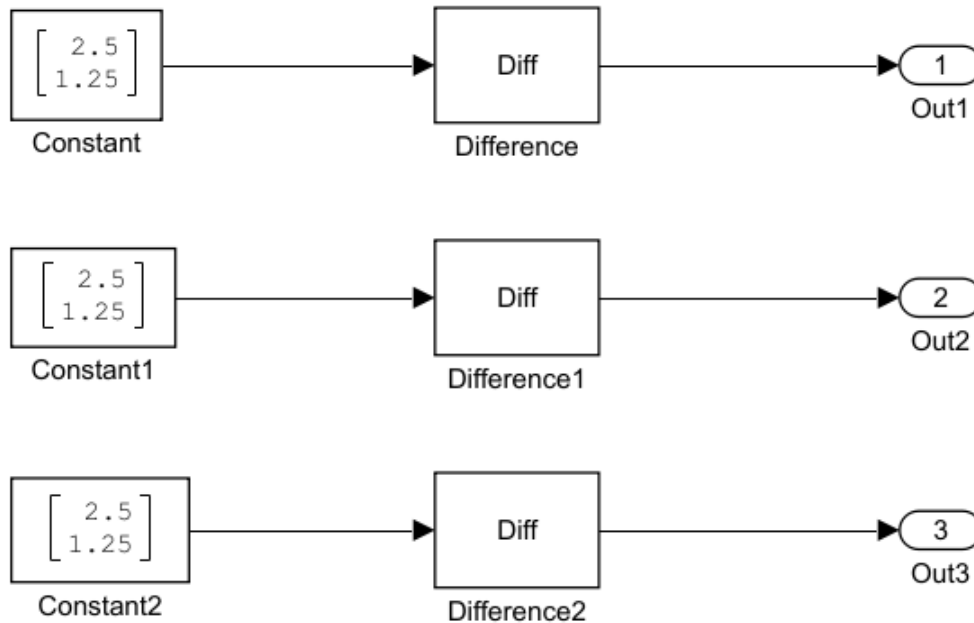
The largest word length allowed for Simulink and toolbox software on any target is 128 bits.

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 19-21 and product data types on page 19-24.

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:

$$WL_{idealaccumulator} = WL_{inputtoaccumulator} + \text{floor}(\log_2(\text{numberofaccumulations})) + 1$$

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

$$WL_{idealaccumulator} = 9 + 0 + 1 = 10$$

$$WL_{idealaccumulator1} = WL_{inputtoaccumulator1} + \text{floor}(\log_2(\text{numberofaccumulations})) + 1$$

$$WL_{idealaccumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$

$$WL_{idealaccumulator1} = 16 + 0 + 1 = 17$$

$$WL_{idealaccumulator2} = WL_{inputtoaccumulator2} + \text{floor}(\log_2(\text{numberofaccumulations})) + 1$$

$$WL_{idealaccumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$

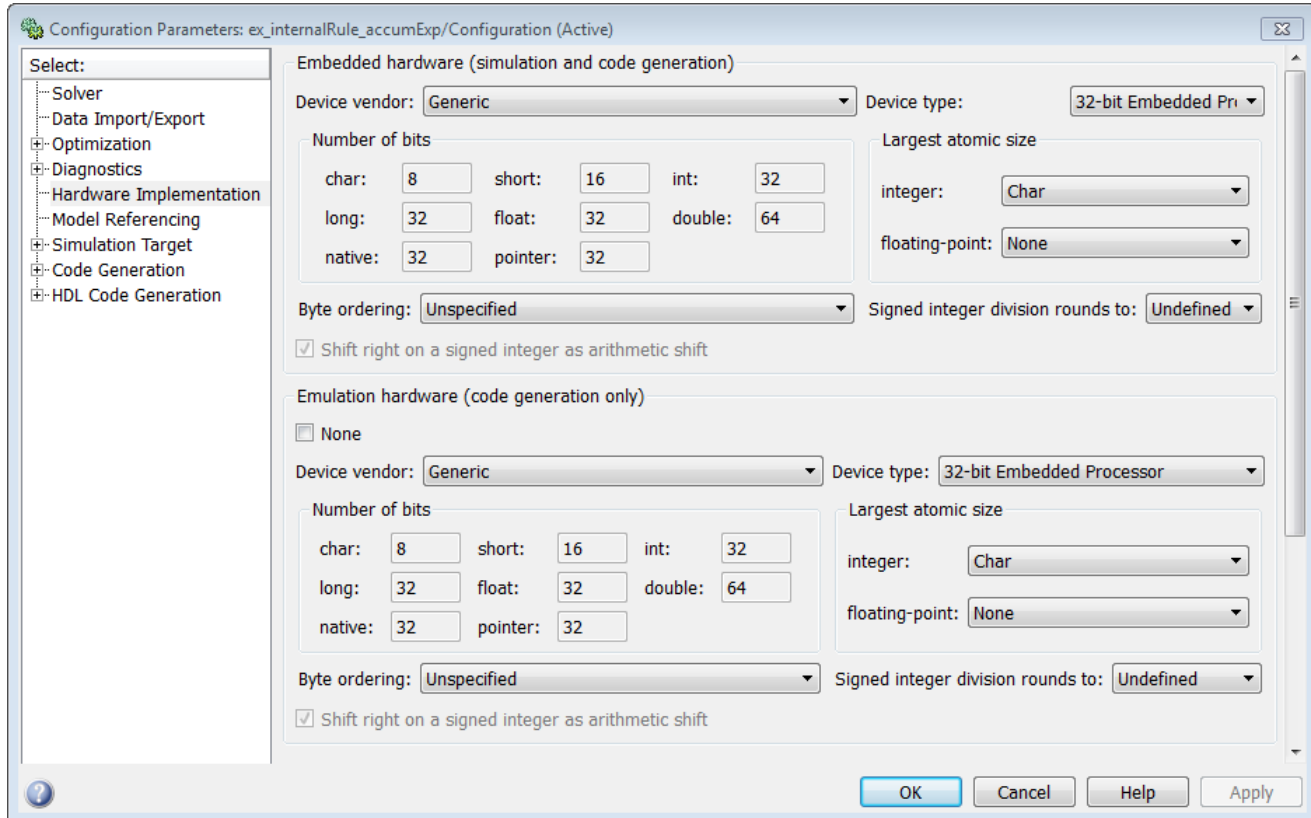
$$WL_{idealaccumulator2} = 127 + 0 + 1 = 128$$

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

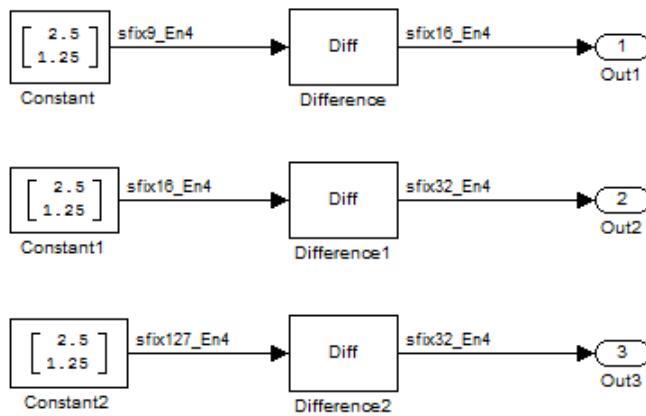
$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$

$$FL_{idealaccumulator} = 4$$

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.

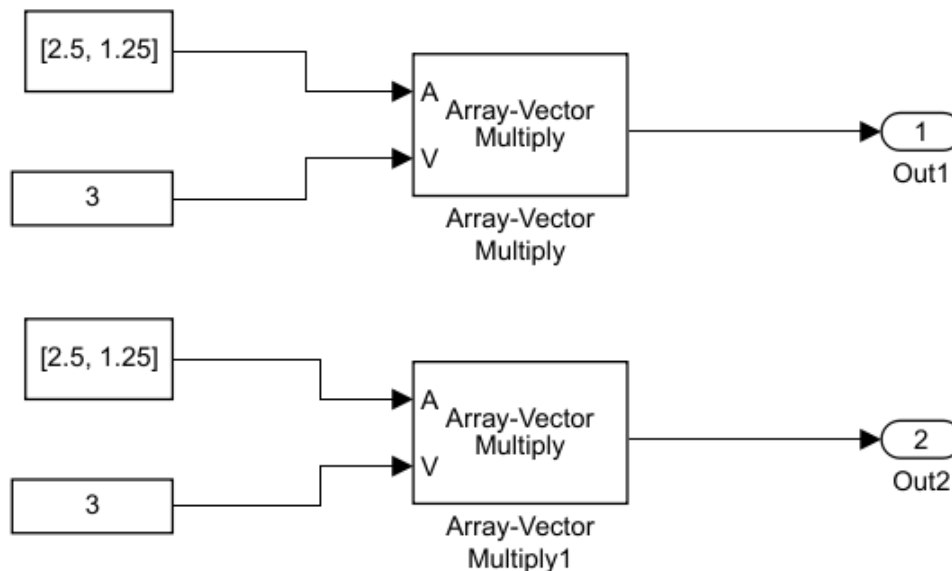


As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 10, 17, and 128 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case. You can see this 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:

$$WL_{idealproduct} = WL_{inputa} + WL_{inputb}$$

$$WL_{idealproduct} = 7 + 5 = 12$$

$$WL_{idealproduct1} = WL_{inputa} + WL_{inputb}$$

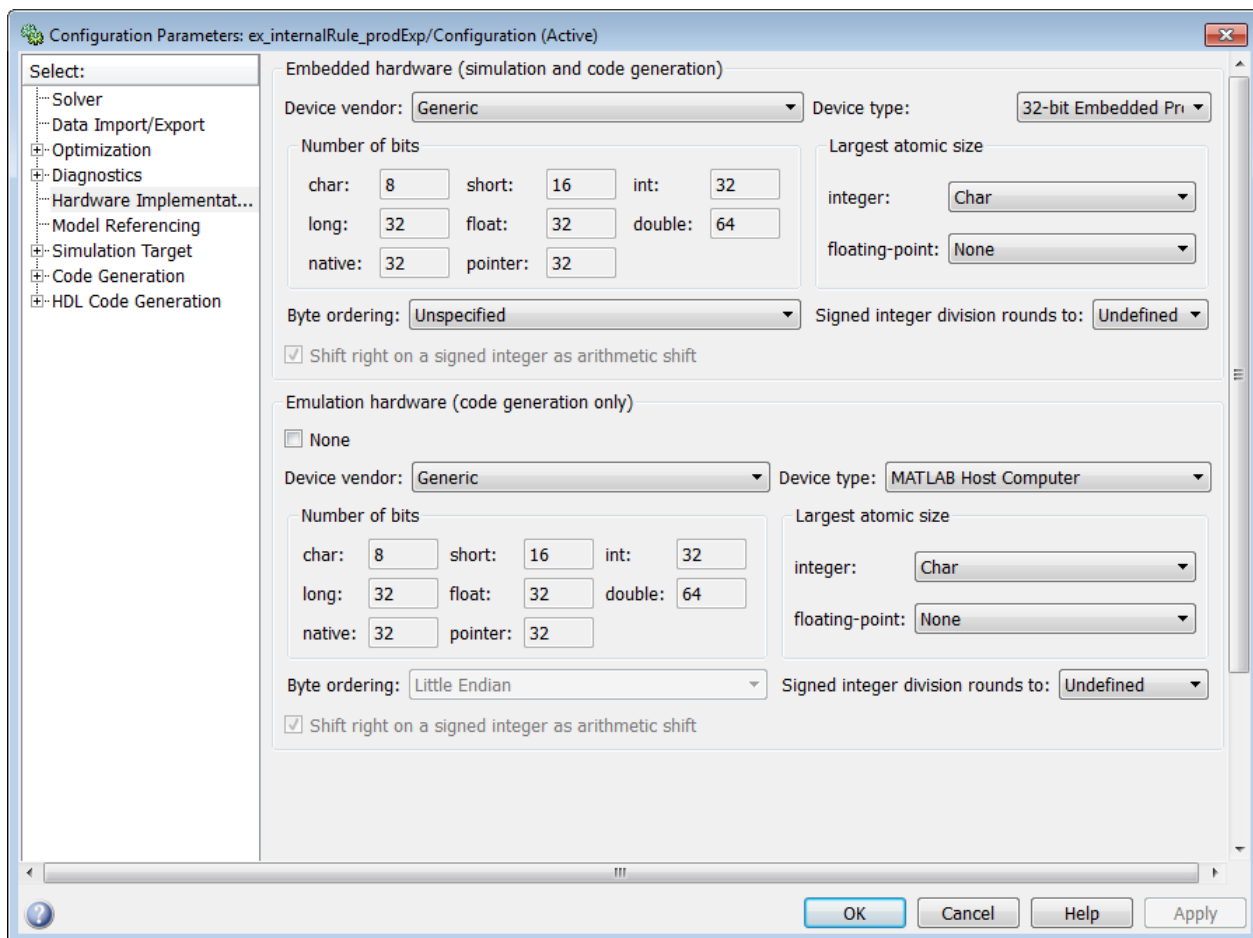
$$WL_{idealproduct1} = 16 + 15 = 31$$

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

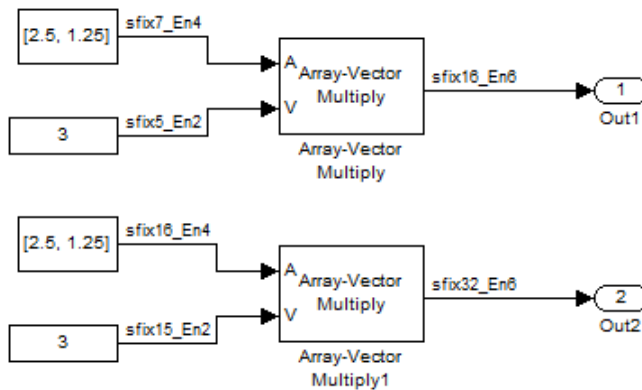
$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$

$$FL_{idealaccumulator} = 4$$

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.



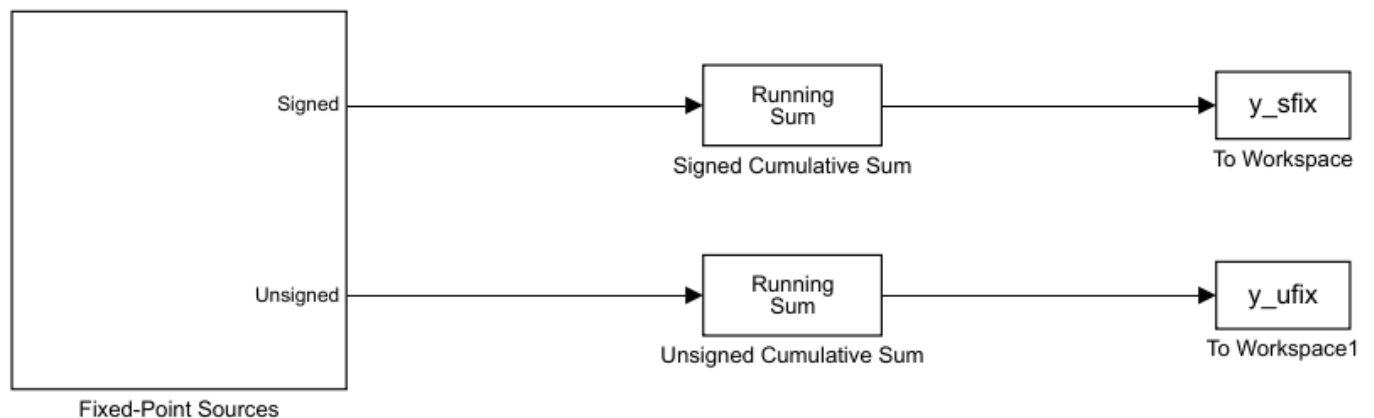
Specify Data Types for Fixed-Point Blocks

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 19-26
- “Use Data Type Override to Find a Floating-Point Benchmark” on page 19-30
- “Use the Fixed-Point Tool to Propose Fraction Lengths” on page 19-30
- “Examine the Results and Accept the Proposed Scaling” on page 19-31

Prepare the Model

- 1 Open the model by typing `ex_fixedpoint_tut` at the MATLAB command line.



Copyright 2009-2010 The MathWorks, Inc.

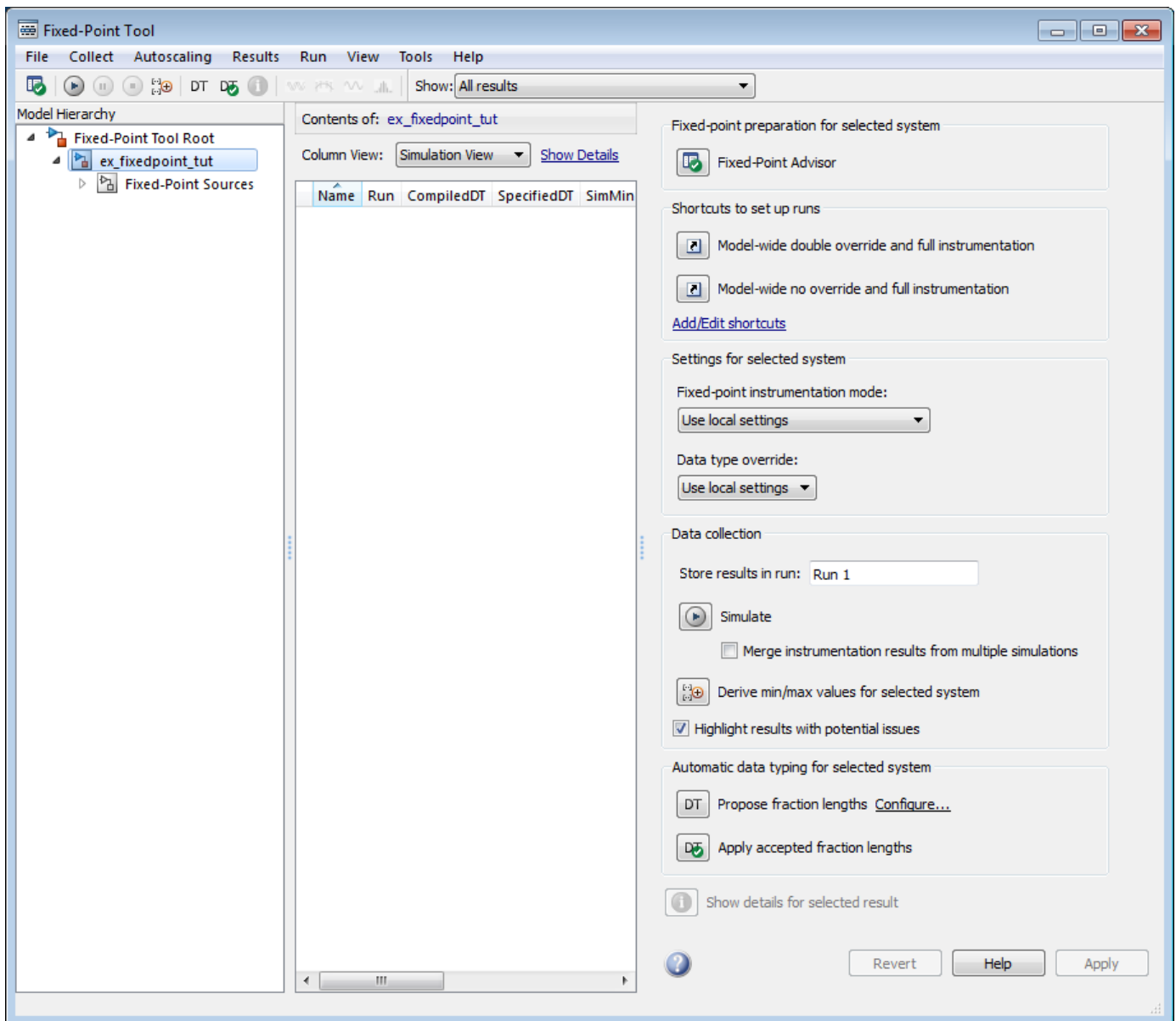
This model uses the Cumulative Sum block to sum the input coming from the Fixed-Point Sources subsystem. The Fixed-Point Sources subsystem outputs two signals with different data types:

- 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.




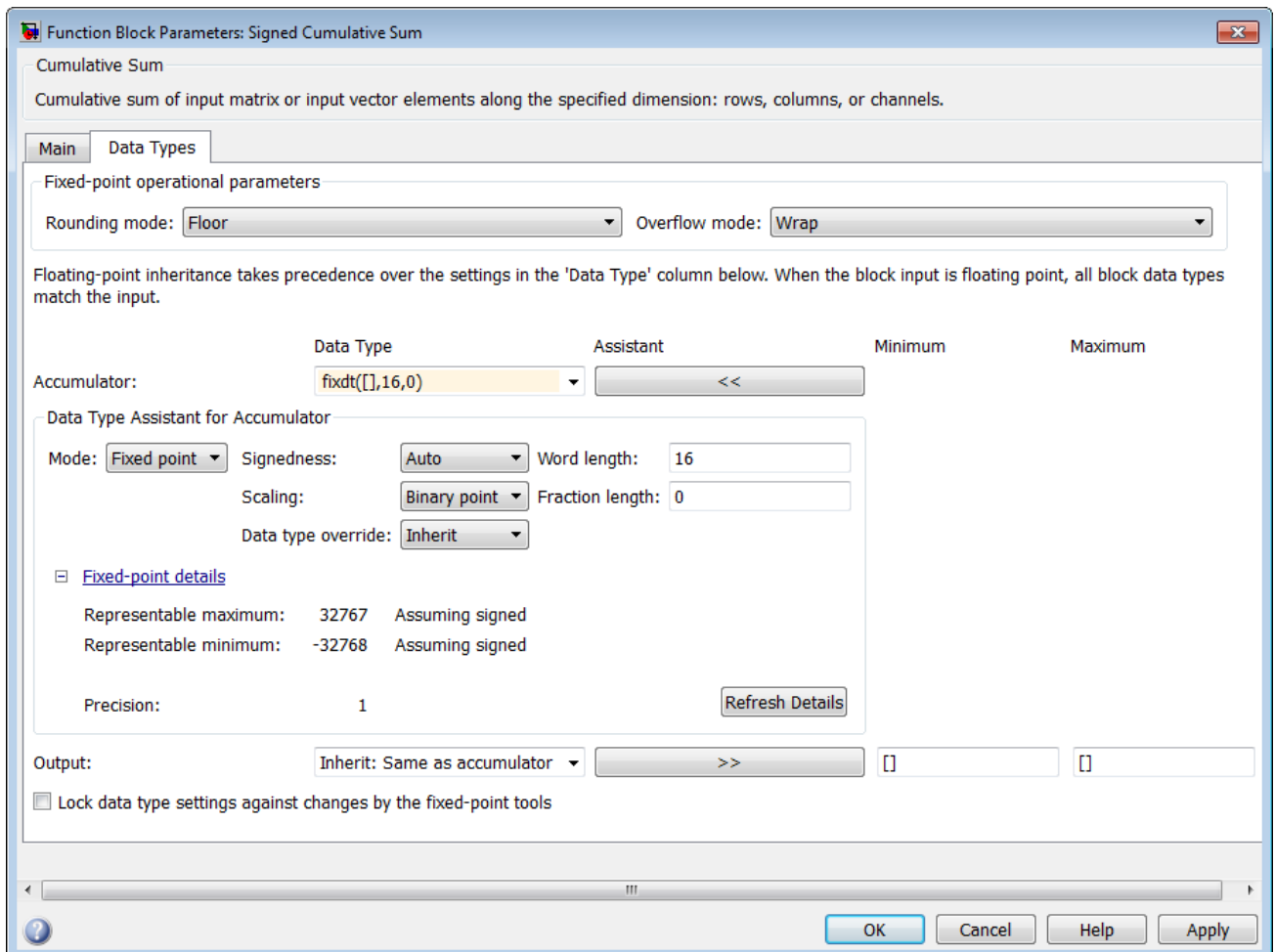
- 5 The results of the simulation appear in a table in the central **Contents** pane of the Fixed-Point Tool. Review the following columns:
- **Name** — Provides the name of each signal in the following format: Subsystem Name/Block Name: Signal Name.
 - **SimDT** — The simulation data type of each logged 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.

You can also see that the **SimMin** and **SimMax** values for the Accumulator data types range from 0 to .9997. The logged results indicate that 8,192 overflows wrapped during simulation in the Accumulator data type of the Signed Cumulative Sum block. Similarly, the Accumulator data type of the Unsigned Cumulative Sum block had 16,383 overflows wrap during simulation.

To get more information about each of these data types, highlight them in the **Contents** pane,

and click the **Show details for selected result** button ()

- 6 Assume a target hardware that supports 32-bit integers, and set the Accumulator word length in both Cumulative Sum blocks to 32. To do so, perform the following steps:
 - 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 “Specify 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)`).

Code Generation and Shared Library

- “Simulink Shared Library Dependencies” on page 20-2
- “Accelerating Simulink Models” on page 20-3
- “Portable C Code Generation for Functions That Use OpenCV Library” on page 20-4

Simulink Shared Library Dependencies

In general, the code you generate from Computer Vision 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 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 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:

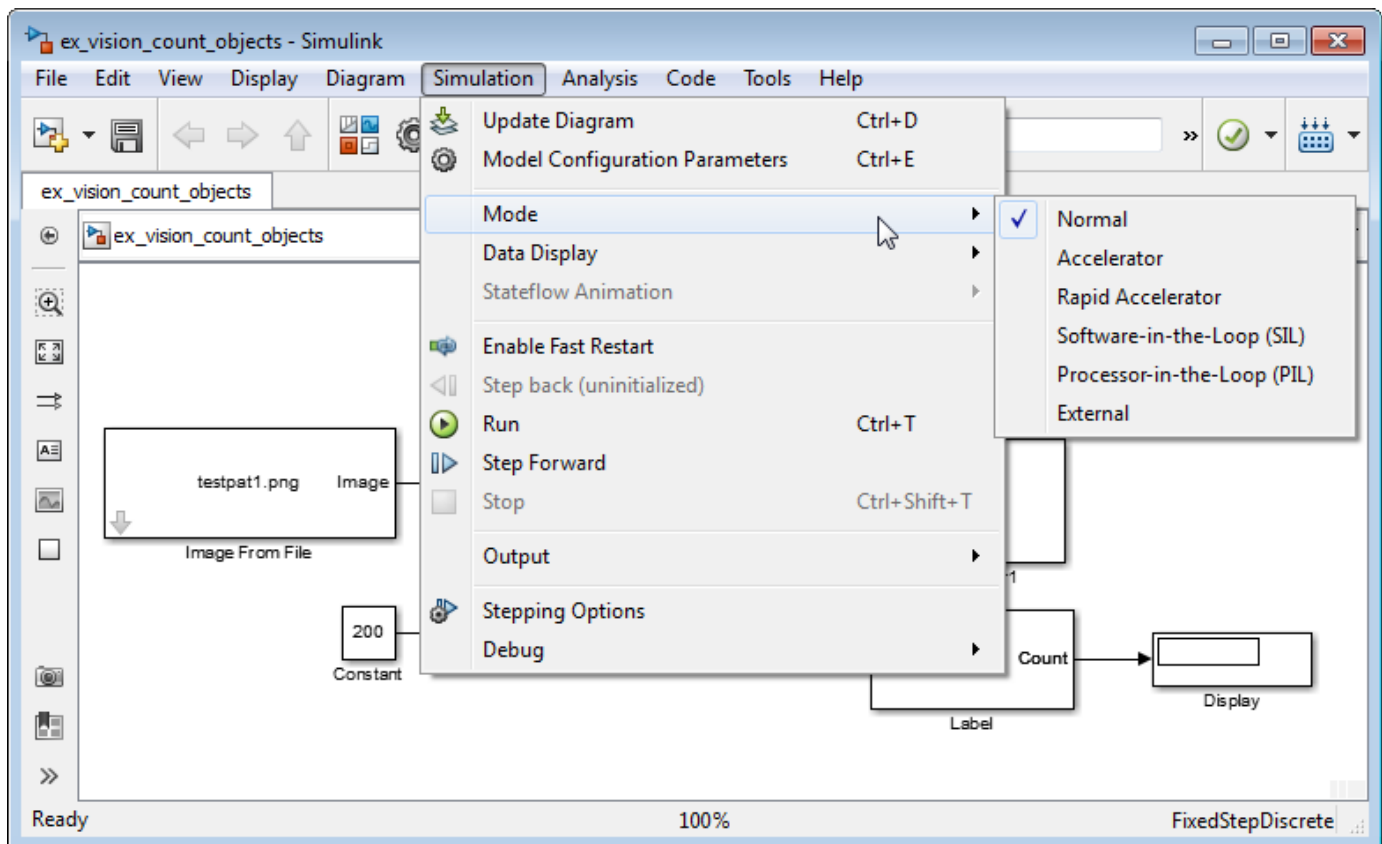
Host computer only. Excludes Simulink Desktop Real-Time™ target.

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

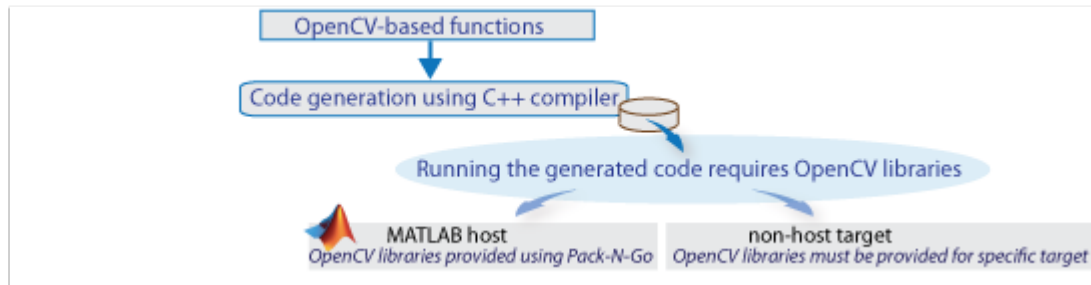
The Simulink software offer Accelerator and Rapid Accelerator simulation modes that remove much of the computational overhead required by Simulink models. These modes compile target code of your model. Through this method, the Simulink environment can achieve substantial performance improvements for larger models. The performance gains are tied to the size and complexity of your model. Therefore, large models that contain Computer Vision 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 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 or Visual Studio 2017
Linux 64 bit	gcc-4.9.3 (g++)
Mac 64 bit	Xcode 6.2.0 (Clang++)

Limitations

Computer Vision Toolbox functions that use the OpenCV library do not support target code generation from Simulink.

Vision Blocks Examples

- “Generate Image Histogram” on page 21-2
- “Export Image to MATLAB Workspace” on page 21-4
- “Import Video from MATLAB Workspace” on page 21-7
- “Find Minimum Value in ROI” on page 21-9
- “Write Image to Binary File” on page 21-13
- “Compute Standard Deviation of ROIs” on page 21-14
- “Read Video Stored as Binary Data” on page 21-17
- “Compare Image Quality Using PSNR” on page 21-21
- “Compute Autocorrelation of Input Matrix” on page 21-23
- “Compute Correlation between Two Matrices” on page 21-24
- “Find Statistics of Circular Blobs in Image” on page 21-25
- “Replace Intensity Values in ROI with its Maximum Value” on page 21-29
- “Median based Image Thresholding” on page 21-33
- “Import Image From MATLAB Workspace” on page 21-36
- “Import Image from Specified Location” on page 21-38
- “Remove Interlacing Effect From Image” on page 21-42
- “Estimate Motion between Two Images” on page 21-45

Generate Image Histogram

This example shows how to generate the histogram of an image using 2-D Histogram block. The model outputs a bar plot that shows the frequency of occurrence for pixels values in the input image.

Read an input image to the MATLAB workspace.

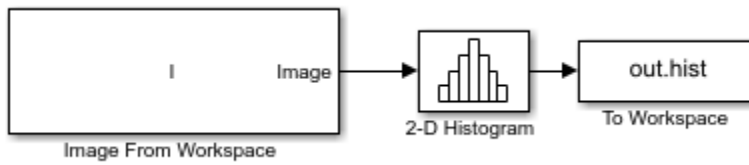
```
I = imread('cameraman.tif');
```

Find the maximum intensity value in the input image.

```
maxI = max(I(:));
```

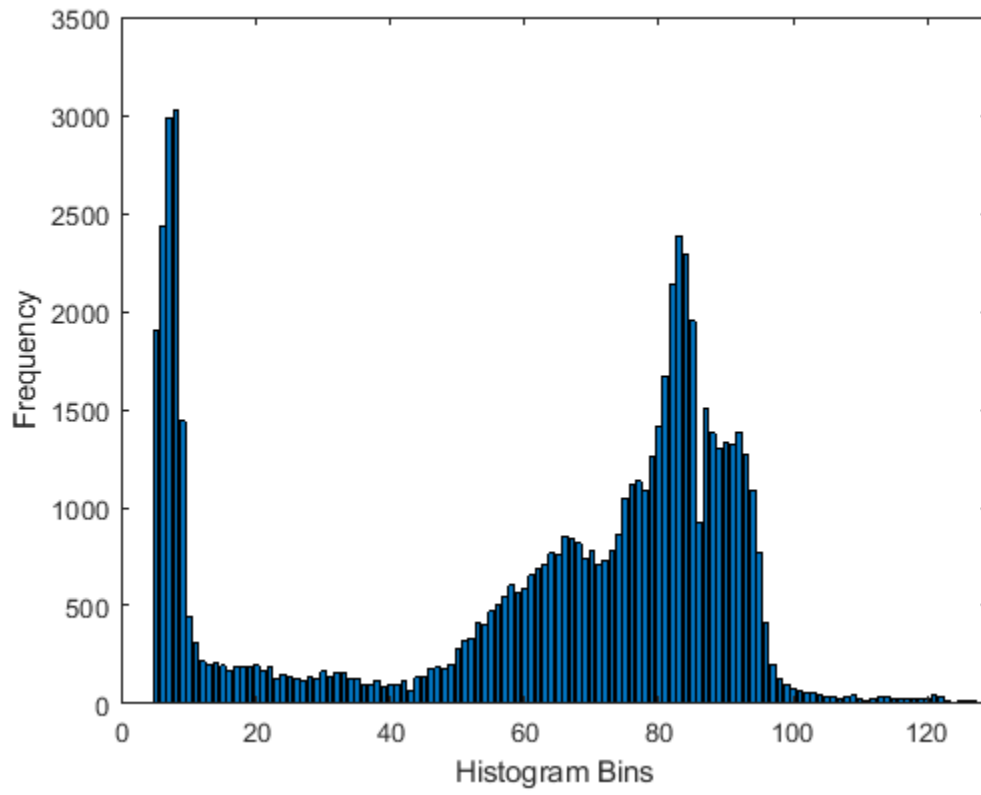
Open the simulink model. The model reads the image stored in variable I from the MATLAB workspace. The Upper limit of histogram parameter of the 2-D Histogram block is set to the maximum value of the intensity image. The Number of Bins parameter of the 2-D Histogram block is set to 128 and the histogram is computed for the entire input.

```
modelName = 'ex_blkhistogram.slx';
open_system(modelname);
```



The model outputs a time series that specifies the frequency of occurrence of pixels within each bin. Export the histogram values to MATLAB workspace and plot the histogram.

```
out = sim(modelname);
bar(out.hist.data)
xlabel('Histogram Bins')
ylabel('Frequency')
```

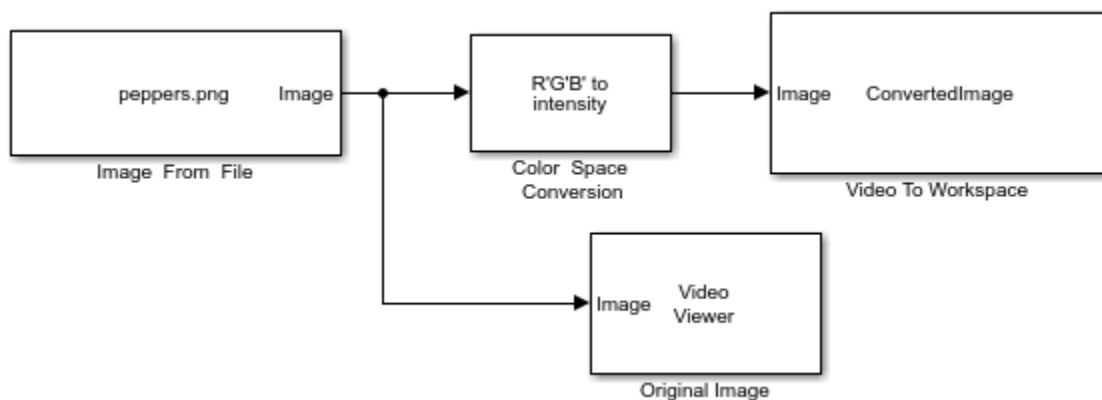
Export Image to MATLAB Workspace

This model shows how to export an image from Simulink to MATLAB workspace by using the Video To Workspace block.

Example Model

This model takes a color image as the input, converts it into a gray scale image and exports the converted image to MATLAB workspace.

```
modelName='ex_blkvideotoworkspace.slx';  
open_system(modelname);
```

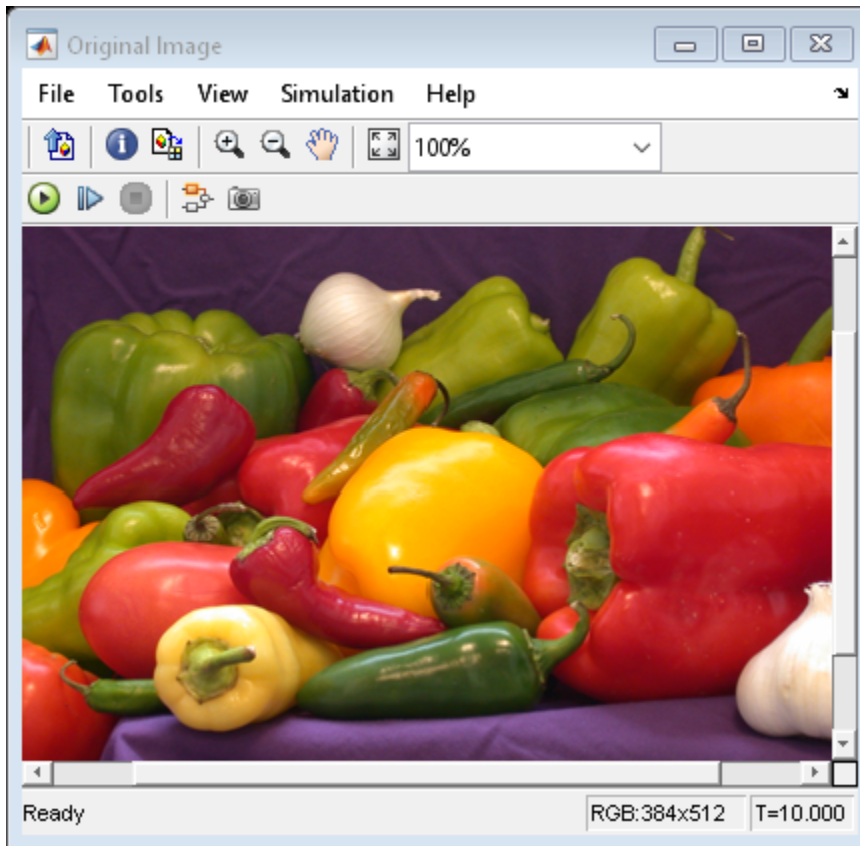


To convert the original image into gray scale, set the Conversion parameter of Color Space Conversion block to R'G'B to intensity. The original image is of size 384-by-512-by-3 and the gray scale image output from the Color Space Conversion block is of size 384-by-512.

Export the converted image to MATLAB workspace as a variable named ConvertedImage using the Video To Workspace block. You can display the original image using the Video Viewer block.

Simulate and Display Results

```
sim(modelname);
```



The Video To Workspace block exports the converted image as a video with two identical frames and is of size 384-by-512-by-2. Use the `imshow` function to display the first frame in the video.

```
imshow(ConvertedImage(:,:,1))
```



Import Video from MATLAB Workspace

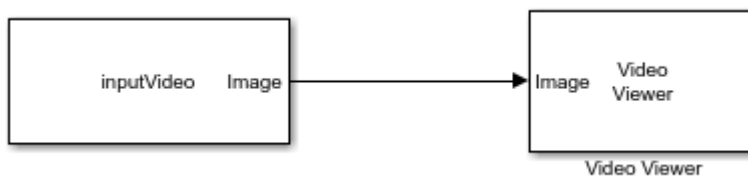
This example shows how to import a video from MATLAB to Simulink workspace by using Video From Workspace block.

Load the video data to MATLAB workspace.

```
load('videosignal.mat')
```

Open the Simulink model.

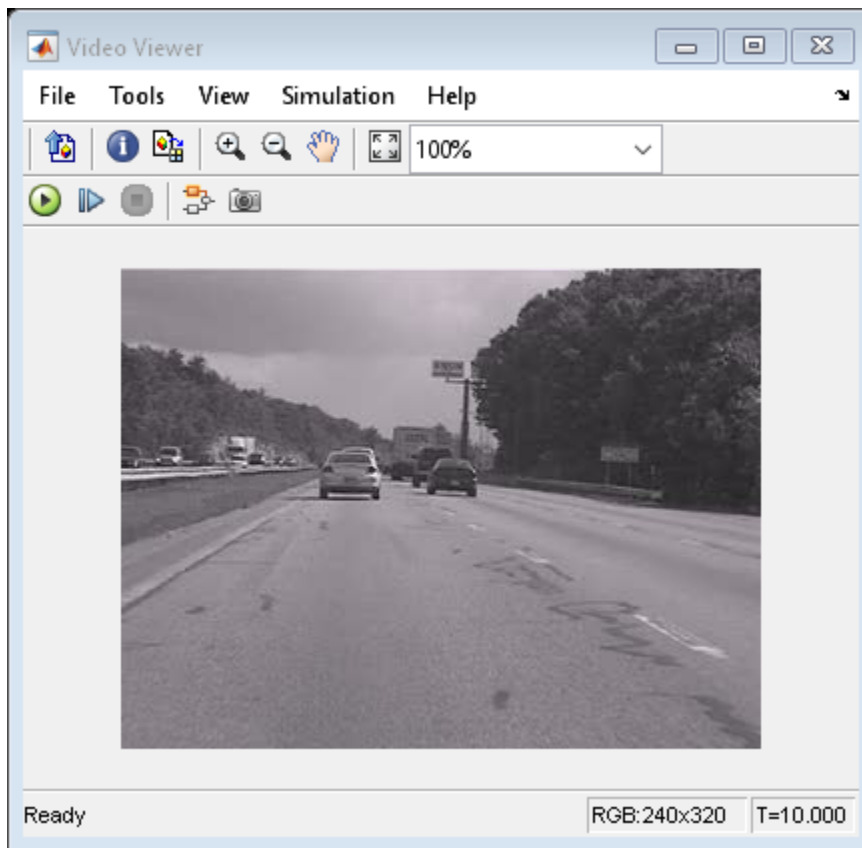
```
modelName='ex_blkvideofromworkspace.slx';  
open_system(modelname);
```



The model reads the video data from MATLAB workspace. Set the Form output after final value by parameter of the Video From Workspace block to Holding final value. This parameter setting repeats the last frame of the video after generating all the frames.

Simulate the model and display the imported video signal by using Video Viewer block.

```
sim(modelname);
```



Find Minimum Value in ROI

This example shows how to calculate the minimum value in an image ROI by using the 2-D Minimum block. By using the minimum value, the model removes indistinct pixels in the image regions.

Example Model

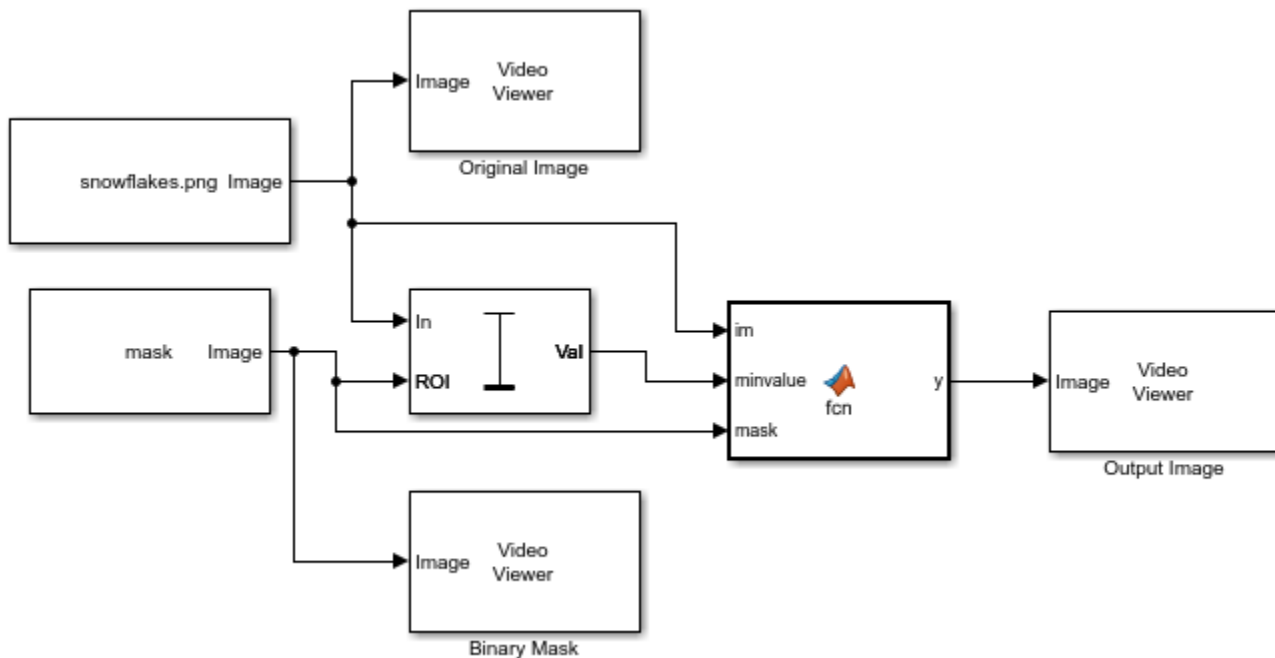
The model reads the original image and the binary mask comprising the ROI for which the minimum value has to be computed. The original image consists of regions with large image structures that are circular in shape and regions with small, indistinct image structures. The binary mask isolates the indistinct structures from the distinct image structures.

Load the binary mask containing the ROI to MATLAB workspace. The ROI corresponding to indistinct structures have intensity value 1 in the binary mask.

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

Open the model.

```
modelName='ex_blk2dminimum';
open_system(modelname);
```



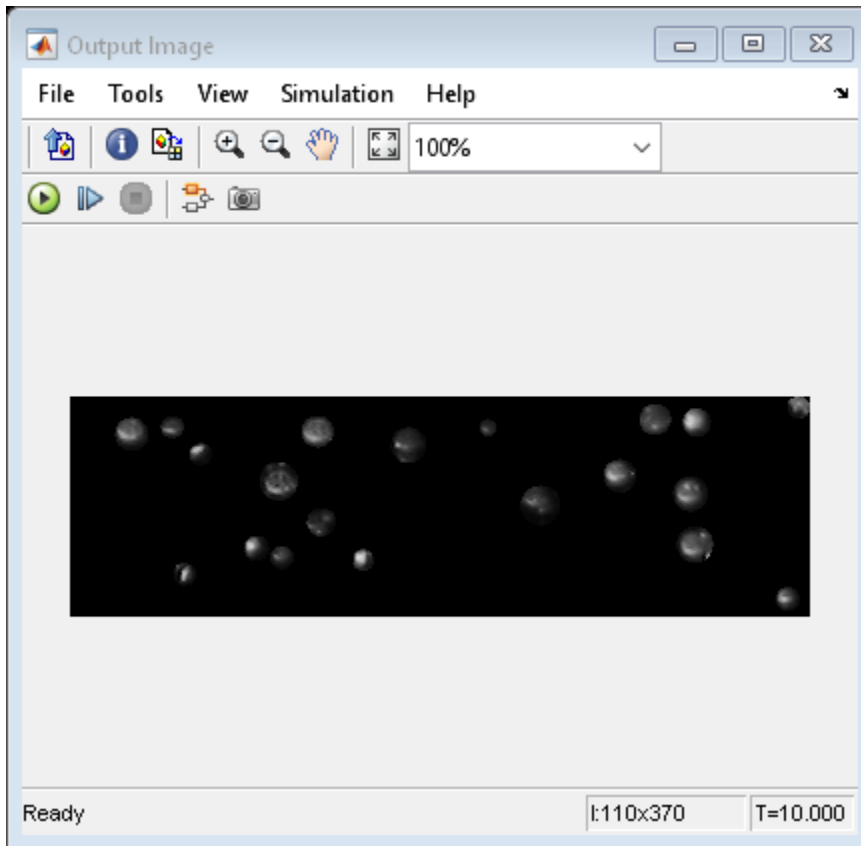
For the 2-D Minimum block to output only the computed minimum, the Mode parameter of the block is set to Value. To perform ROI processing, the Find the minimum value over parameter is set to Entire input. The ROI input to the 2-D Minimum block is a binary image. Hence, the ROI Type parameter is set to Binary mask. The block computes the minimum value of the pixels in the original image that lie in the ROI specified by the binary mask.

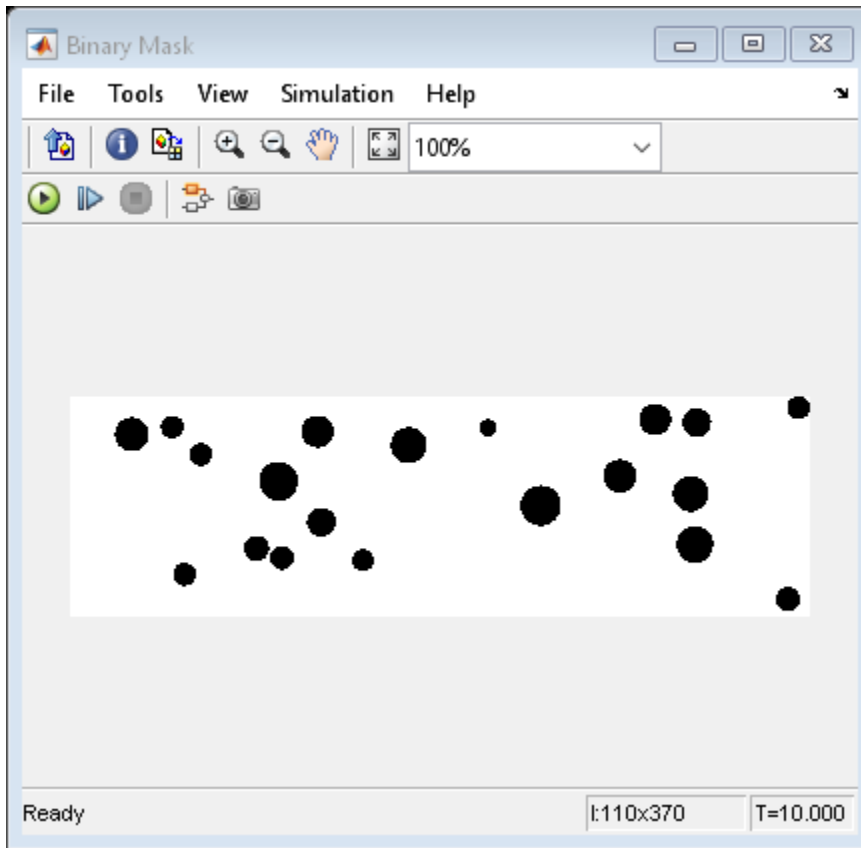
The MATLAB function block replaces pixel values in the ROI with the computed minimum.

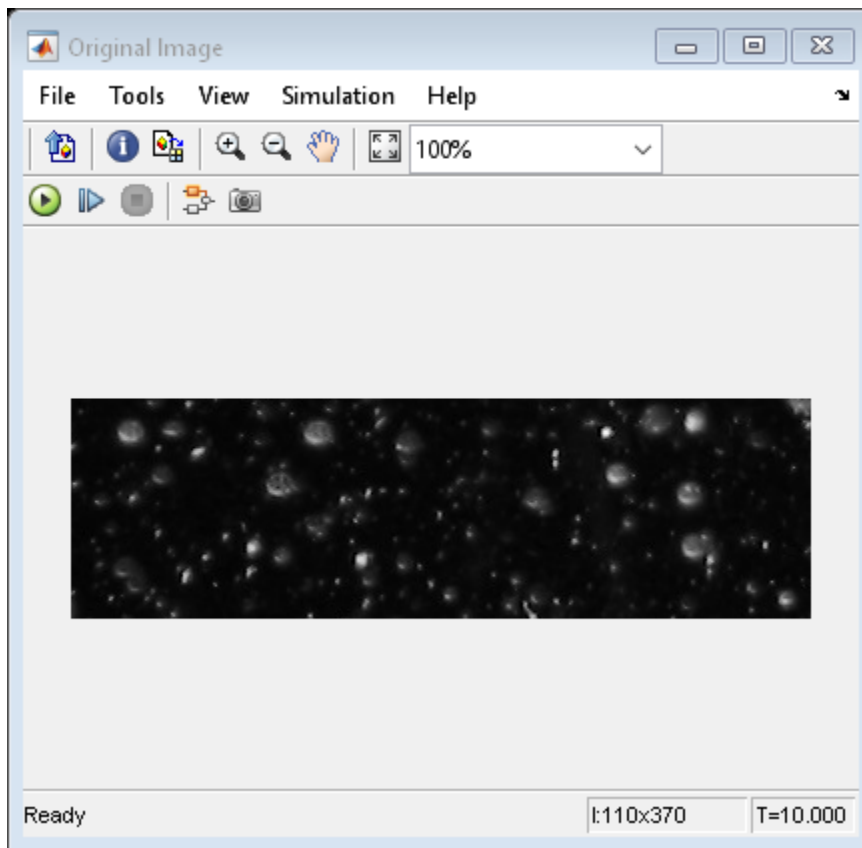
Simulate and Display Results

The model outputs a clean image with only distinct image structures.

```
sim(modelname);
```





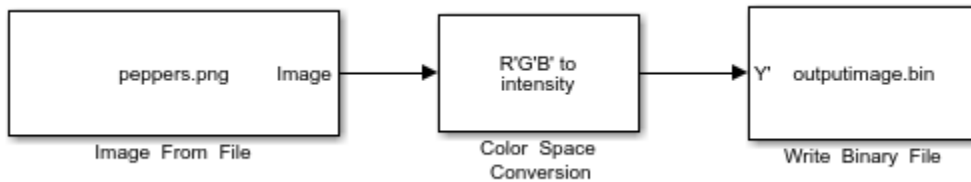


Write Image to Binary File

This example shows how to write an image data to a binary file in a custom format using the Write Binary File block.

Open the Simulink model.

```
modelname = 'ex_blkwritebinaryfile.slx';  
open_system(modelname);
```



The input to the model is a RGB color image. The model converts the color image to grayscale using the Color Space Conversion block. The Conversion parameter of Color Space Conversion block is set to R'G'B' to intensity. The output binary file name is specified in the File name parameter of the Write Binary File block as outputimage.bin. The parameters of the Write Binary File block are configured so the block outputs a custom binary file.

- Video Format : Custom
- Number of inputs : 1
- Component order in binary file : 1

Simulate the model.

```
sim(modelname);
```

The model outputs a binary file named outputimage.bin to the MATLAB workspace. You can read this binary file using the Read Binary File block.

Compute Standard Deviation of ROIs

This example shows how to compute the standard deviation of regions-of-interest (ROIs) in the input image. The input image is composed of different texture regions and ROIs are selected to contain these texture regions.

Read an image into the MATLAB workspace.

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

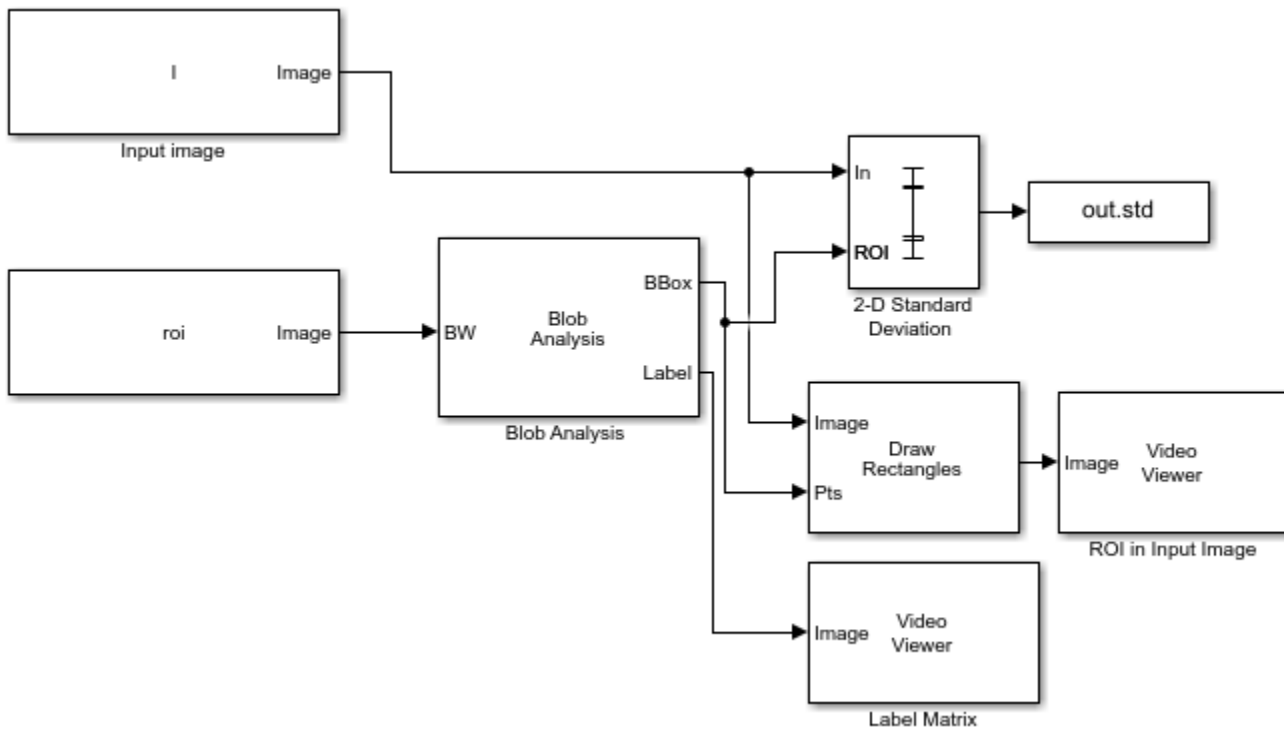
Load the mask image that specifies the ROIs in the input image.

```
load('binaryROI.mat')
```

Example Model

Open the Simulink model.

```
modelName='ex_blk2dstd.slx';
open_system(modelName);
```



The model computes the coordinates for the ROIs by using the **Blob Analysis** block. The maximum number of blobs parameter in the **Blob Analysis** block is set to 5, the number of ROIs.

The **2-D Standard Deviation** block computes the standard deviation value for each ROI.

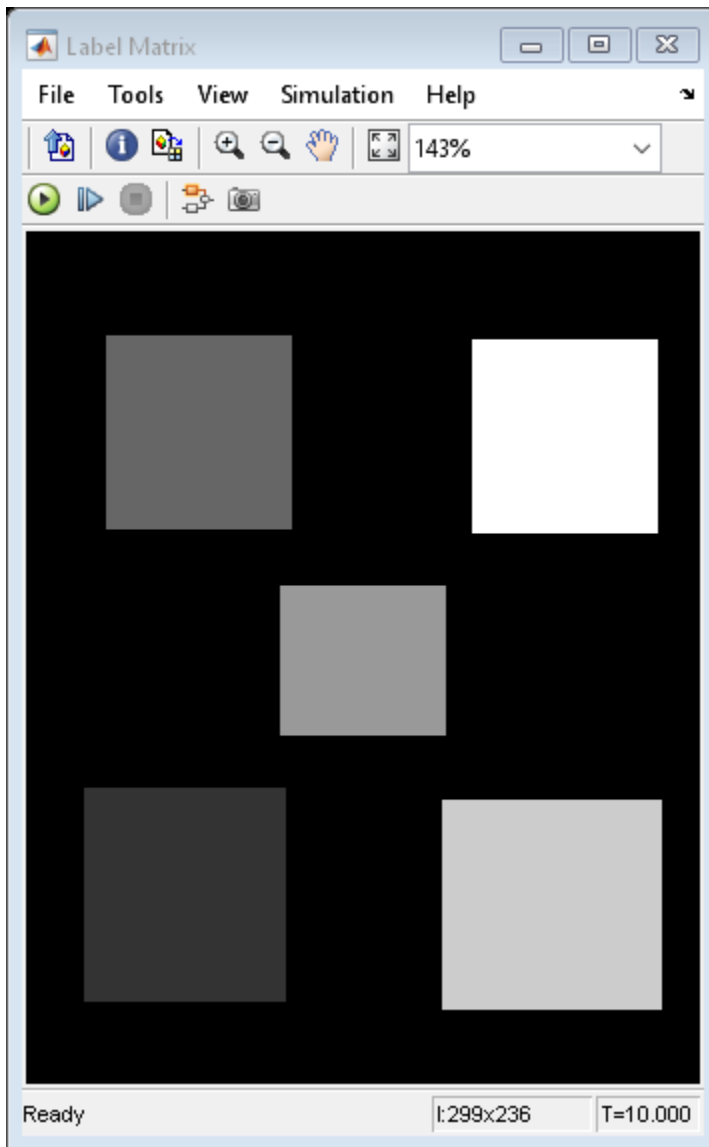
Set these parameters of the **2-D Standard Deviation** block to the specified value in order to compute individual statistics for each ROI.

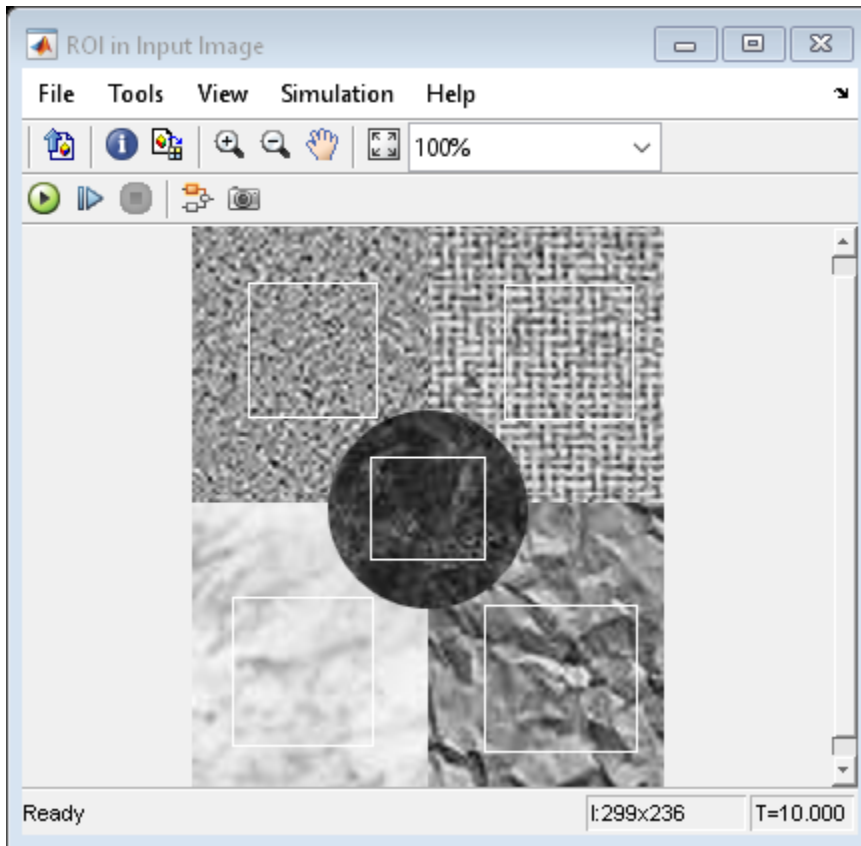
- Set Find the standard deviation value over parameter to Entire input
- Select Enable ROI processing parameter
- Set ROI type parameter to Rectangles
- Set Output parameter to Individual statistics for each ROI

Simulate and Display Results

The values of the standard deviation indicate the dispersion of the pixel values in ROI from the corresponding mean value.

```
out = sim(modelname);
```





The model also displays the input image and the label matrix that correspond to the selected ROIs. The rectangles overlaid on the input image represents the ROIs for which the standard deviation is computed.

Display the standard deviation value for each ROI. The first standard deviation value correspond to the region with label value 1. Similarly, the second standard deviation value correspond to the region with label value 2 and so on.

```
out.std
```

```
ans =
```

```
0.0534
0.1203
0.0775
0.1463
0.1629
```

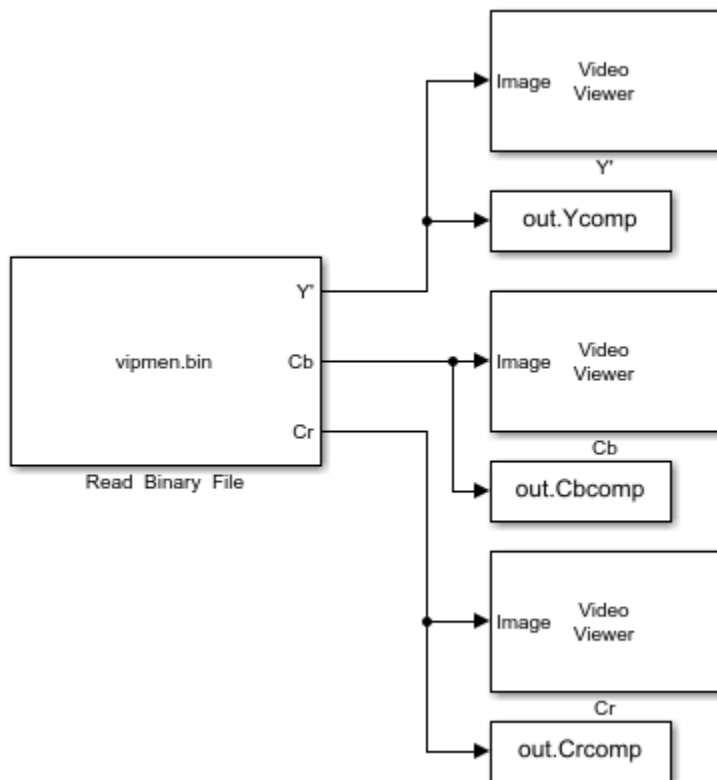
Read Video Stored as Binary Data

This example shows how to read a video data stored in binary format by using the Read Binary File block.

Example Model

Open the Simulink model.

```
modelName = 'ex_blkreadbinaryfile.slx';
open_system(modelname);
```

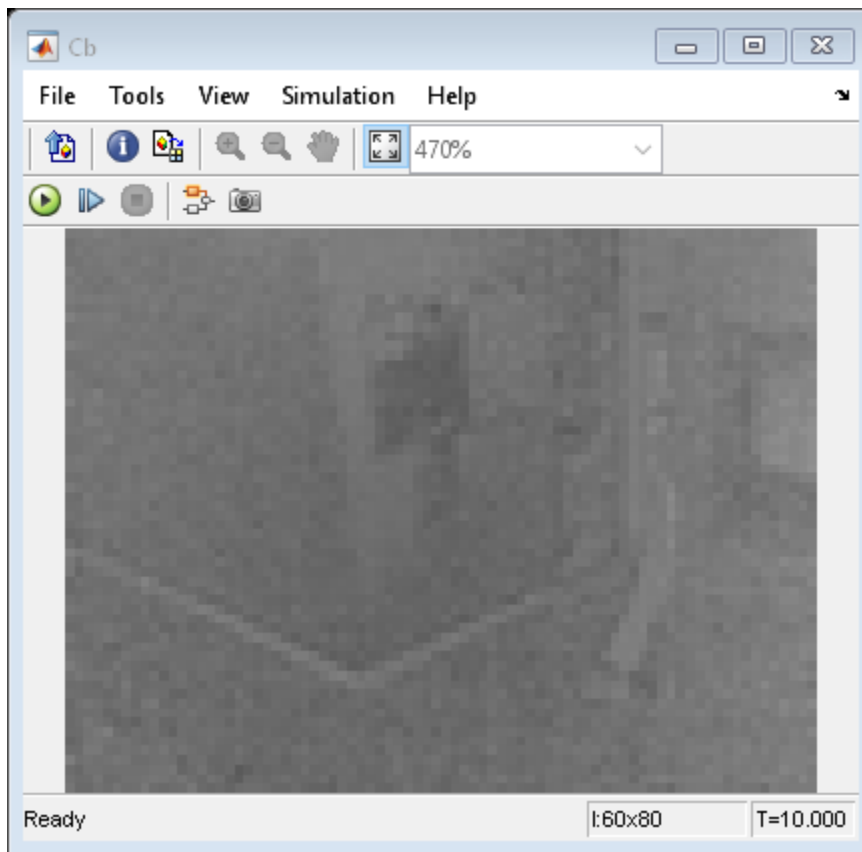


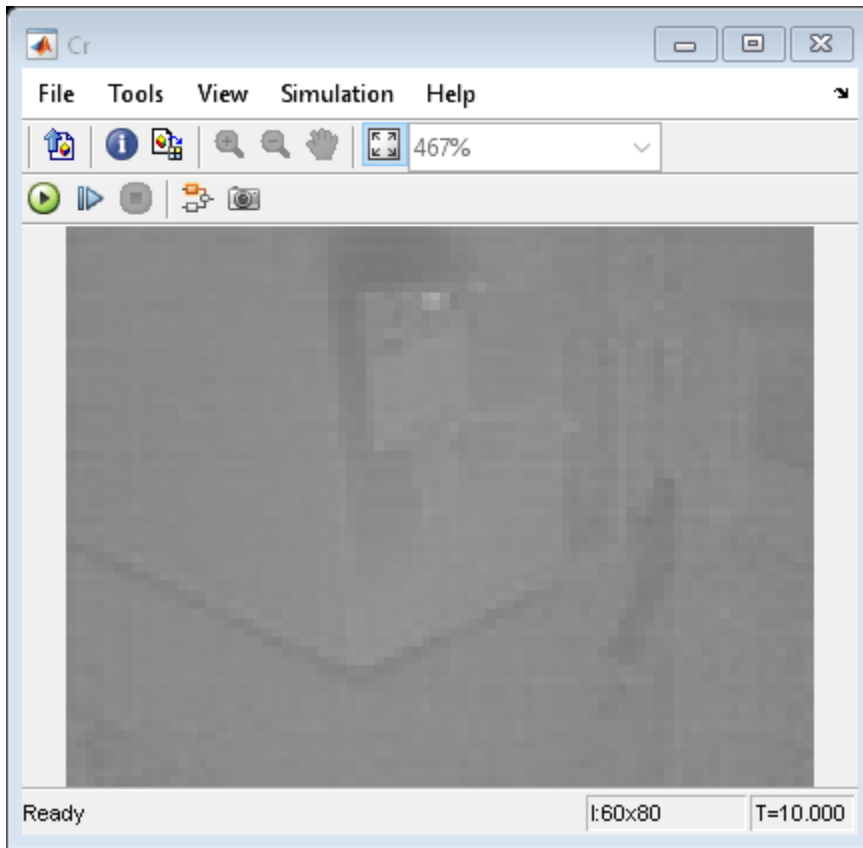
The model reads the binary file specified as 'vipmen.bin' in the File name parameter of the Read Binary File block. The file is played until the end of the simulation because the Number of times to play file parameter of the Read Binary File block is set to inf.

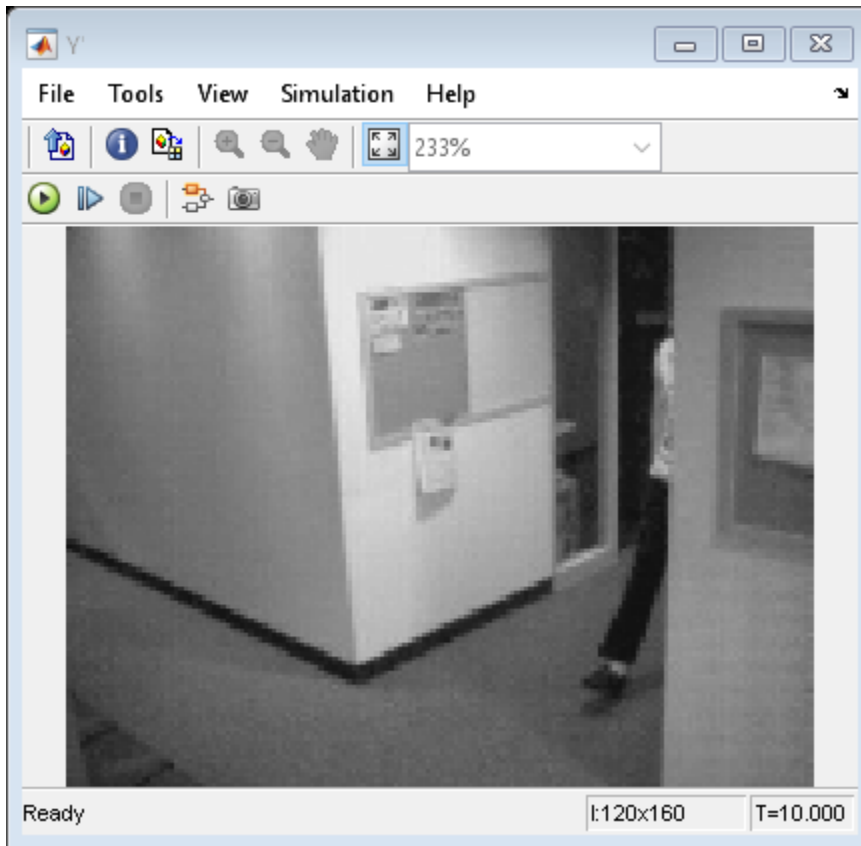
Simulate and Display Results

Run the simulation.

```
sim(modelname);
```







The model outputs the luminance, blue difference, and red difference components of the input video stored as binary data. The Read Binary File block exports the binary data to the MATLAB workspace as video data with a frame size of 120-by-160 pixels. The Video Viewer blocks display the components of the binary data.

Compare Image Quality Using PSNR

This example shows how to compare the quality of a noisy and denoised image from the PSNR value computed using the PSNR block.

Read an image into the MATLAB workspace.

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

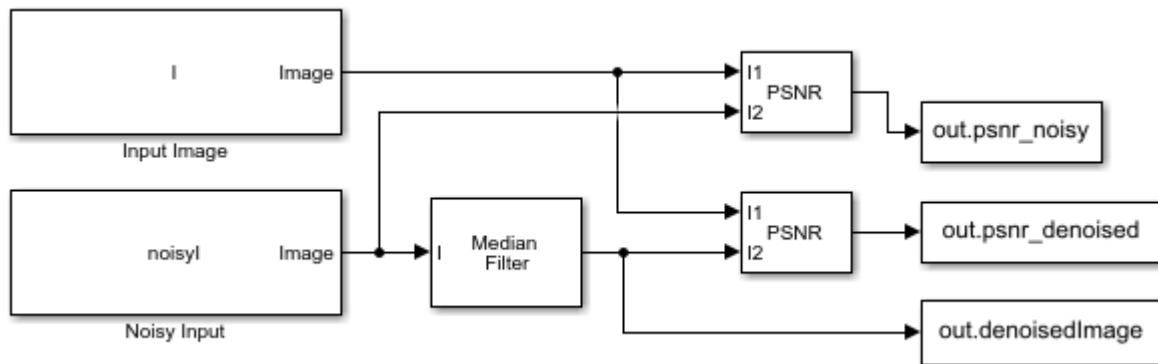
Read the corresponding noisy image into the MATLAB workspace.

```
noisyI = imread('noisyCameraman.tif');
```

Example Model

Open the Simulink model. The model reads the original and the noisy images from the MATLAB workspace and denoises the noisy image by using the Median Filter block.

```
modelName='ex_blkpsnr.slx';
open_system(modelname);
```



The model computes the PSNR value for the noisy and the denoised image with respect to the original image and outputs as variables named `psnr_noisy` and `psnr_denoised` respectively. The denoised image and the computed PSNR values are exported to the MATLAB workspace.

Simulate and Display Results

Simulate the model.

```
out = sim(modelname);
```

Display the noisy image and the corresponding PSNR value

```
imshow(noisyI,[]);
title(['PSNR = ', num2str(out.psnr_noisy)]);
```

PSNR = 22.0255



Display the denoised image and the corresponding PSNR value. The denoised image is of better perceptual quality than the noisy image and hence, has comparatively high PSNR value.

```
imshow(out.denoisedImage, []);  
title(['PSNR = ', num2str(out.psnr_denoised)]);
```

PSNR = 27.1628



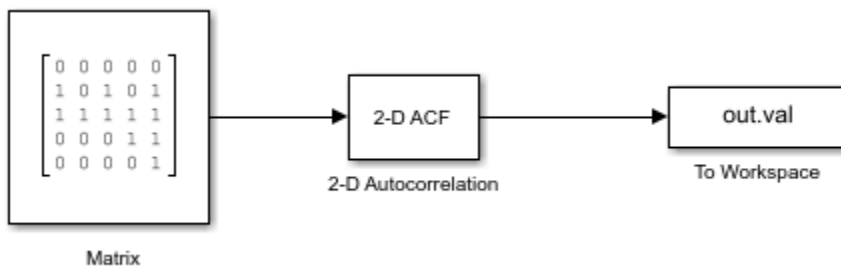
Compute Autocorrelation of Input Matrix

This example shows how to compute autocorrelation of a 5-by-5 input matrix using the 2-D Autocorrelation block.

The output of the model is a 9-by-9 matrix consisting of autocorrelation coefficients.

The coefficient values shows the similarity between the input matrix and its shifted form. The value of the autocorrelation coefficient at a point (i, j) is high, if the values in the original matrix and the shifted matrix are similar.

```
model = 'ex_blkautocorrelation.slx';
open_system(model)
```



Run the model and export the computed autocorrelation coefficients to MATLAB workspace. Display the coefficients using `disp` function.

```
out = sim(model);
disp(out.val)
```

```
0    0    0    0    0    0    0    0    0
1    0    1    0    1    0    0    0    0
2    2    2    2    2    1    0    0    0
2    3    4    5    6    3    2    1    1
2    2    5    5    11   5    5    2    2
1    1    2    3    6    5    4    3    2
0    0    0    1    2    2    2    2    2
0    0    0    0    1    0    1    0    1
0    0    0    0    0    0    0    0    0
```

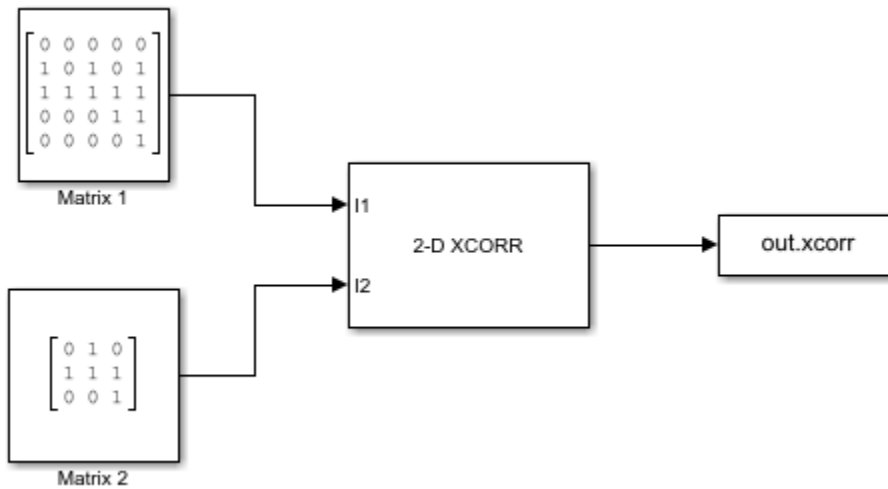
Compute Correlation between Two Matrices

This example shows how to compute correlation between two matrices using 2-D XCORR block.

Example Model

Open the simulink model.

```
open_system('ex_blkxcorr.slx');
```



The model consists of 5-by-5 and 3-by-3 matrices as inputs. To return correlation values that are computed without zero-padding, the **Output size parameter** is set to `Valid`. The range for output correlation value is set to `[0 1]` by enabling the **Normalized output parameter**.

Simulate and Display Results

Run the model and display the output value. The output of the model is an array of correlation coefficients. The correlation value signifies the similarity between the values of the input matrices within a chosen window. The correlation coefficient is high (1) when both the input matrices have similar values within a window.

```
A = sim('ex_blkxcorr.slx');
disp(A.xcorr);
```

```
0.6000    0.4472    0.6000
0.6000    1.0000    0.6761
0.2582    0.4472    0.7303
```

Find Statistics of Circular Blobs in Image

This example shows how to find the centroid, perimeter, and bounding box coordinates of circular blobs in an image by using the Blob Analysis block. The model also outputs the label matrix for reference.

Load Data To MATLAB Workspace

Read an image into the MATLAB workspace.

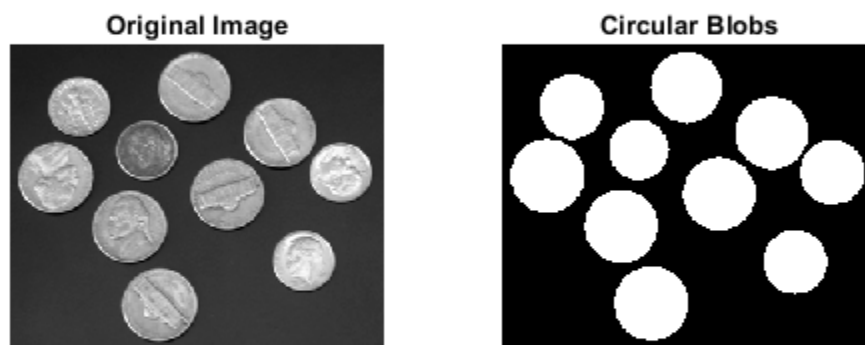
```
I = imread('coins.png');
```

Load a binary mask containing the blobs that represent the segmented objects in the input image.

```
load('maskImage', 'BW');
```

Display the input image and the corresponding binary mask. The binary mask consists of 10 circular blobs of varied statistics.

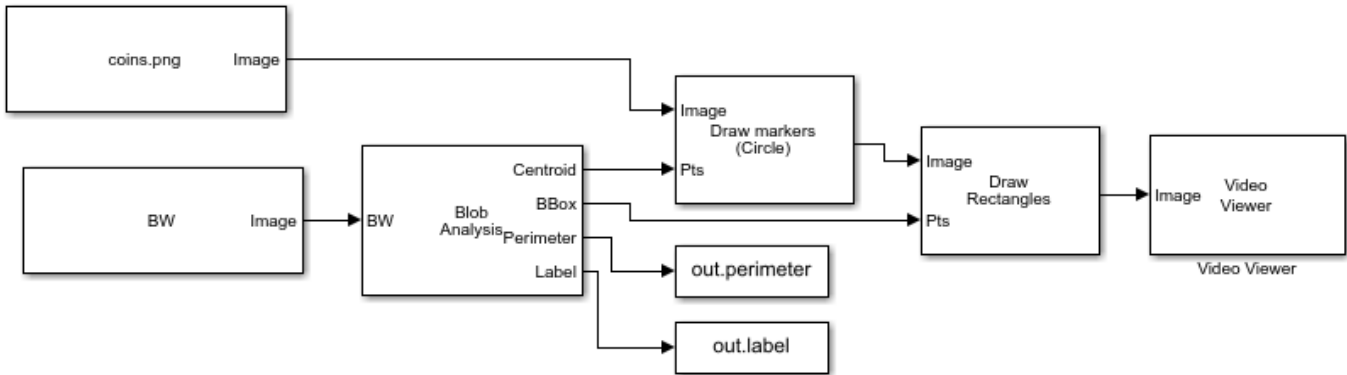
```
figure
subplot(1,2,1)
imshow(I,[]);
title('Original Image');
subplot(1,2,2)
imshow(BW)
title('Circular Blobs');
```



Example Model

Open the simulink model.

```
open_system('ex_blkblobanalysis.slx')
```



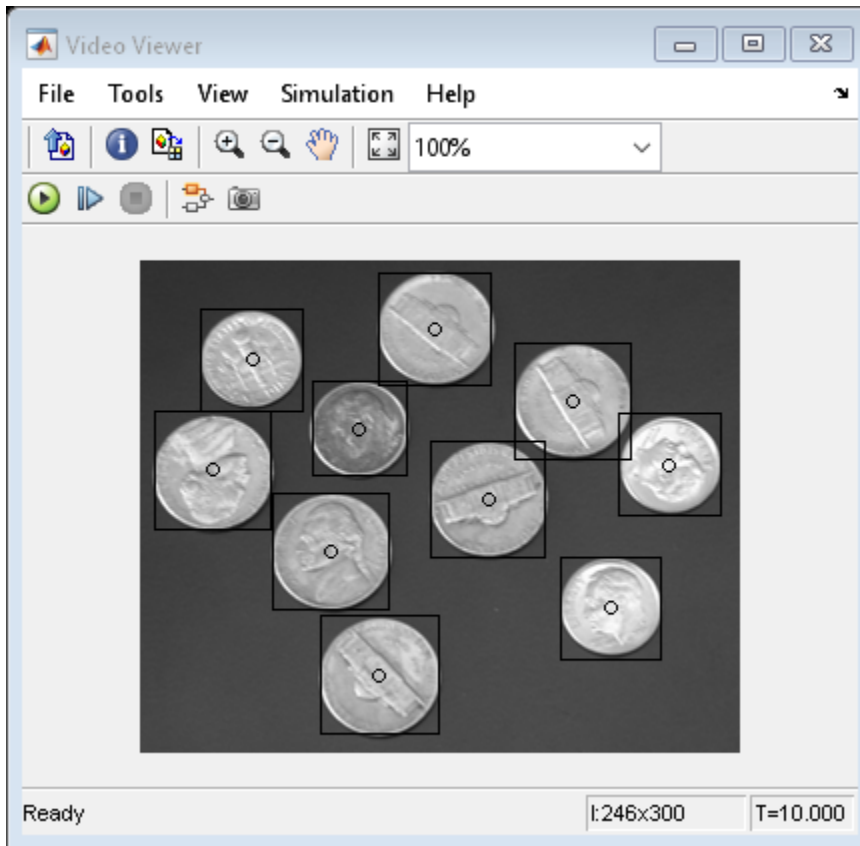
The model computes the centroid, perimeter, and bounding box coordinates for the blobs in the binary image. The computed statistics is overlaid on the input image by using the Draw Markers and Draw Rectangles blocks.

The number of output blobs parameter of Blob Analysis block is set equal to the number of blobs in the binary mask. The Draw Markers block plots the computed centroids and the Draw Rectangles block draws the bounding boxes. The perimeter values are exported as variable `perimeter` to the MATLAB workspace. The label matrix is exported as variable `label` to the MATLAB workspace.

Simulate and Display Results

Run the model and display the results using the Video Viewer block. The computed centroid and the bounding box are overlaid on the original image. The circular markers specifies the centroid of each blob and the rectangles around each blob specifies the computed bounding boxes.

```
out = sim('ex_blkblobanalysis.slx');
```

The first value in all the computed statistics correspond to the blob with label value 1. Similarly, the second values correspond to the blob with label value 2 and so on. The label value 0 corresponding to the background of the mask must be ignored.

Read the unique label values from the label matrix.

```
lb = unique(out.label);
```

Display the perimeter values and the corresponding label values as a table.

```
table(lb(2:end),out.perimeter, 'VariableNames',{'Label','Perimeter'})
```

```
ans =
```

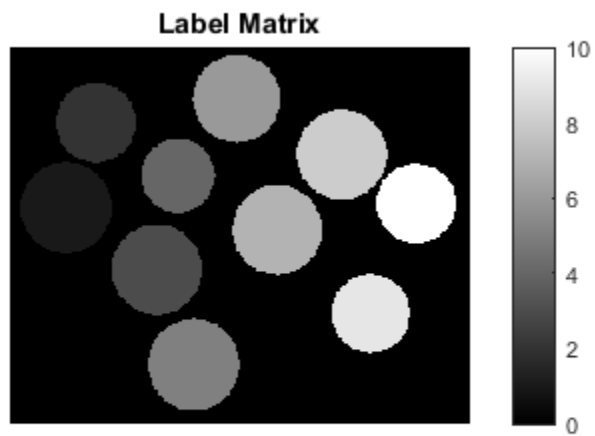
```
10x2 table
```

Label	Perimeter
1	194.17
2	170.02
3	191.58
4	156.37
5	195.58
6	186.51
7	190.75

```
8      192.17
9      167.44
10     168.85
```

Display the label matrix.

```
figure
imshow(out.label,[]);
colorbar
title('Label Matrix');
```



Replace Intensity Values in ROI with its Maximum Value

This example shows how to find maximum intensity value of region of interests (ROI) in the input image and replace the pixels in the ROI with its maximum value

Load Data to MATLAB Workspace

The input to the model is the original image, label matrix, and the label values. The label matrix contain the desired ROIs in the input image. Load the label matrix into MATLAB workspace.

```
load Snowflakes_mask.mat
```

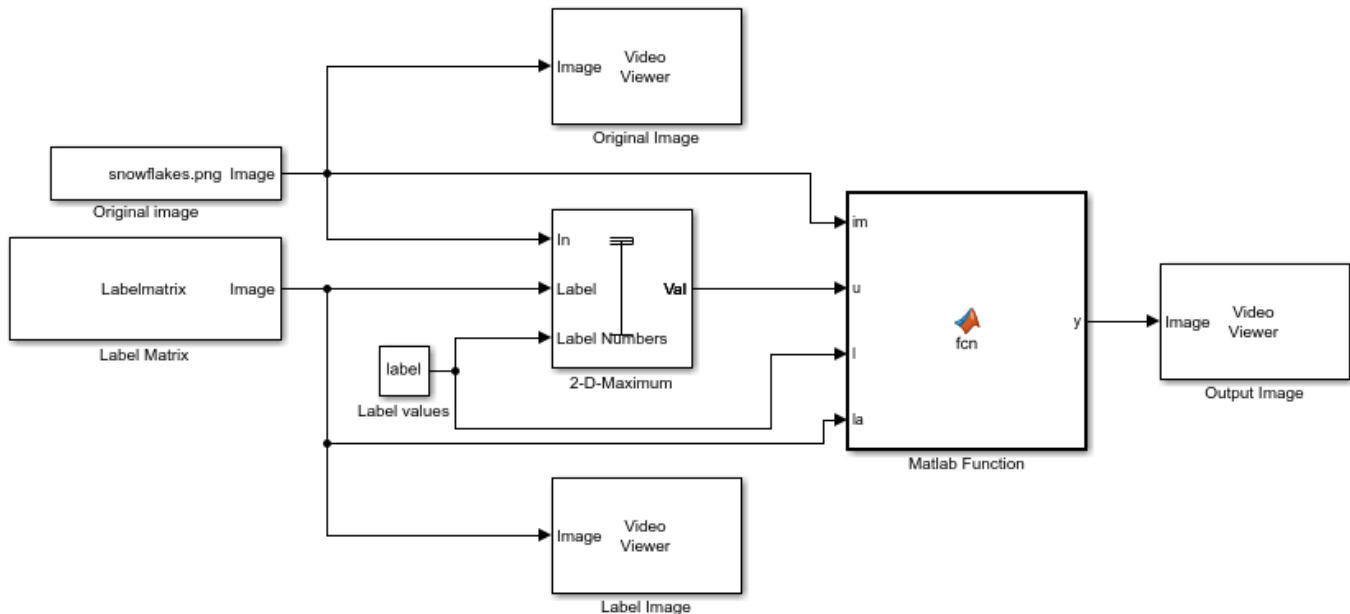
Find the unique label values in the label matrix. The label value 0 corresponds to the background and must be ignored.

```
lb = unique(Labelmatrix);
label = lb(2:end);
```

Example Model

Open the Simulink model. The model reads the input image using the Image From File block.

```
open_system('ex_blk2dmaximum.slx');
```



The model computes the maximum intensity value for each ROI and replaces all the pixel values in the ROI to maximum value. The model computes the maximum value for each ROI individually by setting these 2-D Maximum block parameters to the specified value,

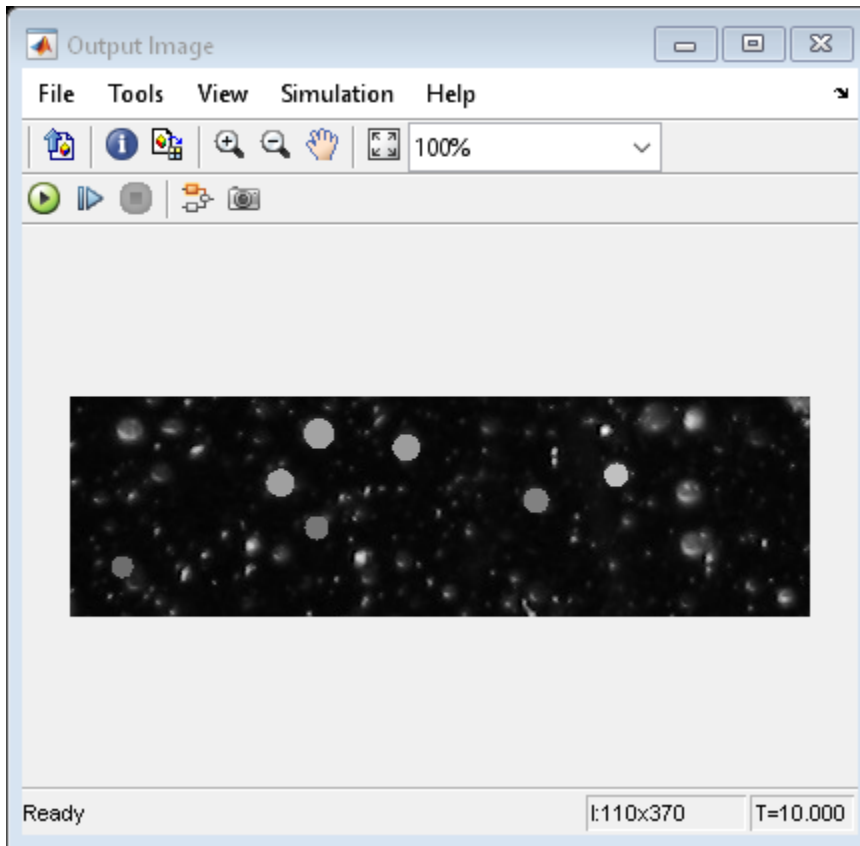
- Set the Mode parameter to Value.
- Set the Find the maximum value over parameter to Entire input.
- Set the Enable ROI processing parameter and set the ROI type parameter as Label matrix.

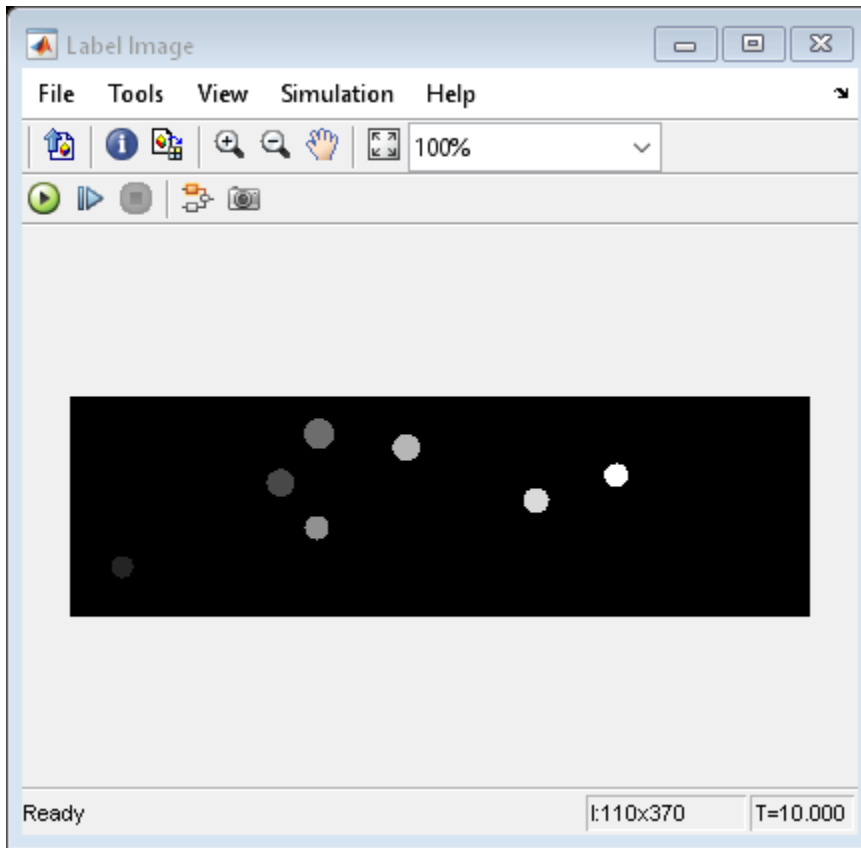
The Matlab Function block replaces the individual ROI's with its maximum intensity value and outputs the resultant image.

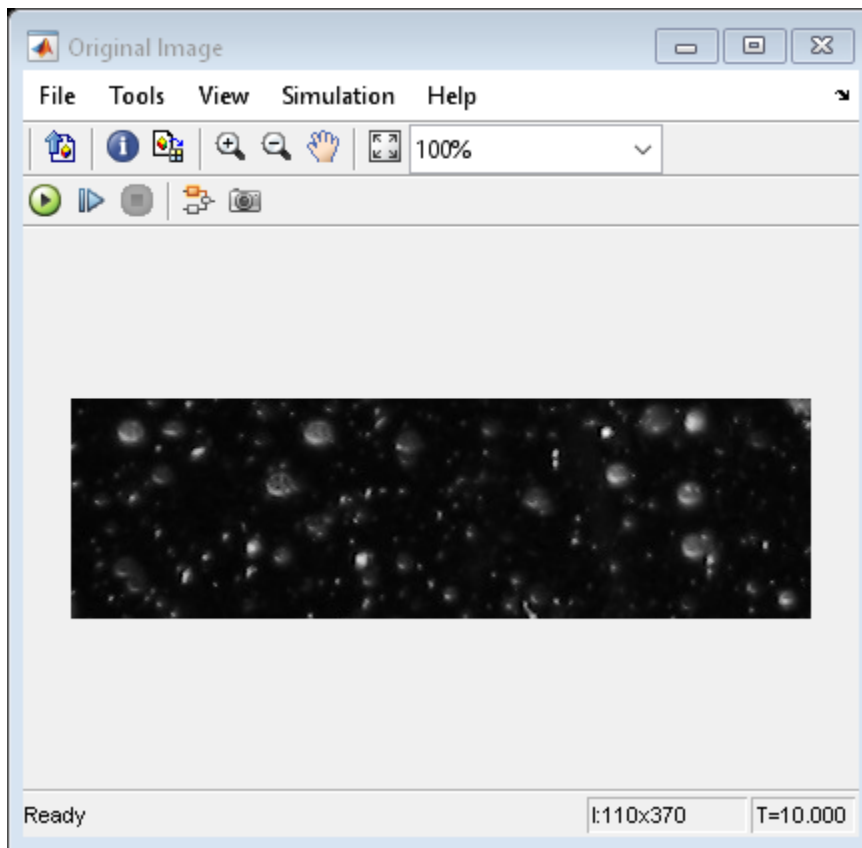
Simulate and Display Results

Run the model and display the images using Video Viewer block.

```
sim('ex_blk2dmaximum.slx');
```







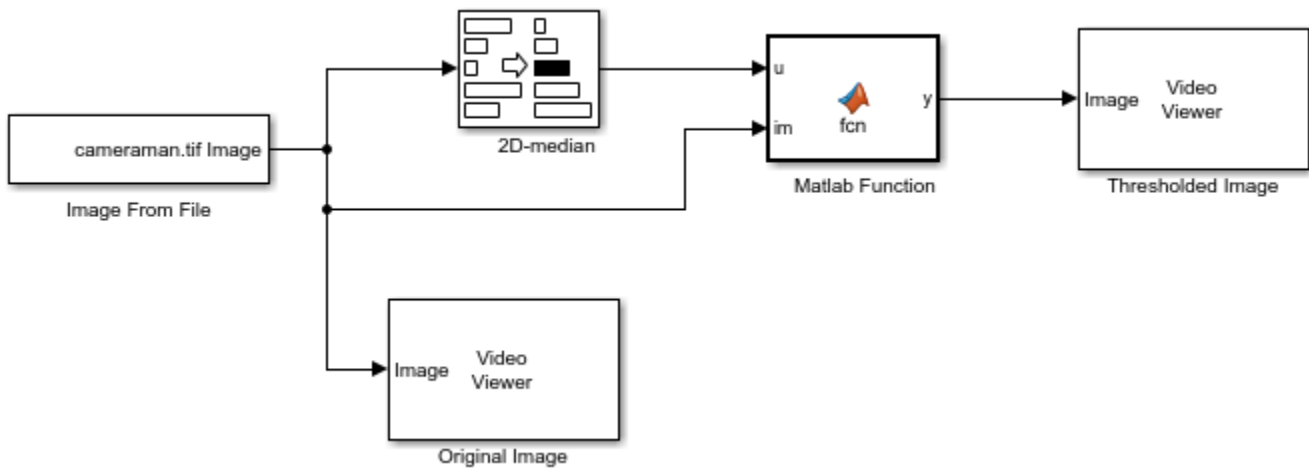
Median based Image Thresholding

This example shows how to perform image thresholding with the median value of the image as a global threshold.

Example Model

Open the simulink model.

```
open_system('ex_blk2dmedian.slx');
```

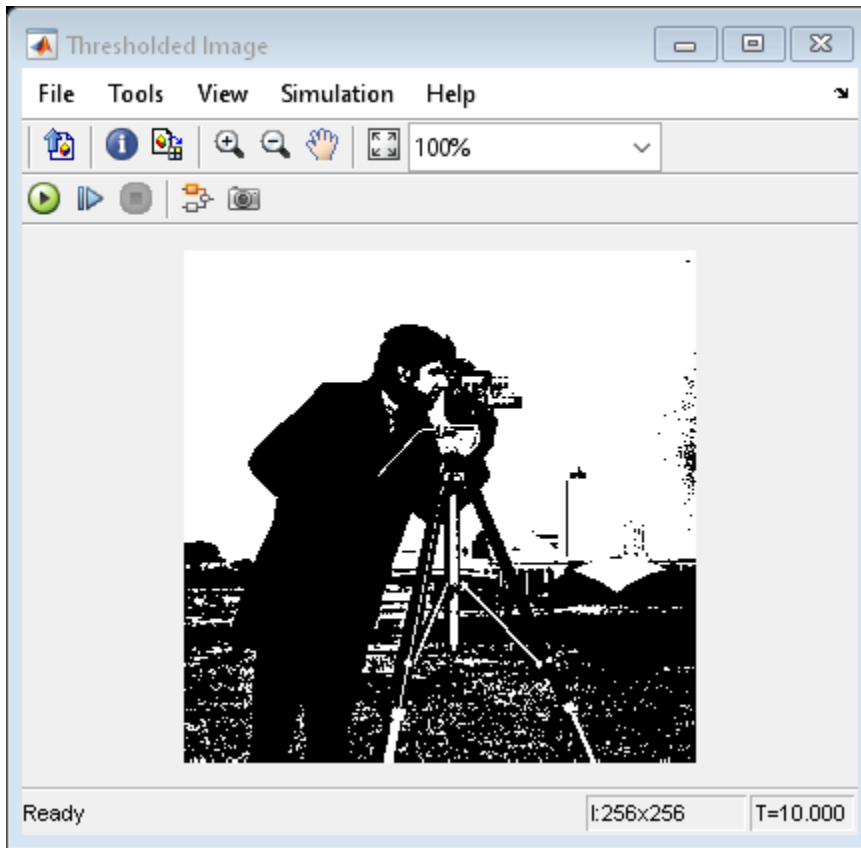


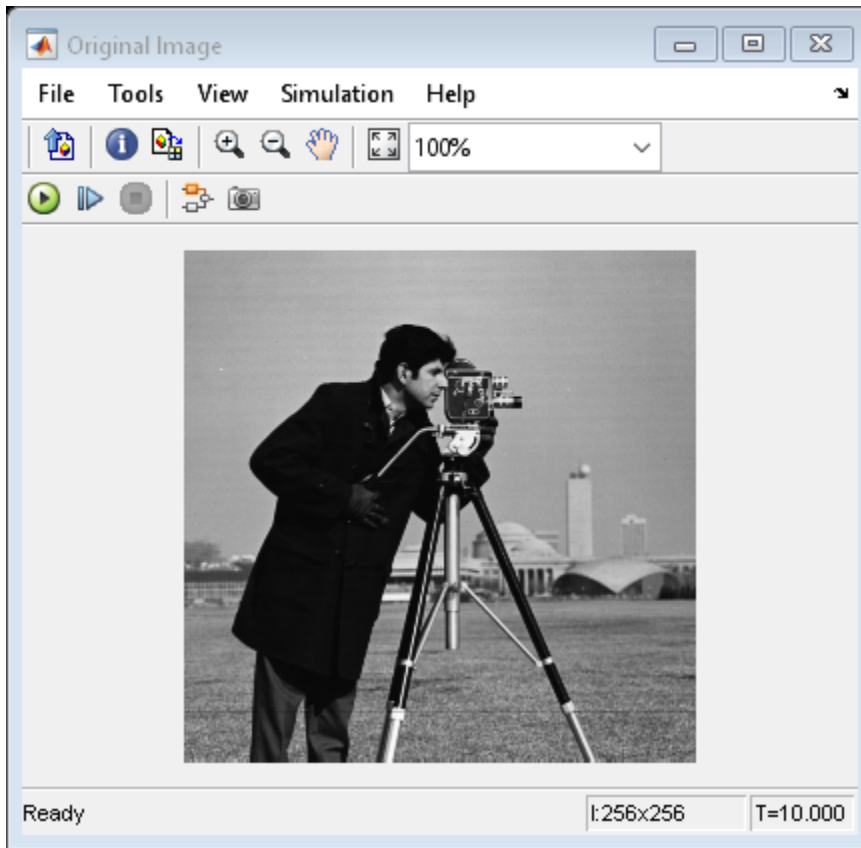
The model computes the median value of an input image by setting the Find the median value over parameter to Entire input in 2-D Median block. The output of the 2-D Median block is a scalar. The Matlab Function block performs image thresholding by taking the output median as a global threshold. If the intensity value in the input image is greater than the median value, it is set to '1'. Otherwise, the intensity value is set to '0'. The output of the model is the thresholded image.

Simulate and Display Results

Run the model and display the results using Video Viewer block.

```
sim('ex_blk2dmedian.slx');
```





Import Image From MATLAB Workspace

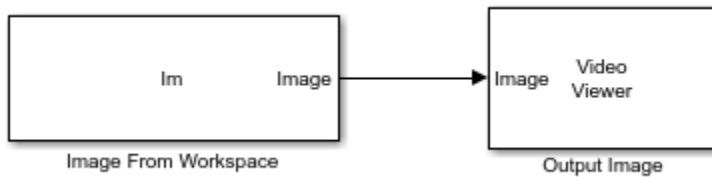
This example shows how to import an image from MATLAB to Simulink workspace using Image From Workspace block.

Load a .mat file containing the image to import from MATLAB workspace. The image is stored in the variable `Im`. Set the `Value` parameter of the Image From Workspace block to the variable in MATLAB workspace.

```
load('inputimage.mat')
```

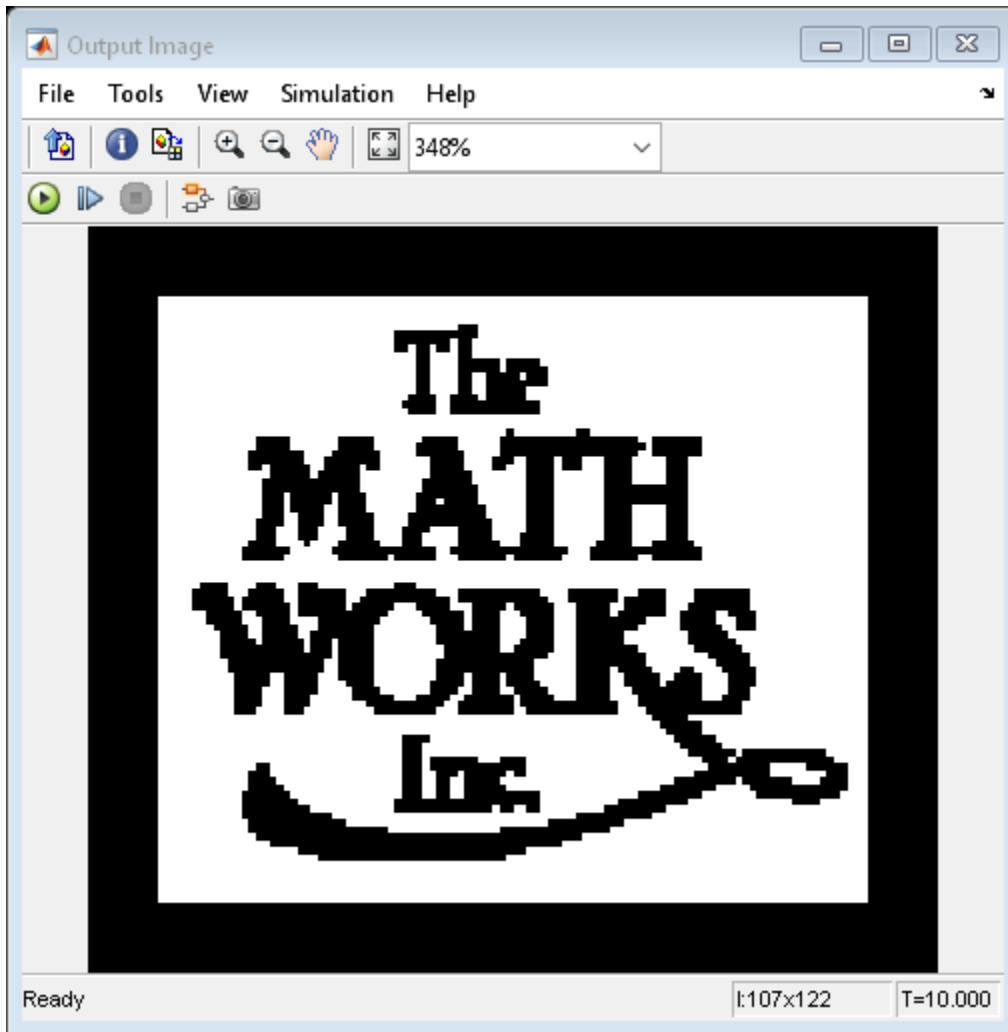
Open the Simulink model.

```
open_system('ex_blkimagefromworkspace.slx');
```



Run the model. The model exports the image to the Simulink workspace and displays the output image.

```
sim('ex_blkimagefromworkspace.slx');
```



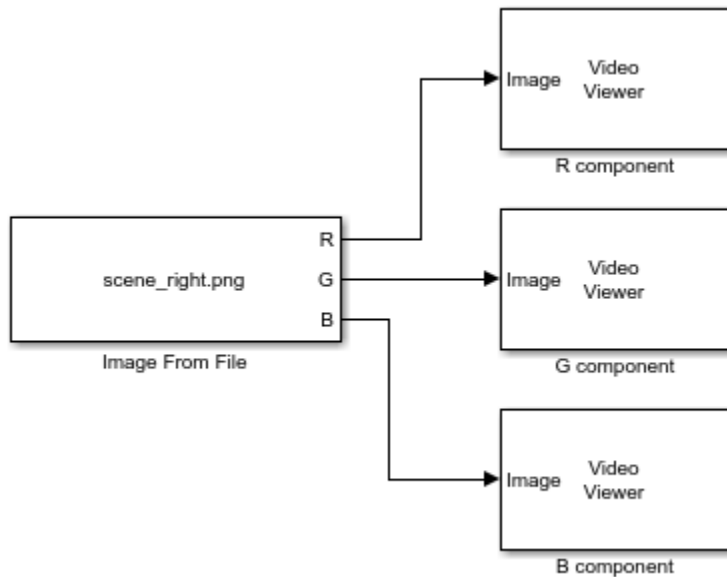
Import Image from Specified Location

This example shows how to import an image from a file in the specified location to Simulink workspace by using the Image From File block.

Example Model

Open the simulink model.

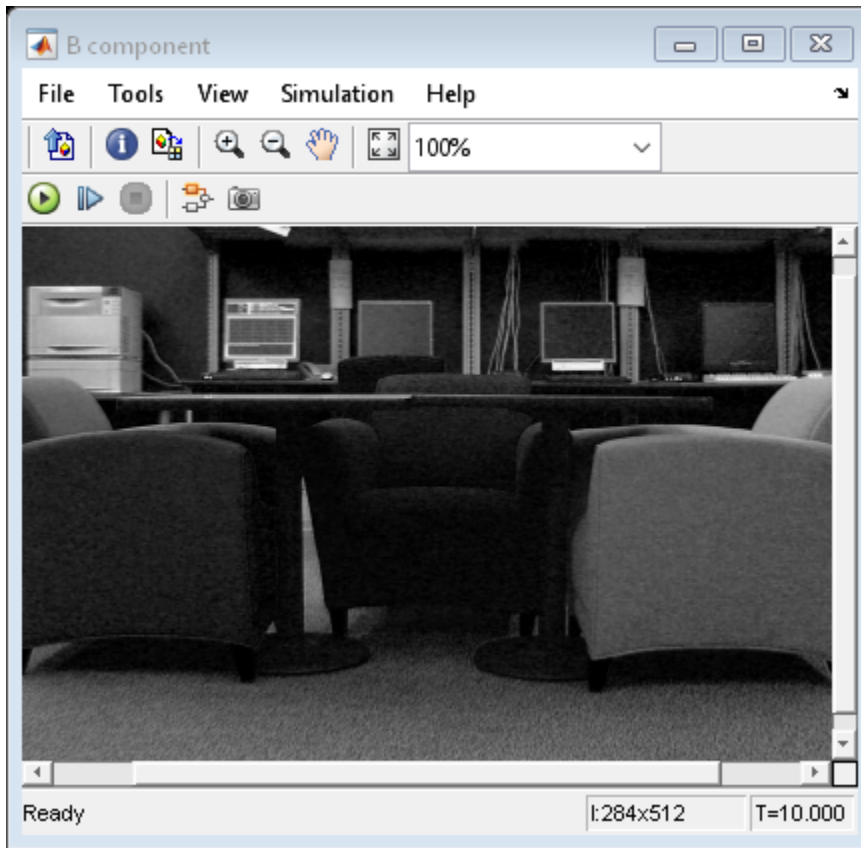
```
open('ex_readcolorimage');
```

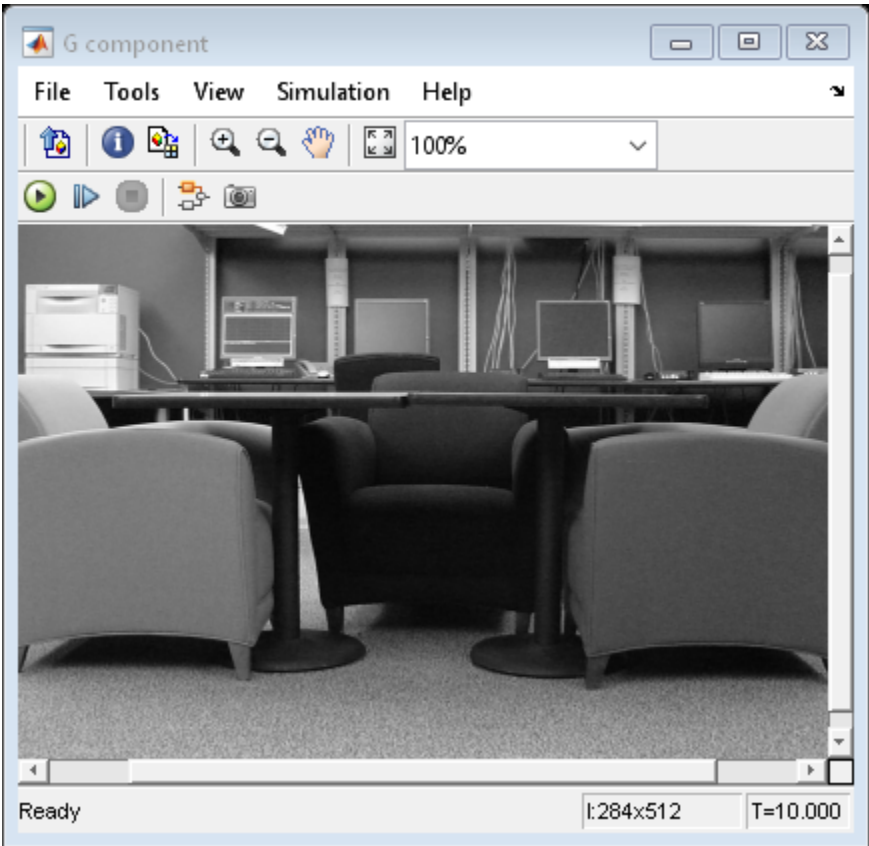


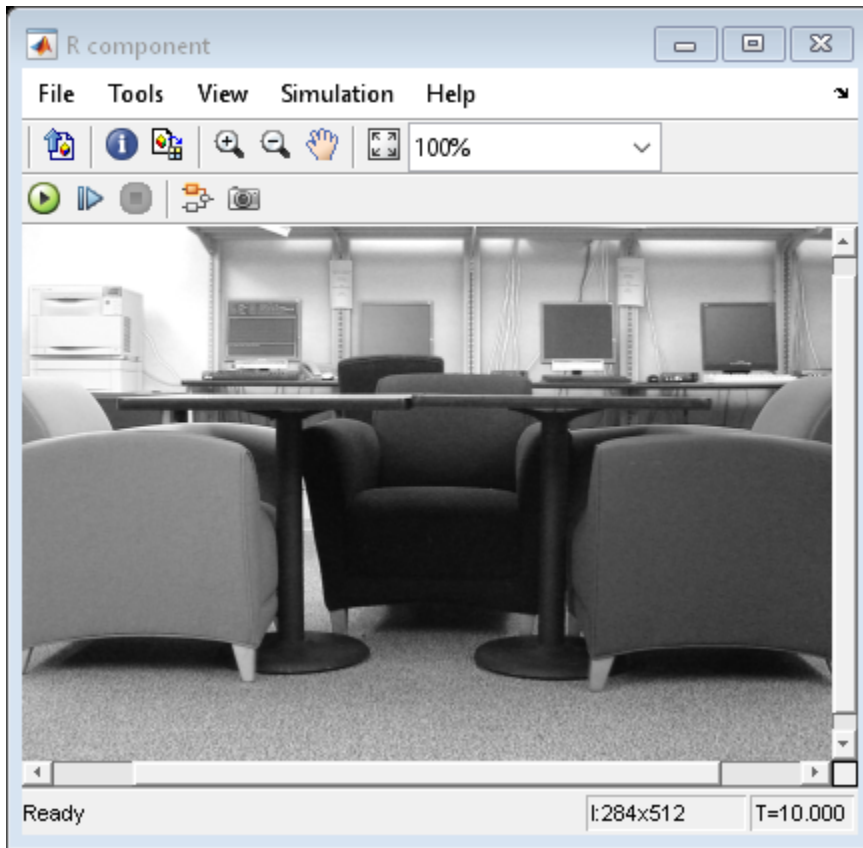
The model imports a color image to the Simulink workspace and displays the R, G, and B color components of the image by using the Video Viewer block. The **Image Signal** parameter of the Image From File block is set to Separate color signals in order to import the RGB color components separately.

Simulate and Display Results

```
sim('ex_readcolorimage');
```







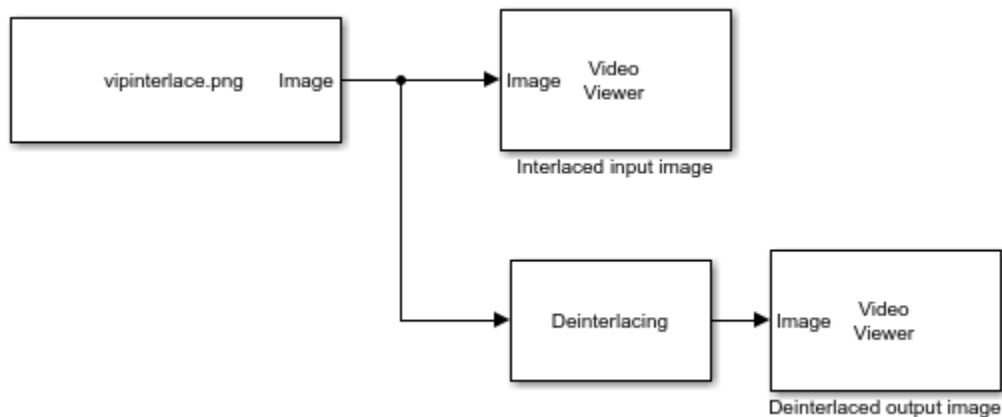
Remove Interlacing Effect From Image

This example shows you how to remove interlacing effects from an image by using the Deinterlacing block.

Example Model

Open the model by calling the open function in MATLAB command prompt. Specify the name of the Simulink file to open.

```
open('blk_deinterlace.slx');
```

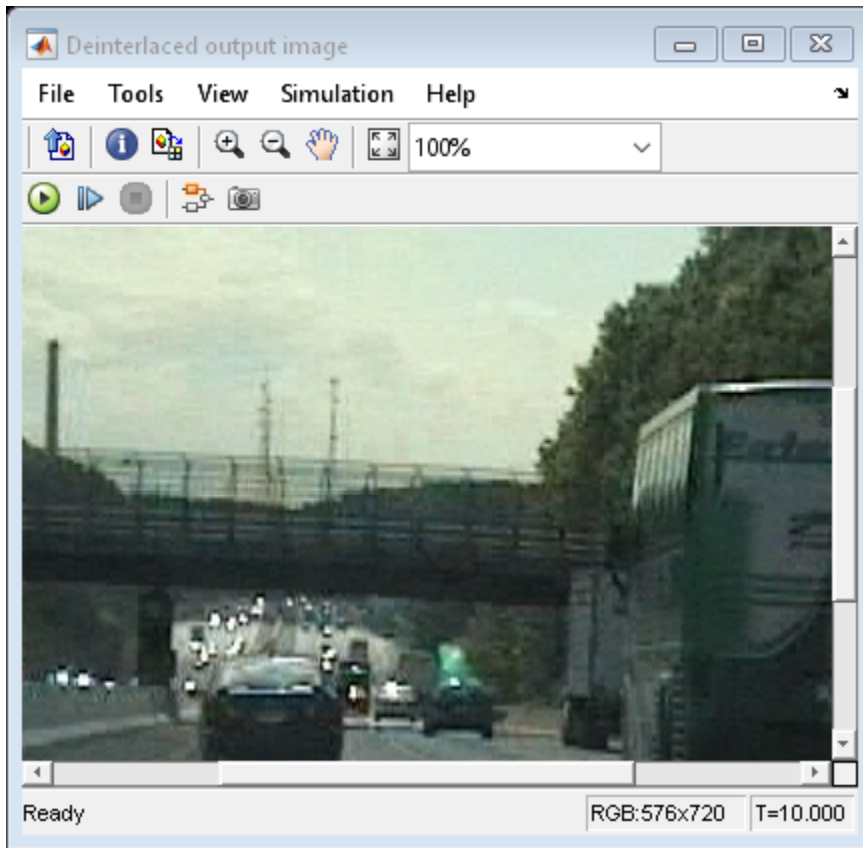


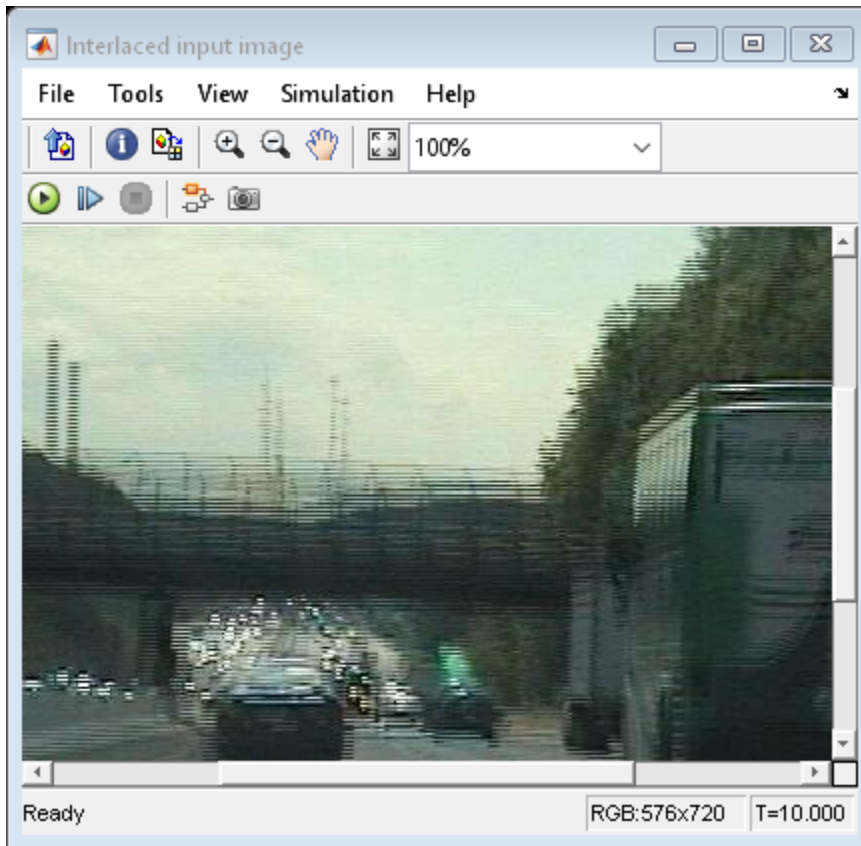
The model uses the Image From File block to read an interlaced image from a specified file location. The Method parameter of the Deinterlacing block is set to Linear interpolation. Hence, the model converts the interlaced image at the input into a deinterlaced image by using the linear interpolation technique. The fixed-point parameters and the data types are set to default values. The model displays the interlaced input image and the deinterlaced output image by using the Video viewer block.

Run Model

Simulate the model and display the results.

```
sim('blk_deinterlace.slx');
```



The interlaced image has jagged lines that are the result of the temporal lag between the top and the bottom fields of the image. The Deinterlacing block removes the jagged lines and the output image is free from visible artifacts.

Estimate Motion between Two Images

This example shows how to use the Block Matching block to estimate motion between two images.

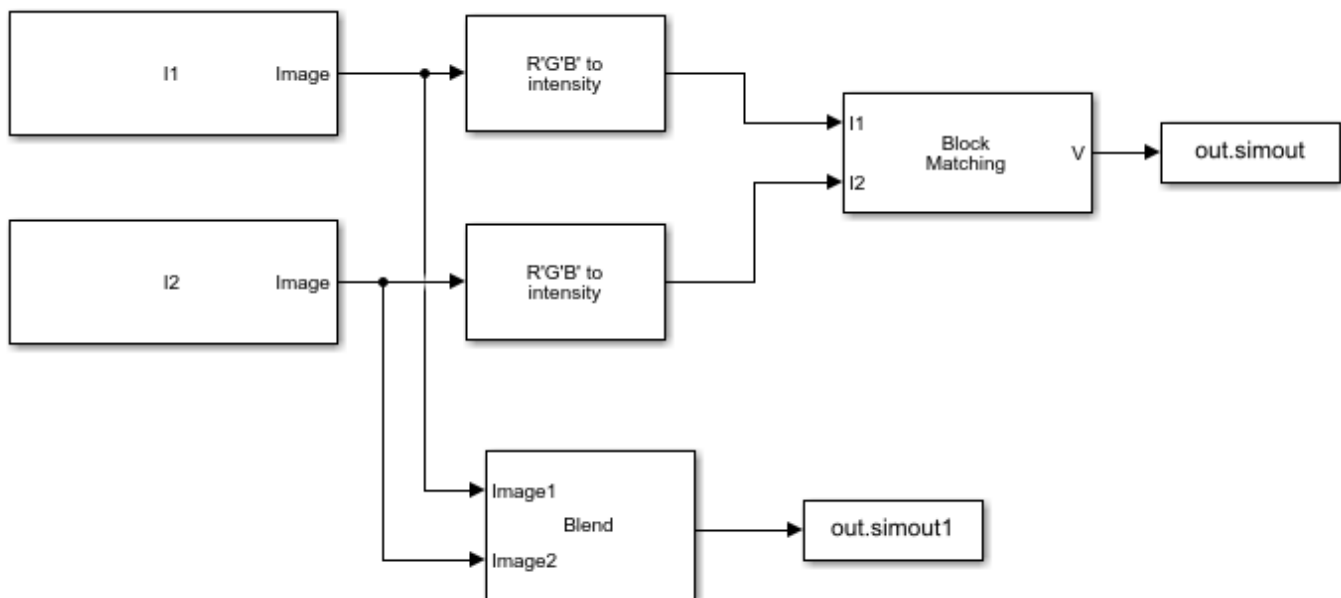
Read image frames for which motion has to be estimated.

```
I1 = imread('car_frame1.png');
I2 = imread('car_frame2.png');
```

Example Model

Open the model by calling the open function in MATLAB command prompt. Specify the name of the Simulink file to open.

```
open_system('ex_blockmatching.slx');
```



Load the images into the model workspace by using the Image From Workspace block. To directly read images from a file location, use the Image From File block instead. The model estimates motion between two RGB images of a moving car that are captured at different time intervals. The model uses the three step block matching algorithm for motion estimation. The cost function for matching the non-overlapping macro blocks is set to mean square error (MSE). The size of the macro blocks is set to 35-by-35 and maximum displacement (in horizontal and vertical direction) allowed for the matching blocks is set to 7 pixels. The velocity output from the Block Matching block consists of both the horizontal and vertical components of the motion vector in complex form.

You can use the Compositing block to overlay both the images.

Run Model

Simulate the model and save the model output to MATLAB workspace. The model outputs the motion vector and the overlaid image.

```
out = sim('ex_blockmatching.slx');
```

Display Results

Read the output motion vector and the overlaid image.

```
vx = real(out.simout.Data);  
vy = imag(out.simout.Data);  
imageOverlay = out.simout1.Data;
```

Specify the points on the image plane relative to the size of the macro blocks.

```
x = 1:35:size(imageOverlay,1);  
y = 1:35:size(imageOverlay,2);
```

Display the overlaid image and plot the horizontal and vertical components of the motion vector by using the `quiver` function.

```
figure,imshow(imageOverlay);  
hold on  
quiver(y',x,vx,vy,0);
```

