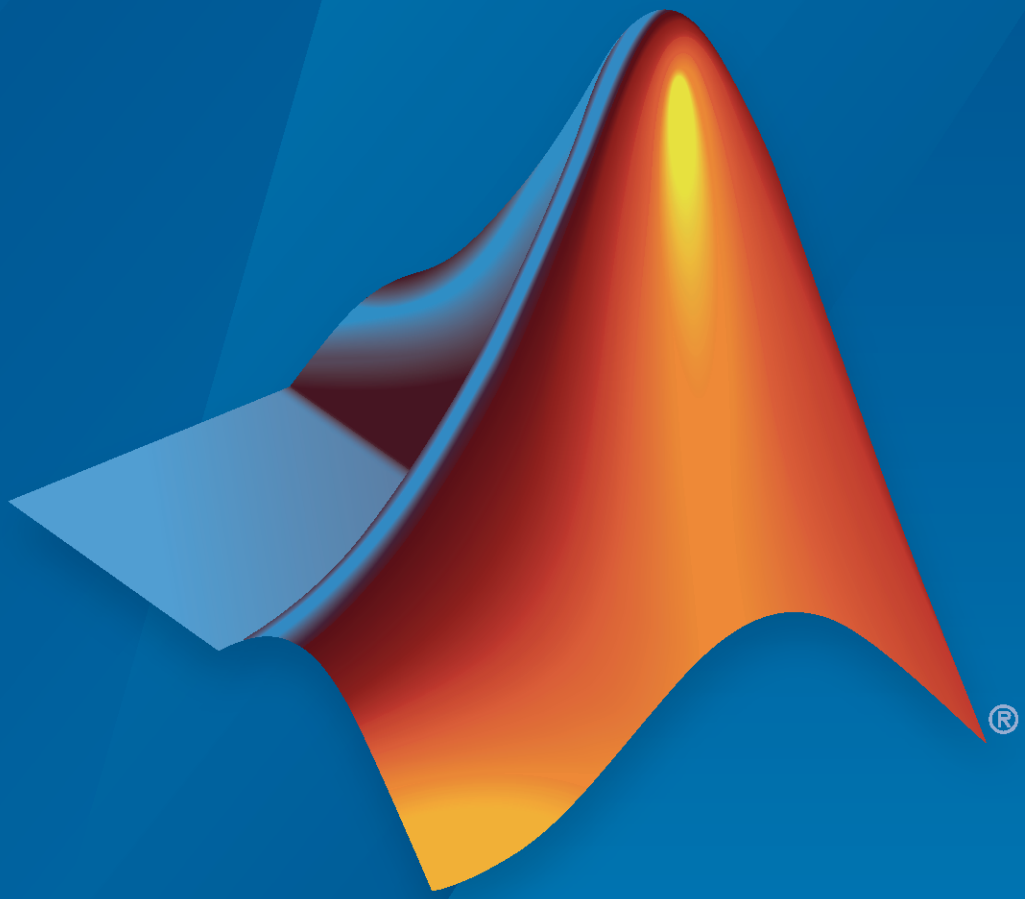
Computer Vision Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2022a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

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

Phone: 508-647-7000

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

Computer Vision Toolbox™ User's Guide

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)
March 2021	Online only	Revised for Version 10.0 (Release R2021a)
September 2021	Online only	Revised for Version 10.1 (Release R2021b)
March 2022	Online only	Revised for Version 10.2 (Release R2022a)

Contents

1 Camera Calibration and SfM Examples

Visual SLAM with an RGB-D Camera **1-2**

Import Stereo Camera Parameters from ROS **1-16**

Import Camera Intrinsic Parameters from ROS **1-20**

Develop Visual SLAM Algorithm Using Unreal Engine Simulation **1-24**

Visual Localization in a Parking Lot **1-38**

Stereo Visual SLAM for UAV Navigation in 3D Simulation **1-44**

Camera Calibration Using AprilTag Markers **1-50**

Configure Monocular Fisheye Camera **1-67**

Monocular Visual Simultaneous Localization and Mapping **1-72**

Structure From Motion From Two Views **1-92**

Stereo Visual Simultaneous Localization and Mapping **1-102**

Evaluating the Accuracy of Single Camera Calibration **1-117**

Measuring Planar Objects with a Calibrated Camera **1-122**

Depth Estimation From Stereo Video **1-131**

Structure From Motion From Multiple Views **1-141**

Uncalibrated Stereo Image Rectification **1-149**

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 **2-9**

Code Generation for Face Tracking with PackNGo **2-16**

Code Generation for Depth Estimation From Stereo Video **2-24**

Detect Face (Raspberry Pi2) **2-29**

Track Face (Raspberry Pi2) **2-35**

Video Display in a Custom User Interface **2-41**

Generate Code for Detecting Objects in Images by Using ACF Object Detector **2-46**

3 Deep Learning, Semantic Segmentation, and Detection Examples

Multiclass Object Detection Using YOLO v2 Deep Learning **3-3**

Generate Adversarial Examples for Semantic Segmentation **3-13**

Classify Defects on Wafer Maps Using Deep Learning **3-24**

Detect Image Anomalies Using Explainable One-Class Classification Neural Network **3-40**

Detect Image Anomalies Using Pretrained ResNet-18 Feature Embeddings **3-59**

Train Object Detectors in Experiment Manager **3-79**

Activity Recognition Using R(2+1)D Video Classification **3-86**

Activity Recognition from Video and Optical Flow Data Using Deep Learning **3-108**

Evaluate a Video Classifier **3-136**

Extract Training Data for Video Classification **3-140**

Classify Streaming Webcam Video Using SlowFast Video Classifier . . . **3-144**

Gesture Recognition using Videos and Deep Learning **3-147**

Explore Semantic Segmentation Network Using Grad-CAM **3-168**

Point Cloud Classification Using PointNet Deep Learning **3-178**

Object Detection Using SSD Deep Learning **3-201**

Object Detection in a Cluttered Scene Using Point Feature Matching **3-211**

Semantic Segmentation Using Deep Learning **3-222**

Calculate Segmentation Metrics in Block-Based Workflow **3-239**

Semantic Segmentation of Multispectral Images Using Deep Learning **3-244**

3-D Brain Tumor Segmentation Using Deep Learning **3-262**

Image Category Classification Using Bag of Features **3-275**

Image Category Classification Using Deep Learning **3-282**

Image Retrieval Using Customized Bag of Features **3-291**

Create SSD Object Detection Network **3-298**

Train YOLO v2 Network for Vehicle Detection **3-301**

Import Pretrained ONNX YOLO v2 Object Detector **3-306**

Export YOLO v2 Object Detector to ONNX **3-313**

Estimate Anchor Boxes From Training Data **3-319**

Object Detection Using YOLO v3 Deep Learning **3-323**

Object Detection Using YOLO v2 Deep Learning **3-338**

Create YOLO v2 Object Detection Network **3-348**

Train Object Detector Using R-CNN Deep Learning **3-351**

Object Detection Using Faster R-CNN Deep Learning **3-364**

Train Classification Network to Classify Object in 3-D Point Cloud . . . **3-374**

Estimate Body Pose Using Deep Learning **3-384**

Generate Image from Segmentation Map Using Deep Learning **3-392**

Train Simple Semantic Segmentation Network in Deep Network Designer **3-407**

Train ACF-Based Stop Sign Detector **3-412**

Train Fast R-CNN Stop Sign Detector **3-415**

Perform Instance Segmentation Using Mask R-CNN **3-418**

Object Detection Using YOLO v4 Deep Learning 3-423

4 Feature Detection and Extraction Examples

Automatically Detect and Recognize Text Using MSER and OCR 4-2

Automatically Detect and Recognize Text Using Pretrained CRAFT Network and OCR 4-14

Digit Classification Using HOG Features 4-21

Find Image Rotation and Scale Using Automated Feature Matching . . . 4-29

Feature Based Panoramic Image Stitching 4-34

Cell Counting 4-40

Object Counting 4-43

Pattern Matching 4-45

Recognize Text Using Optical Character Recognition (OCR) 4-50

Cell Counting 4-64

5 Lidar and Point Cloud Processing Examples

Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment 5-2

Ground Plane and Obstacle Detection Using Lidar 5-12

Augment Point Cloud Data For Deep Learning 5-21

Import Point Cloud Data For Deep Learning 5-26

Encode Point Cloud Data For Deep Learning 5-30

Build a Map from Lidar Data 5-35

Build a Map from Lidar Data Using SLAM 5-55

3-D Point Cloud Registration and Stitching 5-71

6 Computer Vision with Simulink Examples

Multicore Simulation of Video Processing System 6-2

Concentricity Inspection 6-6

Object Counting 6-8

Video Focus Assessment 6-10

Video Compression 6-12

Motion Detection 6-14

Pattern Matching 6-16

Scene Change Detection 6-19

Surveillance Recording 6-21

Traffic Warning Sign Recognition 6-23

Abandoned Object Detection 6-26

Color-based Road Tracking 6-29

Detect and Track Face 6-33

Lane Departure Warning System 6-40

Tracking Cars Using Foreground Detection 6-44

Tracking Cars Using Optical Flow 6-47

Tracking Based on Color 6-49

Video Mosaicking 6-51

Video Stabilization 6-56

Periodic Noise Reduction 6-58

Rotation Correction 6-60

Barcode Recognition Using Live Video Acquisition 6-63

Edge Detection Using Live Video Acquisition 6-65

Noise Removal and Image Sharpening 6-70

Track Marker Using Simulink Images 6-76

7 Video and Image Ground Truth Labeling

Export Ground Truth Object to Custom and COCO JSON Files **7-2**

Automate Ground Truth Labeling for Semantic Segmentation **7-7**

8 Tracking and Motion Estimation Examples

Video Stabilization **8-2**

Video Stabilization Using Point Feature Matching **8-5**

Face Detection and Tracking Using CAMShift **8-15**

Face Detection and Tracking Using the KLT Algorithm **8-20**

Face Detection and Tracking Using Live Video Acquisition **8-26**

Motion-Based Multiple Object Tracking **8-31**

Tracking Pedestrians from a Moving Car **8-40**

Use Kalman Filter for Object Tracking **8-50**

Detect Cars Using Gaussian Mixture Models **8-61**

9 Labelers

Create Automation Algorithm for Labeling **9-2**
Create New Algorithm **9-2**
Import Existing Algorithm **9-3**
Custom Algorithm Execution **9-3**

Label Large Images in the Image Labeler **9-6**
Import Blocked Image into Image Labeler **9-6**
Work with Blocked Images in the Image Labeler **9-8**
Use Blocked Image Automation with Images **9-9**
Postprocess Exported Labels to Create a Labeled Blocked Image **9-12**

Label Pixels for Semantic Segmentation **9-14**
Start Pixel Labeling **9-14**
Label Pixels Using Flood Fill Tool **9-14**
Label Pixels Using Superpixel Tool **9-15**
Label Pixels Using Smart Polygon Tool **9-16**
Label Pixels Using Polygon Tool **9-18**

Label Pixels Using Assisted Freehand Tool 9-18
Replace Pixel Labels 9-19
Refine Labels Using Brush Tool 9-19
Visualize Pixel Labels 9-20
Tips 9-21

Label Objects Using Polygons 9-23
About Polygon Labels 9-23
Load Unlabeled Data 9-23
Create Polygon Labels 9-24
Draw Polygon ROI Labels 9-24
Modify Polygon Preferences and Stacking Order 9-24
Postprocess Exported Labels for Instance or Semantic Segmentation Networks 9-27

Get Started with the Image Labeler 9-30
Load Unlabeled Data 9-30
Create Label Definitions 9-31
Label Ground Truth 9-36
Export Labeled Ground Truth 9-37
Save App Session 9-39

Choose an App to Label Ground Truth Data 9-40

Get Started with the Video Labeler 9-44
Load Unlabeled Data 9-44
Create Label Definitions 9-44
Label Ground Truth 9-50
Export Labeled Ground Truth 9-52
Label Data 9-54
Save App Session 9-55

Use Custom Image Source Reader for Labeling 9-57
Create Custom Reader Function 9-57
Import Data Source into Video Labeler App 9-57
Import Data Source into Ground Truth Labeler App 9-58

Keyboard Shortcuts and Mouse Actions for Video Labeler 9-59
Label Definitions 9-59
Frame Navigation and Time Interval Settings 9-59
Labeling Window 9-59
Polyline Drawing 9-60
Polygon Drawing 9-61
Zooming and Panning 9-61
App Sessions 9-61

Keyboard Shortcuts and Mouse Actions for Image Labeler 9-63
Label Definitions 9-63
Image Browsing and Selection 9-63
Labeling Window 9-63
Polyline Drawing 9-64
Polygon Drawing 9-65
Zooming 9-65
Zooming and Panning 9-66
App Sessions 9-66

Share and Store Labeled Ground Truth Data **9-67**
Share Ground Truth **9-67**
Move Ground Truth **9-70**
Store Ground Truth **9-71**
Extract Labeled Video Scenes **9-71**

View Summary of Ground Truth Labels **9-73**
View Label Summary **9-73**
Compare Selected Labels **9-75**

Temporal Automation Algorithms **9-77**
Create Temporal Automation Algorithm **9-77**
Run Temporal Automation Algorithm **9-77**

Blocked Image Automation Algorithms **9-79**
Create Blocked Image Automation Algorithm **9-79**
Run Blocked Image Automation Algorithm **9-79**

Use Sublabels and Attributes to Label Ground Truth Data **9-80**
When to Use Sublabels vs. Attributes **9-80**
Draw Sublabels **9-81**
Copy and Paste Sublabels **9-81**
Delete Sublabels **9-82**
Sublabel Limitations **9-83**

Training Data for Object Detection and Semantic Segmentation **9-84**

10 Featured Examples

Localize and Read Multiple Barcodes in Image **10-2**

Monocular Visual Odometry **10-22**

Detect and Track Vehicles Using Lidar Data **10-35**

Semantic Segmentation Using Dilated Convolutions **10-54**

Define Custom Pixel Classification Layer with Tversky Loss **10-59**

Track a Face in Scene **10-66**

Create 3-D Stereo Display **10-71**

Measure Distance from Stereo Camera to a Face **10-72**

Reconstruct 3-D Scene from Disparity Map **10-73**

Visualize Stereo Pair of Camera Extrinsic Parameters **10-76**

Remove Distortion from an Image Using the Camera Parameters Object **10-79**

11 Structure from Motion and Visual SLAM

Choose SLAM Workflow Based on Sensor Data **11-2**
Choose SLAM Workflow **11-2**

Implement Visual SLAM in MATLAB **11-8**
Terms Used in Visual SLAM **11-8**
Typical Feature-based Visual SLAM Workflow **11-8**
Key Frame and Map Data Management **11-9**
Map Initialization **11-10**
Tracking **11-11**
Local Mapping **11-12**
Loop Detection **11-14**
Drift Correction **11-14**
Visualization **11-15**

12 Point Cloud Processing

Getting Started with Point Clouds Using Deep Learning **12-2**
Import Point Cloud Data **12-2**
Augment Data **12-2**
Encode Point Cloud Data to Image-like Format **12-3**
Train a Deep Learning Classification Network with Encoded Point Cloud Data **12-3**

Implement Point Cloud SLAM in MATLAB **12-4**
Mapping and Localization Workflow **12-4**
Manage Data for Mapping and Localization **12-6**
Preprocess Point Clouds **12-6**
Register Point Clouds **12-6**
Detect Loops **12-8**
Correct Drift **12-9**
Assemble Map **12-9**
Localize Vehicle in Map **12-9**
Alternate Workflows **12-9**

The PLY Format **12-12**
File Header **12-12**
Data **12-13**
Common Elements and Properties **12-14**

13 Using the Installer for Computer Vision System Toolbox Product

Install Computer Vision Toolbox Add-on Support Files **13-2**

Install OCR Language Data Files **13-3**
Installation **13-3**
Pretrained Language Data and the ocr function **13-3**

Install and Use Computer Vision Toolbox Interface for OpenCV in MATLAB **13-6**
Installation **13-6**
Support Package Contents **13-6**

Build MEX-Files for OpenCV Interface **13-8**
Create MEX-File from OpenCV C++ file **13-8**
Create Your Own OpenCV MEX-files **13-8**
Run OpenCV Examples **13-8**

Use Prebuilt MATLAB Interface to OpenCV **13-10**
Call MATLAB Functions **13-11**
Call Functions in OpenCV Library **13-11**
Display Help for MATLAB Functions **13-12**
Display Help for MATLAB Interface to OpenCV Library **13-12**
Limitations **13-13**

Perform Edge-Preserving Image Smoothing Using OpenCV in MATLAB **13-15**

Subtract Image Background by Using OpenCV in MATLAB **13-19**

Perform Face Detection by Using OpenCV in MATLAB **13-22**

Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink **13-24**
Installation **13-24**
Import OpenCV Code into Simulink **13-24**
Limitations **13-29**

Draw Different Shapes by Using OpenCV Code in Simulink **13-31**

Convert RGB Image to Grayscale Image by Using OpenCV Importer . . **13-38**

Smile Detection by Using OpenCV Code in Simulink **13-45**

Shadow Detection by Using OpenCV Code in Simulink **13-55**

Vehicle and Pedestrian Detector by Using OpenCV Importer **13-60**

Video Cartoonizer by Using OpenCV Code in Simulink **13-64**

Convert Between Simulink Image Type and Matrices **13-69**
Copy Example Model to a Writable Location **13-69**
Example Model **13-69**
Simulate Model **13-69**
Generate C++ Code **13-70**

14 Input, Output, and Conversions

Export to Video Files **14-2**
Setting Block Parameters for this Example **14-2**
Configuration Parameters **14-3**

Import from Video Files **14-4**
Setting Block Parameters for this Example **14-4**
Configuration Parameters **14-5**

Batch Process Image Files **14-6**
Configuration Parameters **14-6**

Convert R'G'B' to Intensity Images **14-7**

Process Multidimensional Color Video Signals **14-10**

Video Formats **14-12**
Defining Intensity and Color **14-12**
Video Data Stored in Column-Major Format **14-12**

Image Formats **14-13**
Binary Images **14-13**
Intensity Images **14-13**
RGB Images **14-13**

15 Display and Graphics

Choose Function to Visualize Detected Objects **15-2**

Display, Stream, and Preview Videos **15-5**
View Streaming Video in MATLAB **15-5**
Preview Video in MATLAB **15-5**
View Video in Simulink **15-5**

Draw Shapes and Lines **15-7**
Rectangle **15-7**
Line and Polyline **15-7**
Polygon **15-9**
Circle **15-9**

16 Registration and Stereo Vision

Select Calibration Pattern and Set Properties **16-2**

Prepare Camera and Capture Images **16-4**
Camera Setup **16-4**
Capture Images **16-4**

Calibration Patterns **16-6**
What Are Calibration Patterns? **16-6**
Supported Patterns **16-8**
Checkerboard Pattern **16-8**
Circle Grid Patterns **16-9**
Custom Detector Pattern **16-9**

Fisheye Calibration Basics **16-11**
Fisheye Camera Model **16-12**
Fisheye Camera Calibration in MATLAB **16-14**

Using the Single Camera Calibrator App **16-18**
Camera Calibrator Overview **16-18**
Choose a Calibration Pattern **16-18**
Capture Calibration Images **16-18**
Using the Camera Calibrator App **16-19**

Using the Stereo Camera Calibrator App **16-31**
Stereo Camera Calibrator Overview **16-31**
Choose a Calibration Pattern **16-32**
Capture Calibration Images **16-32**
Using the Stereo Camera Calibrator App **16-32**

What Is Camera Calibration? **16-43**
Camera Models **16-43**
Pinhole Camera Model **16-44**
Camera Calibration Parameters **16-45**
Distortion in Camera Calibration **16-46**

Structure from Motion Overview **16-49**
Structure from Motion from Two Views **16-49**
Structure from Motion from Multiple Views **16-50**

17 Object Detection

Getting Started with Anomaly Detection Using Deep Learning **17-2**
Prepare Training and Calibration Data **17-2**
Train Model **17-3**
Calibrate Model **17-3**
Perform Classification Using Model **17-3**
Deploy Model **17-3**

Getting Started with Video Classification Using Deep Learning **17-4**
Create Training Data for Video Classification **17-5**
Create Video Classifier **17-5**
Train Video Classifier and Evaluate Results **17-12**
Classify Using Deep Learning Video Classifiers **17-13**

Choose an Object Detector **17-14**

Getting Started with SSD Multibox Detection **17-21**
Predict Objects in the Image **17-21**
Design an SSD Detection Network **17-22**
Train an Object Detector and Detect Objects with an SSD Model **17-22**
Transfer Learning **17-23**
Code Generation **17-23**
Label Training Data for Deep Learning **17-23**

Getting Started with Object Detection Using Deep Learning **17-24**
Create Training Data for Object Detection **17-24**
Create Object Detection Network **17-25**
Train Detector and Evaluate Results **17-25**
Detect Objects Using Deep Learning Detectors **17-25**
Detect Objects Using Pretrained Object Detection Models **17-26**

How Labeler Apps Store Exported Pixel Labels **17-27**
Location of Pixel Label Data Folder **17-27**
View Exported Pixel Label Data **17-27**
Examples **17-28**

Anchor Boxes for Object Detection **17-32**
What Is an Anchor Box? **17-32**
Advantage of Using Anchor Boxes **17-32**
How Do Anchor Boxes Work? **17-33**
Anchor Box Size **17-36**

Getting Started with YOLO v2 . . . . . **17-37**
Predicting Objects in the Image **17-37**
Transfer Learning **17-38**
Design a YOLO v2 Detection Network **17-38**
Train an Object Detector and Detect Objects with a YOLO v2 Model . . . **17-39**
Code Generation **17-39**
Label Training Data for Deep Learning **17-39**

Getting Started with YOLO v3 . . . . . **17-41**
Predicting Objects in the Image **17-41**
Design a YOLO v3 Detection Network **17-42**
Transfer Learning **17-42**
Train an Object Detector and Detect Objects with a YOLO v3 Model . . . **17-42**
Label Training Data for Deep Learning **17-42**

Getting Started with YOLO v4 . . . . . **17-44**
Predict Objects Using YOLO v4 . . . . . **17-45**
Create YOLO v4 Object Detection Network **17-45**
Train and Detect Objects Using YOLOv4 Network **17-46**
Transfer Learning **17-47**
Label Training Data for Deep Learning **17-47**

Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN **17-49**
Object Detection Using R-CNN Algorithms **17-49**
Comparison of R-CNN Object Detectors **17-51**
Transfer Learning **17-51**
Design an R-CNN, Fast R-CNN, and a Faster R-CNN Model **17-52**
Label Training Data for Deep Learning **17-53**

Getting Started with Mask R-CNN for Instance Segmentation **17-55**
Mask R-CNN Network Architecture **17-55**
Prepare Mask R-CNN Training Data **17-56**
Train Mask R-CNN Model **17-61**

Getting Started with Semantic Segmentation Using Deep Learning . . **17-62**
Label Training Data for Semantic Segmentation **17-62**
Train and Test a Semantic Segmentation Network **17-63**
Segment Objects Using Pretrained DeepLabv3+ Network **17-63**

Point Feature Types **17-65**
Functions That Return Points Objects **17-65**
Functions That Accept Points Objects **17-67**

Local Feature Detection and Extraction **17-71**
What Are Local Features? **17-71**
Benefits and Applications of Local Features **17-71**
What Makes a Good Local Feature? **17-72**
Feature Detection and Feature Extraction **17-72**
Choose a Feature Detector and Descriptor **17-73**
Use Local Features **17-75**
Image Registration Using Multiple Features **17-81**

Get Started with Cascade Object Detector **17-89**
Why Train a Detector? **17-89**
What Kinds of Objects Can You Detect? **17-89**
How Does the Cascade Classifier Work? **17-89**
Create a Cascade Classifier Using the trainCascadeObjectDetector . . . **17-90**
Troubleshooting **17-93**
Examples **17-95**
Train Stop Sign Detector **17-99**

Train Optical Character Recognition for Custom Fonts **17-102**
Open the OCR Trainer App **17-102**
Train OCR **17-102**
App Controls **17-104**

Troubleshoot ocr Function Results **17-106**
Performance Options with the ocr Function **17-106**

Create a Custom Feature Extractor **17-107**
Example of a Custom Feature Extractor **17-107**

Image Retrieval with Bag of Visual Words **17-110**
Retrieval System Workflow **17-111**
Evaluate Image Retrieval **17-112**

Image Classification with Bag of Visual Words **17-113**
Step 1: Set Up Image Category Sets **17-113**
Step 2: Create Bag of Features **17-113**
Step 3: Train an Image Classifier With Bag of Visual Words **17-114**
Step 4: Classify an Image or Image Set **17-115**

18 Motion Estimation and Tracking

Multiple Object Tracking **18-2**
Detection **18-2**
Prediction **18-2**
Data Association **18-3**
Track Management **18-4**

Video Mosaicking **18-5**

19 Filters, Transforms, and Enhancements

Adjust the Contrast of Intensity Images **19-2**

Adjust the Contrast of Color Images **19-6**

Remove Salt and Pepper Noise from Images **19-10**

Sharpen an Image **19-14**

20 Statistics and Morphological Operations

Correct Nonuniform Illumination **20-2**

Count Objects in an Image **20-8**

21 Fixed-Point Design

Fixed-Point Signal Processing **21-2**
Fixed-Point Features **21-2**
Benefits of Fixed-Point Hardware **21-2**
Benefits of Fixed-Point Design with System Toolboxes Software **21-2**

Fixed-Point Concepts and Terminology **21-4**
Fixed-Point Data Types **21-4**
Scaling **21-5**
Precision and Range **21-6**

Arithmetic Operations **21-8**
Modulo Arithmetic **21-8**
Two's Complement **21-8**
Addition and Subtraction **21-9**
Multiplication **21-10**
Casts **21-12**

Fixed-Point Support for MATLAB System Objects **21-15**
Getting Information About Fixed-Point System Objects **21-15**
Setting System Object Fixed-Point Properties **21-15**

Specify Fixed-Point Attributes for Blocks **21-16**
Fixed-Point Block Parameters **21-16**
Specify System-Level Settings **21-18**
Inherit via Internal Rule **21-18**
Specify Data Types for Fixed-Point Blocks **21-25**

22 Code Generation and Shared Library

Simulink Shared Library Dependencies **22-2**

Accelerating Simulink Models **22-3**

Portable C Code Generation for Functions That Use OpenCV Library . . **22-4**
Limitations **22-4**

23 Vision Blocks Examples

Rotate ROI in Image **23-3**

Apply Horizontal Shear Transformation to Image **23-6**

Find Location of Object in Image Using Template Matching **23-9**

Compute Optical Flow Velocities **23-12**

Rotate an Image **23-14**

Generate Image Histogram **23-17**

Export Image to MATLAB Workspace **23-19**

Import Video from MATLAB Workspace **23-22**

Find Minimum Value in ROI **23-24**

Write Image to Binary File **23-28**

Compute Standard Deviation of ROIs **23-29**

Read Video Stored as Binary Data **23-32**

Compare Image Quality Using PSNR **23-36**

Compute Autocorrelation of Input Matrix **23-38**

Compute Correlation between Two Matrices **23-39**

Find Statistics of Circular Blobs in Image **23-40**

Replace Intensity Values in ROI with its Maximum Value **23-44**

Median based Image Thresholding **23-48**

Import Image From MATLAB Workspace **23-51**

Import Image from Specified Location **23-53**

Remove Interlacing Effect From Image **23-57**

Estimate Motion between Two Images **23-60**

Enhance Contrast of Grayscale Image Using Histogram Equalization **23-62**

Enhance Contrast of Color Image Using Histogram Equalization **23-65**

Compute Mean of ROIs in Image **23-68**

Detect Corners in Image **23-71**

Edge Detection of Intensity Image **23-75**

Read, Process, and Write Video Frames to File **23-78**

Find Local Maxima in Image **23-80**

Read, Convert, and View Video from File **23-83**

Read and Display YCbCr Video from File **23-85**

Display Frame Rate of Input Video **23-87**

Draw Rectangles on Image **23-88**

Draw Circles on Image **23-90**

Overlay Images Using Binary Mask **23-92**

Linearly Combine Two Images **23-97**

Pad Zeros to Image **23-101**

Insert Text into Image **23-104**

Compress Image Using 2-D DCT **23-107**

Draw Markers on Image **23-111**

Read and Display RGB Video from File **23-114**

Label Objects in Binary Image **23-116**

Boundary Extraction of Binary Image **23-120**

Select String to Insert into Image **23-124**

Insert Two Strings into Image at Different Locations **23-127**

Dilation of Binary Image **23-129**

Find Complement of Intensity Image **23-131**

Perform Top-Hat Filtering of Binary Image **23-134**

Perform Bottom-hat Filtering of Binary Image **23-137**

Perform Opening of Binary Image **23-140**

Perform Closing of Binary Image **23-143**

Blur Image Using Gaussian Kernel **23-146**

Convert Image Color Space from RGB to YCbCr **23-149**

Convert Data Type and Color Space of Image from RGB to HSV **23-152**

Perform Gamma Correction of Image **23-155**

Adjust Contrast of Image **23-158**

Remove Impulse Noise from Image **23-161**

Draw Hough Lines on Image **23-164**

Construct Laplacian Pyramid Image **23-166**

Camera Calibration and SfM Examples

- “Visual SLAM with an RGB-D Camera” on page 1-2
- “Import Stereo Camera Parameters from ROS” on page 1-16
- “Import Camera Intrinsic Parameters from ROS” on page 1-20
- “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” on page 1-24
- “Visual Localization in a Parking Lot” on page 1-38
- “Stereo Visual SLAM for UAV Navigation in 3D Simulation” on page 1-44
- “Camera Calibration Using AprilTag Markers” on page 1-50
- “Configure Monocular Fisheye Camera” on page 1-67
- “Monocular Visual Simultaneous Localization and Mapping” on page 1-72
- “Structure From Motion From Two Views” on page 1-92
- “Stereo Visual Simultaneous Localization and Mapping” on page 1-102
- “Evaluating the Accuracy of Single Camera Calibration” on page 1-117
- “Measuring Planar Objects with a Calibrated Camera” on page 1-122
- “Depth Estimation From Stereo Video” on page 1-131
- “Structure From Motion From Multiple Views” on page 1-141
- “Uncalibrated Stereo Image Rectification” on page 1-149

calib

Visual SLAM with an RGB-D Camera

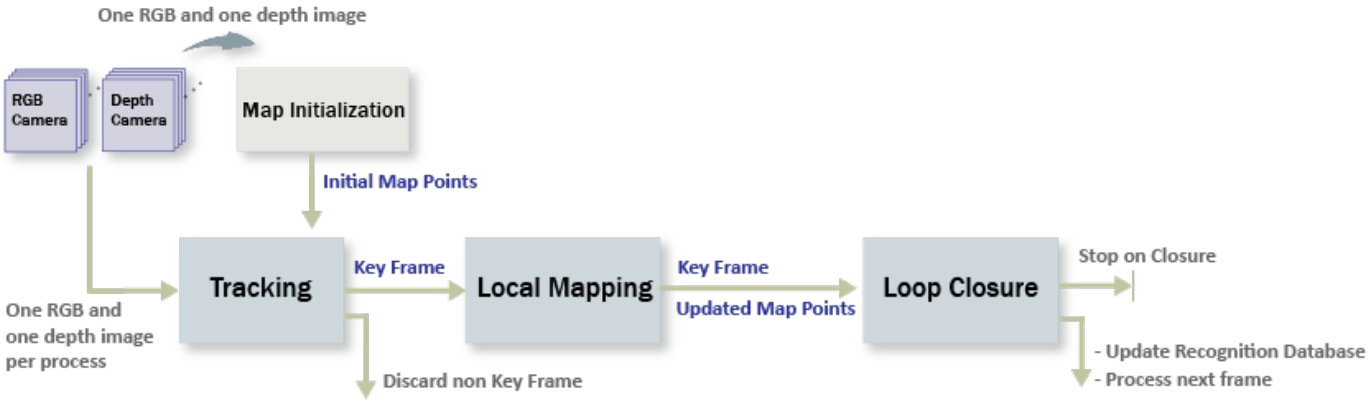
Visual simultaneous localization and mapping (vSLAM), refers to the process of calculating the position and orientation of a camera with respect to its surroundings, while simultaneously mapping the environment.

You can perform vSLAM using a monocular camera. However, the depth cannot be accurately calculated, and the estimated trajectory is unknown and drifts over time. To produce an initial map, which cannot be triangulated from the first frame, you must use multiple views of a monocular camera. A better, more reliable solution is to use an RGB-D camera, which is composed of one RGB color image and one depth image.

This example shows how to process RGB-D image data to build a map of an indoor environment and estimate the trajectory of the camera. The example uses a version of the ORB-SLAM2 [1] algorithm, which is feature-based and supports RGB-D cameras.

Overview of Processing Pipeline

The pipeline for RGB-D vSLAM is very similar to the monocular vSLAM pipeline in the "Monocular Visual Simultaneous Localization and Mapping" on page 1-72 example. The major difference is that in the **Map Initialization** stage, the 3-D map points are created from a pair of images consisting of one color image and one depth image instead of two frames of color images.



- **Map Initialization**: The initial 3-D world points can be constructed by extracting ORB feature points from the color image and then computing their 3-D world locations from the depth image. The color image is stored as the first key frame.
- **Tracking**: Once a map is initialized, the pose of the camera is estimated for each new RGB-D image by matching features in the color image to features in the last key frame.
- **Local Mapping**: If the current color image is identified as a key frame, new 3-D map points are computed from the depth image. 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 is from the TUM RGB-D benchmark [2]. You can download the data to a temporary folder using a web browser or by running the following code:

```
baseDownloadURL = 'https://vision.in.tum.de/rgbd/dataset/freiburg3/rgbd_dataset_freiburg3_long_o
dataFolder      = fullfile(tempdir, 'tum_rgbd_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
imageFolder = [dataFolder, 'rgbd_dataset_freiburg3_long_office_household/'];
```

Create two `imageDatastore` objects to store the color and depth images, respectively.

```
imgFolderColor = [imageFolder,'rgb/'];
imgFolderDepth = [imageFolder,'depth/'];
imdsColor      = imageDatastore(imgFolderColor);
imdsDepth      = imageDatastore(imgFolderDepth);
```

Note that the color and depth images are generated in an un-synchronized way in the dataset. Therefore, we need to associate color images to depth images based on the time stamp.

```
% Load time stamp data of color images
timeColor = helperImportTimestampFile([imageFolder, 'rgb.txt']);

% Load time stamp data of depth images
timeDepth = helperImportTimestampFile([imageFolder, 'depth.txt']);

% Align the time stamp
indexPairs = helperAlignTimestamp(timeColor, timeDepth);

% Select the synchronized image data
imdsColor     = subset(imdsColor, indexPairs(:, 1));
imdsDepth     = subset(imdsDepth, indexPairs(:, 2));

% Inspect the first RGB-D image
currFrameIdx  = 1;
currIcolor    = readimage(imdsColor, currFrameIdx);
currIdepth    = readimage(imdsDepth, currFrameIdx);
imshowpair(currIcolor, currIdepth, 'montage');
```

Map Initialization

The 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 the final SLAM result. Initial ORB feature points are extracted from the first color image using `helperDetectAndExtractFeatures` on page 1-0 . Their

corresponding 3-D world locations can be computed from the pixel coordinates of the feature points and the depth value using `helperReconstructFromRGBD` on page 1-0 .

```
% 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
focalLength    = [535.4, 539.2];    % in units of pixels
principalPoint = [320.1, 247.6];    % in units of pixels
imageSize      = size(currIcolor,[1,2]); % in pixels [mrows, ncols]
depthFactor    = 5e3;
intrinsics     = cameraIntrinsics(focalLength,principalPoint,imageSize);

% Detect and extract ORB features from the color image
scaleFactor = 1.2;
numLevels   = 8;
[currFeatures, currPoints] = helperDetectAndExtractFeatures(currIcolor, scaleFactor, numLevels);

initialPose = rigid3d();
[xyzPoints, validIndex] = helperReconstructFromRGBD(currPoints, currIdepth, intrinsics, initialPo
```

Initialize Place Recognition Database

Loop detection is performed using the bags-of-words approach. A visual vocabulary represented as a `bagOfFeatures` object is created offline with the ORB descriptors extracted from a large set of images in the dataset by calling:

```
bag = bagOfFeatures(imds,'CustomExtractor',
@helperORBFeatureExtractorFunction, 'TreeProperties', [5, 10],
'StrongestFeatures', 1);
```

where `imds` is an `imageDatastore` object storing the training images and `helperORBFeatureExtractorFunction` is the ORB feature extractor function. See “Image Retrieval with Bag of Visual Words” on page 17-110 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 ORB features.

```
% Load the bag of features data created offline
bofData         = load('bagOfFeaturesDataSLAM.mat');

% Initialize the place recognition database
loopDatabase    = invertedImageIndex(bofData.bof,"SaveFeatureLocations", false);

% Add features of the first key frame to the database
currKeyFrameId = 1;
addImageFeatures(loopDatabase, currFeatures, currKeyFrameId);
```

Data Management and Visualization

After the map is initialized using the first pair of color and depth image, you can use `imageviewset`, `worldpointset` and `helperViewDirectionAndDepth` to store the first key frames and the corresponding map points:

```
% 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(xyzPoints, 1));

% Add the first key frame
vSetKeyFrames = addView(vSetKeyFrames, currKeyFrameId, initialPose, 'Points', currPoints,...
    'Features', currFeatures.Features);

% Add 3-D map points
[mapPointSet, rgbdMapPointsIdx] = addWorldPoints(mapPointSet, xyzPoints);

% Add observations of the map points
mapPointSet = addCorrespondences(mapPointSet, currKeyFrameId, rgbdMapPointsIdx, validIndex);

% Visualize matched features in the first key frame
featurePlot = helperVisualizeMatchedFeaturesRGBD(currIcolor, currIdepth, currPoints(validIndex))
```

```
% Visualize initial map points and camera trajectory
xLim = [-4 4];
yLim = [-3 1];
zLim = [-1 6];
mapPlot  = helperVisualizeMotionAndStructure(vSetKeyFrames, mapPointSet, xLim, yLim, zLim);

% Show legend
showLegend(mapPlot);
```

Tracking

The tracking process is performed using every RGB-D image and determines when to insert a new key frame.

```
% ViewId of the last key frame
lastKeyFrameId    = currKeyFrameId;
```

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

% Indices of all the key frames in the input image sequence
addedFramesIdx    = lastKeyFrameIdx;

currFrameIdx      = 2;
isLoopClosed      = false;
```

Each frame is processed as follows:

1. ORB features are extracted for each new color image 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 using *Perspective-n-Point* algorithm, which estimates the pose of a calibrated camera given a set of 3-D points and their corresponding 2-D projections 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 `matchFeaturesInRadius.`
4. With 3-D to 2-D correspondences 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 `matchFeaturesInRadius` and refine the camera pose again using `bundleAdjustmentMotion`.
6. The last step of tracking is to decide if the current frame should be a new key frame. A frame is a key frame if both of the following conditions are satisfied:

- At least 20 frames have passed since the last key frame or the current frame tracks fewer than 100 map points or 25% of points tracked by the reference key frame.
- The map points tracked by the current frame are fewer than 90% of points tracked by the reference key frame.

If the current frame is to become a key frame, continue to the **Local Mapping** process. Otherwise, start **Tracking** for the next frame.

```
% Main loop
isLastFrameKeyFrame = true;
while ~isLoopClosed && currFrameIdx < numel(imdsColor.Files)

    currIcolor = readimage(imdsColor, currFrameIdx);
    currIdepth = readimage(imdsDepth, currFrameIdx);

    [currFeatures, currPoints]    = helperDetectAndExtractFeatures(currIcolor, scaleFactor, numLe

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

    if isempty(currPose) || numel(trackedMapPointsIdx) < 30
        currFrameIdx = currFrameIdx + 1;
        continue
    end
```

```
% Track the local map and 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 100 map points.
% 2. The map points tracked by the current frame are fewer than 90% of
%    points tracked by the reference key frame.
%
% localKeyFrameIds:   ViewId of the connected key frames of the current frame
numSkipFrames     = 20;
numPointsKeyFrame = 100;
[localKeyFrameIds, currPose, trackedMapPointsIdx, trackedFeatureIdx, isKeyFrame] = ...
    helperTrackLocalMap(mapPointSet, directionAndDepth, vSetKeyFrames, trackedMapPointsIdx,
    trackedFeatureIdx, currPose, currFeatures, currPoints, intrinsics, scaleFactor, numLevels
    isLastFrameKeyFrame, lastKeyFrameIdx, currFrameIdx, numSkipFrames, numPointsKeyFrame);

% Match feature points between the stereo images and get the 3-D world positions
[xyzPoints, validIndex] = helperReconstructFromRGBD(currPoints, currIdepth, ...
    intrinsics, currPose, depthFactor);

% Visualize matched features
updatePlot(featurePlot, currIcolor, currIdepth, currPoints(trackedFeatureIdx));

if ~isKeyFrame
    currFrameIdx = currFrameIdx + 1;
    isLastFrameKeyFrame = false;
    continue
else
    [untrackedFeatureIdx, ia] = setdiff(validIndex, trackedFeatureIdx);
    xyzPoints = xyzPoints(ia, :);
    isLastFrameKeyFrame = true;
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, trackedMapPointsIdx, trackedFeatureIdx, localKeyFrame

% Remove outlier map points that are observed in fewer than 3 key frames
if currKeyFrameId == 2
```

```
    triangulatedMapPointsIdx = [];
end

[mapPointSet, directionAndDepth, trackedMapPointsIdx] = ...
    helperCullRecentMapPoints(mapPointSet, directionAndDepth, trackedMapPointsIdx, triangulat
    rgbdMapPointsIdx);

% Add new map points computed from disparity
[mapPointSet, rgbdMapPointsIdx] = addWorldPoints(mapPointSet, xyzPoints);
mapPointSet = addCorrespondences(mapPointSet, currKeyFrameId, rgbdMapPointsIdx, ...
    untrackedFeatureIdx);

% Create new map points by triangulation
minNumMatches = 10;
minParallax   = 0.35;
[mapPointSet, vSetKeyFrames, triangulatedMapPointsIdx, rgbdMapPointsIdx] = helperCreateNewMap
    mapPointSet, vSetKeyFrames, currKeyFrameId, intrinsics, scaleFactor, minNumMatches, minPa
    untrackedFeatureIdx, rgbdMapPointsIdx);

% Update view direction and depth
directionAndDepth = update(directionAndDepth, mapPointSet, vSetKeyFrames.Views, ...
    [trackedMapPointsIdx; triangulatedMapPointsIdx; rgbdMapPointsIdx], true);

% Local bundle adjustment
[mapPointSet, directionAndDepth, vSetKeyFrames, triangulatedMapPointsIdx, rgbdMapPointsIdx] =
    helperLocalBundleAdjustmentStereo(mapPointSet, directionAndDepth, vSetKeyFrames, ...
    currKeyFrameId, intrinsics, triangulatedMapPointsIdx, rgbdMapPointsIdx);

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

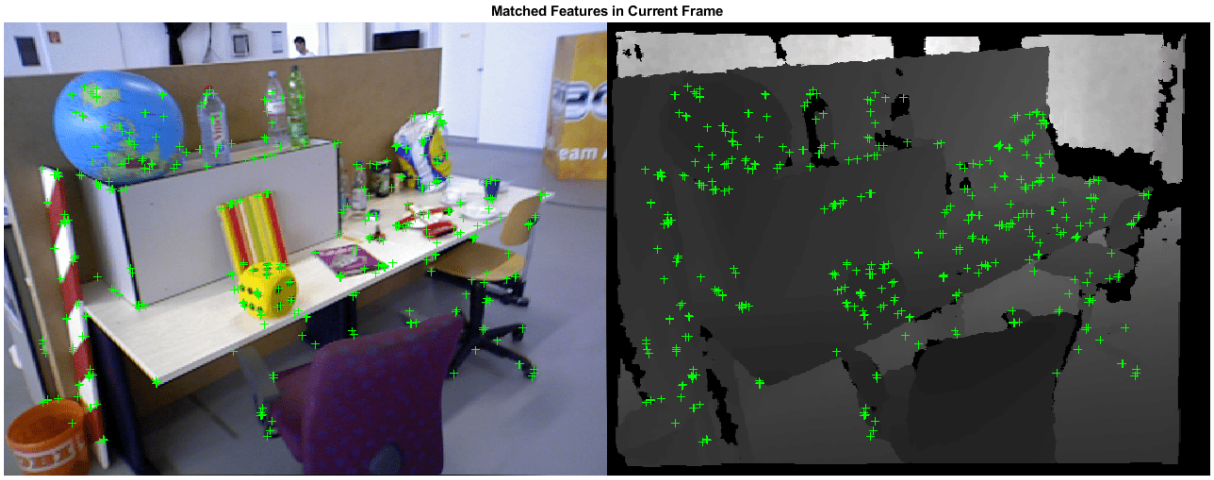
Loop Closure

The loop closure detection step takes the current key frame processed by the local mapping process and tries to detect and close the loop. 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, use `estimateGeometricTransform3D` to compute the relative pose between the loop candidate frame and the current key frame. The relative pose represents a 3-D rigid transformation stored in a rigid3d object. Then add the loop connection with the relative pose and update `mapPointSet` and `vSetKeyFrames`.

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

    % Minimum number of feature matches of loop edges
    loopEdgeNumMatches = 120;

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

    if isDetected
        % Add loop closure connections
        maxDistance = 0.1;
```

```
            [isLoopClosed, mapPointSet, vSetKeyFrames] = helperAddLoopConnectionsStereo(...
                mapPointSet, vSetKeyFrames, validLoopCandidates, currKeyFrameId, ...
                currFeatures, loopEdgeNumMatches, maxDistance);
        end
    end

    % If no loop closure is detected, add current features into the database
    if ~isLoopClosed
        addImageFeatures(loopDatabase,  currFeatures, currKeyFrameId);
    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: 9 and 94
Loop edge added between keyframe: 8 and 94
```

Finally, apply pose graph optimization over the essential graph in `vSetKeyFrames` to correct the drift. 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.

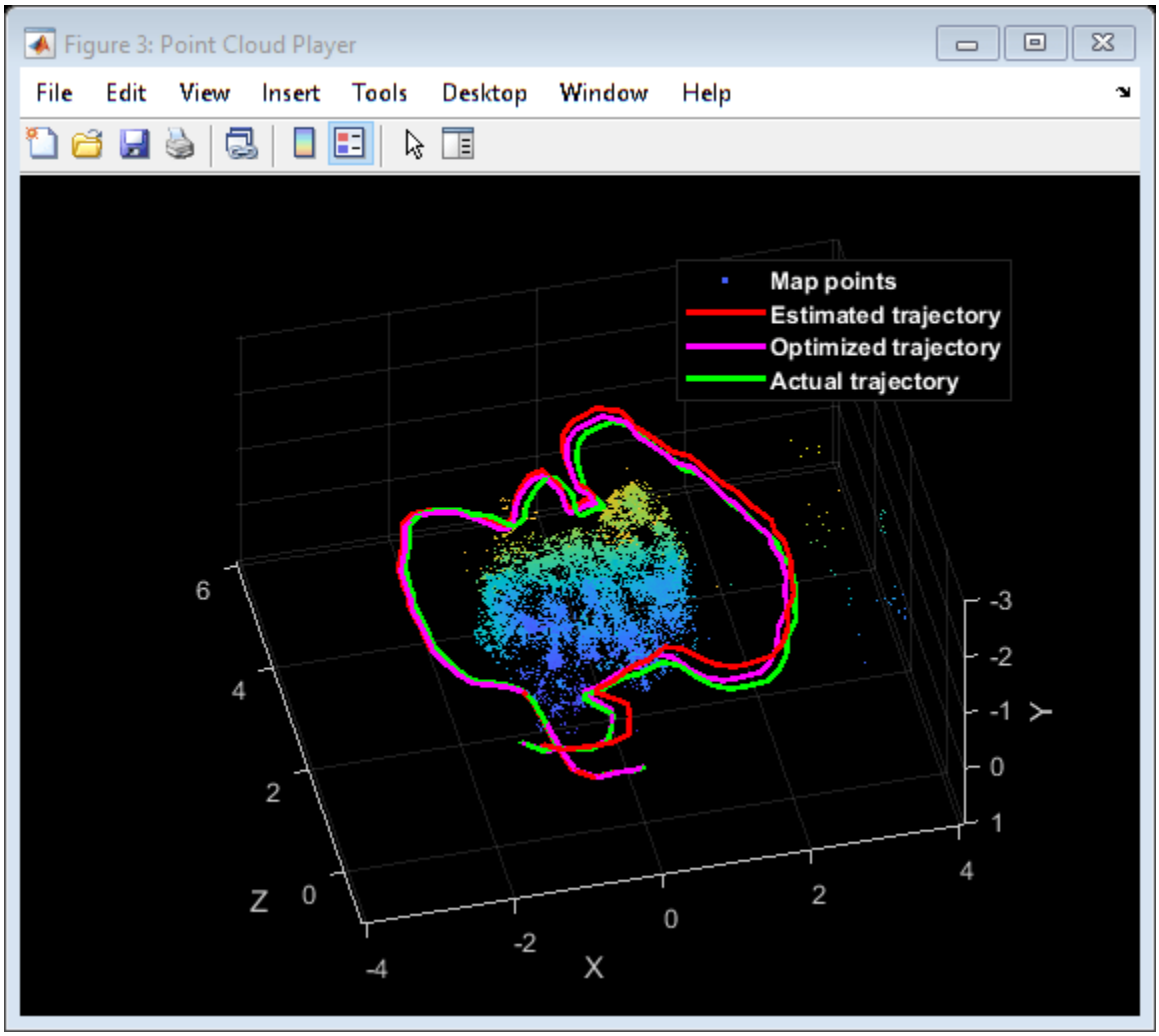
```
% Optimize the poses
minNumMatches      = 50;
vSetKeyFramesOptim = optimizePoses(vSetKeyFrames, minNumMatches, 'Tolerance', 1e-16);

% 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. 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('orbslamGroundTruth.mat');
gTruth     = gTruthData.gTruth;

% Plot the actual camera trajectory
plotActualTrajectory(mapPlot, gTruth(indexPairs(addedFramesIdx, 1)), optimizedPoses);

% Show legend
showLegend(mapPlot);
```



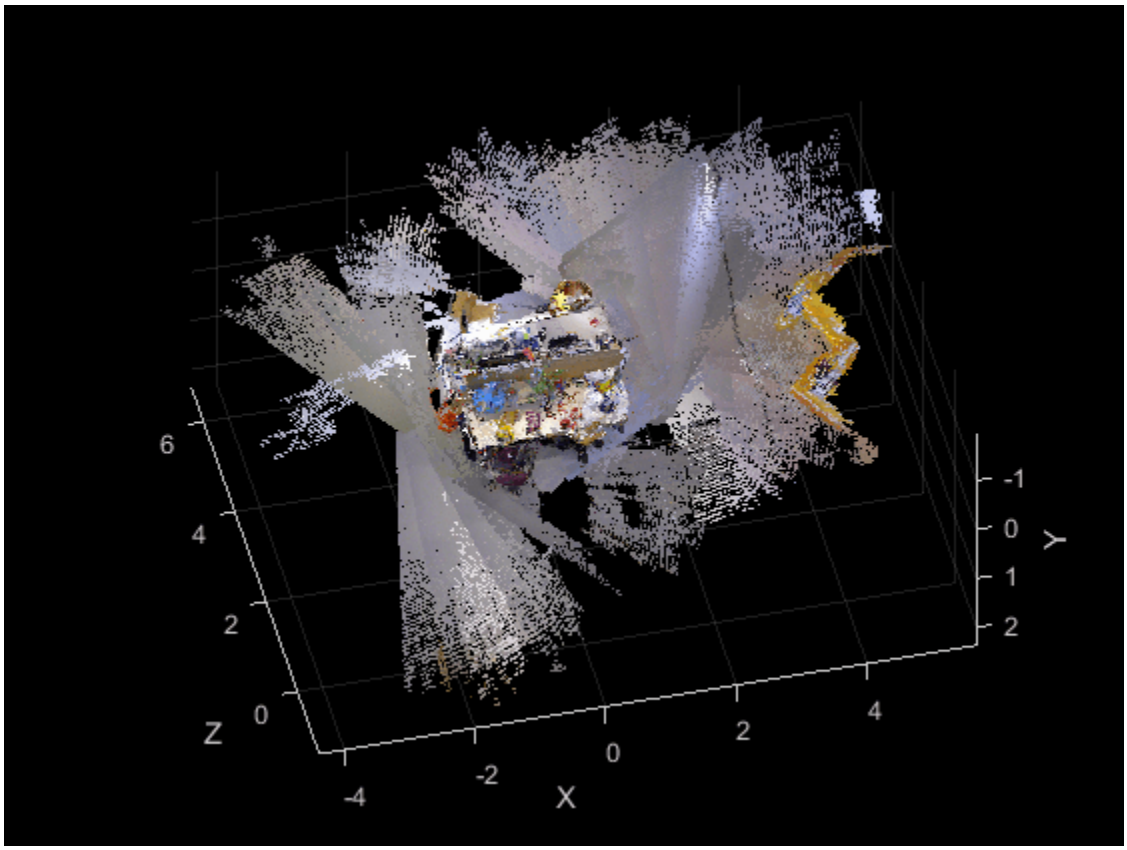
```
% Evaluate tracking accuracy
helperEstimateTrajectoryError(gTruth(indexPairs(addedFramesIdx, 1)), optimizedPoses);
```

```
Absolute RMSE for key frame trajectory (m): 0.12806
```

Dense Reconstruction from Depth Image

Given the refined camera poses, you can reproject all the valid image points in the associated depth images back to the 3-D space to perform dense reconstruction.

```
% Create an array of pointCloud objects to store the world points constructed
% from the key frames
ptClouds =  repmat(pointCloud(zeros(1, 3)), numel(addedFramesIdx), 1);

% Ignore image points at the boundary
offset = 40;
[X, Y] = meshgrid(offset:2:imageSize(2)-offset, offset:2:imageSize(1)-offset);

for i = 1: numel(addedFramesIdx)
    Icolor = readimage(imdsColor, addedFramesIdx(i));
    Idepth = readimage(imdsDepth, addedFramesIdx(i));

    [xyzPoints, validIndex] = helperReconstructFromRGBD([X(:), Y(:)], ...
        Idepth, intrinsics, optimizedPoses.AbsolutePose(i), depthFactor);

    colors = zeros(numel(X), 1, 'like', Icolor);
    for j = 1:numel(X)
        colors(j, 1:3) = Icolor(Y(j), X(j), :);
    end
    ptClouds(i) = pointCloud(xyzPoints, 'Color', colors(validIndex, :));
end

% Concatenate the point clouds
pointCloudsAll = pccat(ptClouds);

figure
pcshow(pointCloudsAll,'VerticalAxis', 'y', 'VerticalAxisDir', 'down');
xlabel('X')
ylabel('Y')
zlabel('Z')
```



Supporting Functions

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

`helperImportTimestampFile` Import time stamp file

```
function timestamp = helperImportTimestampFile(filename)

% Input handling
dataLines = [4, Inf];

%% Set up the Import Options and import the data
opts = delimitedTextImportOptions("NumVariables", 2);

% Specify range and delimiter
opts.DataLines = dataLines;
opts.Delimiter = " ";

% Specify column names and types
opts.VariableNames = ["VarName1", "Var2"];
opts.SelectedVariableNames = "VarName1";
opts.VariableTypes = ["double", "string"];

% Specify file level properties
opts.ExtraColumnsRule = "ignore";
opts.EmptyLineRule = "read";
opts.ConsecutiveDelimitersRule = "join";
opts.LeadingDelimitersRule = "ignore";
```

```
% Specify variable properties
opts = setvaropts(opts, "Var2", "WhitespaceRule", "preserve");
opts = setvaropts(opts, "Var2", "EmptyFieldRule", "auto");

% Import the data
data = readtable(filename, opts);

% Convert to output type
timestamp = table2array(data);
end
```

`helperAlignTimestamp` align time stamp of color and depth images.

```
function indexPairs = helperAlignTimestamp(timeColor, timeDepth)
idxDepth = 1;
indexPairs = zeros(numel(timeColor), 2);
for i = 1:numel(timeColor)
    for j = idxDepth : numel(timeDepth)
        if abs(timeColor(i) - timeDepth(j)) < 1e-4
            idxDepth = j;
            indexPairs(i, :) = [i, j];
            break
        elseif timeDepth(j) - timeColor(i) > 1e-3
            break
        end
    end
end
indexPairs = indexPairs(indexPairs(:,1)>0, :);
end
```

`helperDetectAndExtractFeatures` detect and extract and ORB features from the image.

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

numPoints = 1000;

% 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
```

`helperReconstructFromRGBD` reconstruct scene from color and depth image.

```
function [xyzPoints, validIndex] = helperReconstructFromRGBD(points, ...
    depthMap, intrinsics, currPose, depthFactor)

% Find the corresponding world point of the matched feature points
if ~isnumeric(points)
    points  = points.Location;
end
```

```
xyzPoints   = zeros(size(points, 1), 3);
maxRange    = 5; % In meters

for i = 1:size(points, 1)
    Z  = double(depthMap(floor(points(i, 2)), floor(points(i, 1)))) / depthFactor;
    XY = (points(i, :) - intrinsics.PrincipalPoint) ./ intrinsics.FocalLength * Z;
    xyzPoints(i, :)= [XY, Z];
end

isPointValid = xyzPoints(:, 3) > 0 & xyzPoints(:, 3) < maxRange;
xyzPoints    = xyzPoints(isPointValid, :);
xyzPoints    = xyzPoints * currPose.Rotation + currPose.Translation;
validIndex   = find(isPointValid);
end
```

`helperCullRecentMapPoints` cull recently added map points.

```
function [mapPointSet, directionAndDepth, mapPointsIdx] = ...
    helperCullRecentMapPoints(mapPointSet, directionAndDepth, mapPointsIdx, newPointIdx, rgbdMapI
outlierIdx = setdiff([newPointIdx; rgbdMapPointsIndices], 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)

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
```

References

[1] Mur-Artal, Raul, and Juan D. Tardós. "ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras." *IEEE Transactions on Robotics* 33, no. 5 (2017): 1255-1262.

Import Stereo Camera Parameters from ROS

The ROS camera calibration package estimates stereo camera parameters using the OpenCV camera calibration tools [1]. After calibrating a stereo camera in ROS, you can export its camera parameters to an INI file using the camera calibration parser. To use the calibrated stereo camera with Computer Vision Toolbox™ functions, such as `rectifyStereoImages`, you must read the camera parameters from the INI file and convert them into a `stereoParameters` object using `stereoParametersFromOpenCV`.

Note: The `stereoParametersFromOpenCV` function supports importing stereo camera parameters for only those pinhole camera models that use the `ROS plumb-bob` distortion model.

Read Stereo Camera Parameters from ROS INI File

Read the stereo camera parameters stored in stereo`Params.ini` using the helper function `helperReadINI`.

```
stereoParamsINI = helperReadINI("stereoParams.ini");
```

Compute Baseline Parameters of Stereo Camera

The baseline parameters of a stereo camera describe the relative translation and rotation of the two cameras in the stereo camera pair. The relative rotation and translation of camera 2 with respect to camera 1 is required to create the `stereoParameters` object using `stereoParametersFromOpenCV`. You can compute these from the rectification and projection matrices read from the ROS INI file [2].

Extract the two camera parameters from the `stereoParams` structure.

```
cameraParams1 = stereoParamsINI.narrow_stereo_left;
cameraParams2 = stereoParamsINI.narrow_stereo_right;
```

Extract the translation of camera 2 relative to camera 1 from the last column of the projection matrix.

```
translationOfCamera2 = cameraParams2.projection(:,end);
```

The rotation of camera 2 relative to camera 1, R_{21}, is derived from the rectification matrices of the stereo pair R_1 and R_2. The rectification matrices are the rotation matrices that align the camera coordinate system to the ideal stereo image plane such that epipolar lines in both stereo images are parallel. Compute the rotation of camera 2 relative to camera 1 as $R_{21} = R_2 * R_1^T$.

```
rotationOfCamera2 = cameraParams2.rectification*cameraParams1.rectification';
```

Create `stereoParameters` Object using `stereoParametersFromOpenCV`

Extract the intrinsic matrices and distortion coefficients of the two cameras from the `stereoParams` structure.

```
intrinsicMatrix1 = cameraParams1.camera_matrix;
intrinsicMatrix2 = cameraParams2.camera_matrix;

distortionCoefficients1 = cameraParams1.distortion;
distortionCoefficients2 = cameraParams2.distortion;
```

Obtain the image size from the `image` field of the `stereoParams` structure.

```
imageSize = [stereoParamsINI.image.height stereoParamsINI.image.width];
```

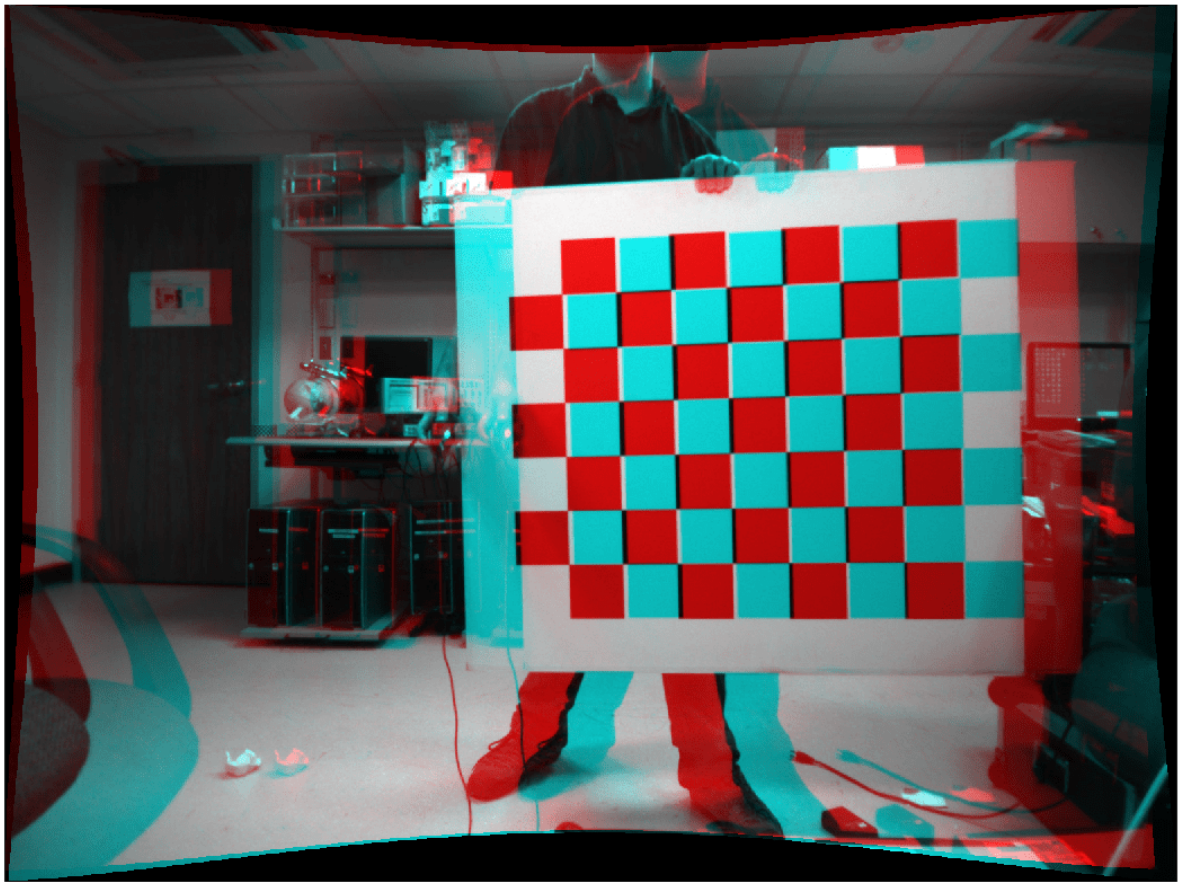
Use `stereoParametersFromOpenCV` to create a `stereoParameters` object from the ROS stereo camera parameters.

```
stereoParametersObj = stereoParametersFromOpenCV(intrinsicMatrix1, ...
    distortionCoefficients1,intrinsicMatrix2,distortionCoefficients2, ...
    rotationOfCamera2,translationOfCamera2,imageSize);
```

Rectify Pair of Stereo Images

Use the imported stereo parameters with `rectifyStereoImages` to rectify an image pair captured using the calibrated stereo camera.

```
% Load the image pair.
imageDir = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo');
leftImages = imageDatastore(fullfile(imageDir,'left'));
rightImages = imageDatastore(fullfile(imageDir,'right'));
I1 = readimage(leftImages,1);
I2 = readimage(rightImages,1);

% Rectify the image pair.
[J1,J2] = rectifyStereoImages(I1,I2,stereoParametersObj,OutputView='full');

% Display the results.
figure
J = stereoAnaglyph(J1,J2);
imshow(J)
```



Supporting Functions

helperReadINI

The `helperReadINI` function reads the camera parameters from its input INI file that has been exported from ROS.

```
function cameraParams = helperReadINI(filename)
% helperReadINI reads a ROS INI file, filename, and returns a structure with
% these fields: image, <camera_name1>, <camera_name2>. image is a
% structure describing the height and width of the image captured by the
% cameras of the stereo pair. The fields <camera_name1> and <camera_name2>
% are structures named after the camera names present in the INI file, and they contain
% these fields: camera_matrix, distortion, rectification_matrix,
% and projection_matrix. These fields are stored in the INI file with their
% values placed in a new line followed by their name.

    f = fopen(filename,'r');
    sectionName = '';

    while ~feof(f)
        % Read line from file.
        line = fgetl(f);
```

```
        % Trim leading and trailing whitespaces.
        line = strtrim(line);

        if isempty(line) || line(1)=='#'
            % Skip empty line and comments.
            continue
        elseif line(1) == '[' && line(end) == ']'
            % Identify section names and continue reading.
            sectionName = line(2:end-1);
            sectionName = strrep(sectionName,'/','_');
            continue
        end

        % Replace blankspaces with underscores to create valid MATLAB variable
        % name.
        name = line;
        name(name == ' ') = '_';

        % Read the value data in upcoming lines.
        value = [];
        while ~feof(f)
            line = fgetl(f);
            line = strtrim(line);

            if isempty(line)
                % A empty line indicates end of value data.
                break
            elseif line(1)=='#'
                % Skip comment lines.
                continue
            end
            line = str2num(line); %#ok
            value = [value; line]; %#ok
        end

        % Store post-processed value.
        if isempty(sectionName)
            cameraParams.(name) = value;
        else
            cameraParams.(sectionName).(name) = value;
        end
    end

    fclose(f);
end
```

References

[1] http://wiki.ros.org/camera_calibration

[2] http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/CameraInfo.html

Import Camera Intrinsic Parameters from ROS

The ROS camera calibration package estimates camera intrinsic parameters using the OpenCV camera calibration tools [1]. After calibrating a camera in ROS, you can import its intrinsic parameters to a YAML file using the camera calibration parser in ROS. To use the calibrated camera with Computer Vision Toolbox™ functions, such as `undistortImage`, you must read the camera parameters from the YAML file and then convert them into a `cameraIntrinsics` object using `cameraIntrinsicsFromOpenCV`.

Note: The `cameraIntrinsicsFromOpenCV` function supports importing camera intrinsic parameters for only those pinhole camera models that use the `ROS plumb-bob` distortion model.

Read Camera Intrinsic Parameters from a ROS YAML File

Read the camera parameters stored in `cameraParams.yaml` using the helper function `helperReadYAML`.

```
intrinsicsParams = helperReadYAML('cameraParams.yaml');
```

Create `cameraIntrinsics` Object Using `cameraIntrinsicsFromOpenCV`

Use the `cameraIntrinsicsFromOpenCV` function to create a `cameraIntrinsics` object from the camera matrix and the distortion coefficients.

```
imageSize = [intrinsicsParams.image_height intrinsicsParams.image_width];
intrinsicMatrix = intrinsicsParams.camera_matrix;
distortionCoefficients = intrinsicsParams.distortion_coefficients;

intrinsicsObj = cameraIntrinsicsFromOpenCV(intrinsicMatrix,distortionCoefficients,imageSize);
```

Undistort Image

Use the imported camera intrinsics with `undistortImage` to undistort an image captured using the calibrated camera.

```
% Load the captured image.
imageName = fullfile(toolboxdir('vision'),'visiondata','calibration','stereo','left','left01.png
I = imread(imageName);

% Undistort the image.
J = undistortImage(I,intrinsicsObj,'OutputView','full');

% Display the result.
figure
montage({I,J})
```

Supporting Functions

helperReadYAML

The `helperReadYAML` function reads the monocular camera parameters from the input YAML file that was exported from ROS.

```
function cameraParams = helperReadYAML(filename)
% helperReadYAML reads a ROS YAML file, filename, and returns a structure
% with these fields: image_width, image_height, camera_name,
% camera_matrix, distortion_model, distortion_coefficients,
% rectification_matrix, and projection_matrix. These fields are stored
% in the YAML file colon separated from their values in different lines.

    f = fopen(filename,'r');
    stringFields = {'camera_name','distortion_model'};

    while ~feof(f)

        [name,value,isEmptyLine] = helperReadYAMLLine(f);
        if isEmptyLine
            continue
        end

        if ~isempty(value)
            % Convert all values to numbers except for known string
            % fields.
            if ~any(contains(name, stringFields))
                value = str2num(value); %#ok
            end
        else
            % An empty value in ROS YAML files indicates a matrix in
            % upcoming lines. Read the matrix from the upcoming lines.
            value = helperReadYAMLMatrix(f);
        end

        % Store post-processed value.
        cameraParams.(name) = value;
```

```
    end

    fclose(f);
end
```

helperReadYAMLMatrix

The `helperReadYAMLMatrix` function reads the rows, columns and data fields of a matrix in the ROS YAML file.

```
function matrix = helperReadYAMLMatrix(f)
%   helperReadYAMLMatrix reads a matrix from the ROS YAML file. A matrix in
%   a ROS YAML file has three fields: rows, columns and data. rows and col
%   describe the matrix size. data is a continguous array of the matrix
%   elements in row-major order. This helper function assumes the presence
%   of all three fields of a matrix to return the correct matrix.

    numRows = 0;
    numCols = 0;
    data = [];

    % Read numRows, numCols and matrix data.
    while ~feof(f)
        [name,value,isEmptyLine] = helperReadYAMLLine(f);

        if isEmptyLine
            continue
        end

        switch name
            case 'rows'
                numRows = str2num(value); %#ok
            case 'cols'
                numCols = str2num(value); %#ok
            case 'data'
                data    = str2num(value); %#ok

                % Terminate the while loop as data is the last
                % field of a matrix in the ROS YAML file.
                break
            otherwise
                % Terminate the while loop if any other field is
                % encountered.
                break
        end
    end

    if numel(data) == numRows*numCols
        % Reshape the matrix using row-major order.
        matrix = reshape(data,[numCols numRows])';
    end
end
```

helperReadYAMLLine

The `helperReadYAMLLine` function reads a line of a ROS YAML file.

```
function [name,value,isEmptyLine] = helperReadYAMLLine(f)
```

```
    % Read line from file.
    line = fgetl(f);

    % Trim leading and trailing whitespaces.
    line = strtrim(line);

    if isempty(line) || line(1)=='#'
        % Empty line or comment.
        name = '';
        value = '';
        isEmptyLine = true;
    else
        % Split the line to get name and value.
        c = strsplit(line,':');
        assert(length(c)==2,'Unexpected file format')

        name = c{1};
        value = strtrim(c{2}); % Trim leading whitespace.
        isEmptyLine = false;
    end
end
```

References

[1] http://wiki.ros.org/camera_calibration

Develop Visual SLAM Algorithm Using Unreal Engine Simulation

This example shows how to develop a visual Simultaneous Localization and Mapping (SLAM) algorithm using image data obtained from the Unreal Engine® simulation environment.

Visual SLAM is the process of calculating the position and orientation of a camera with respect to its surroundings while simultaneously mapping the environment. Developing a visual SLAM algorithm and evaluating its performance in varying conditions is a challenging task. One of the biggest challenges is generating the ground truth of the camera sensor, especially in outdoor environments. The use of simulation enables testing under a variety of scenarios and camera configurations while providing precise ground truth.

This example demonstrates the use of Unreal Engine simulation to develop a visual SLAM algorithm for either a monocular or a stereo camera in a parking scenario. For more information about the implementation of the visual SLAM pipelines, see the “Monocular Visual Simultaneous Localization and Mapping” on page 1-72 example and the “Stereo Visual Simultaneous Localization and Mapping” on page 1-102 example.

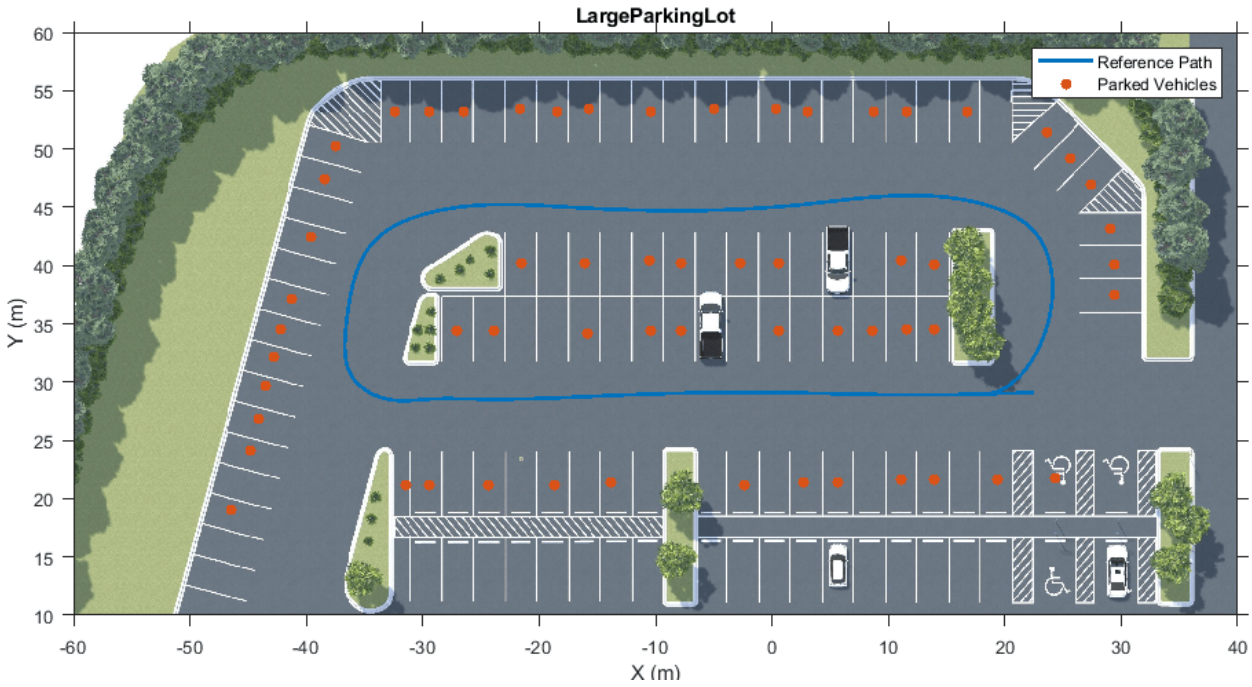
Set Up Scenario in Simulation Environment

Use the Simulation 3D Scene Configuration block to set up the simulation environment. Select the built-in Large Parking Lot scene, which contains several parked vehicles. The visual SLAM algorithm matches features across consecutive images. To increase the number of potential feature matches, you can use the Parked Vehicles subsystem to add more parked vehicles to the scene. To specify the parking poses of the vehicles, use the `helperAddParkedVehicle` function. If you select a more natural scene, the presence of additional vehicles is not necessary. Natural scenes usually have enough texture and feature variety suitable for feature matching.

You can follow the “Select Waypoints for Unreal Engine Simulation” (Automated Driving Toolbox) example to interactively select a sequence of parking locations. You can use the same approach to select a sequence of waypoints and generate a reference trajectory for the ego vehicle. This example uses a recorded reference trajectory and parked vehicle locations.

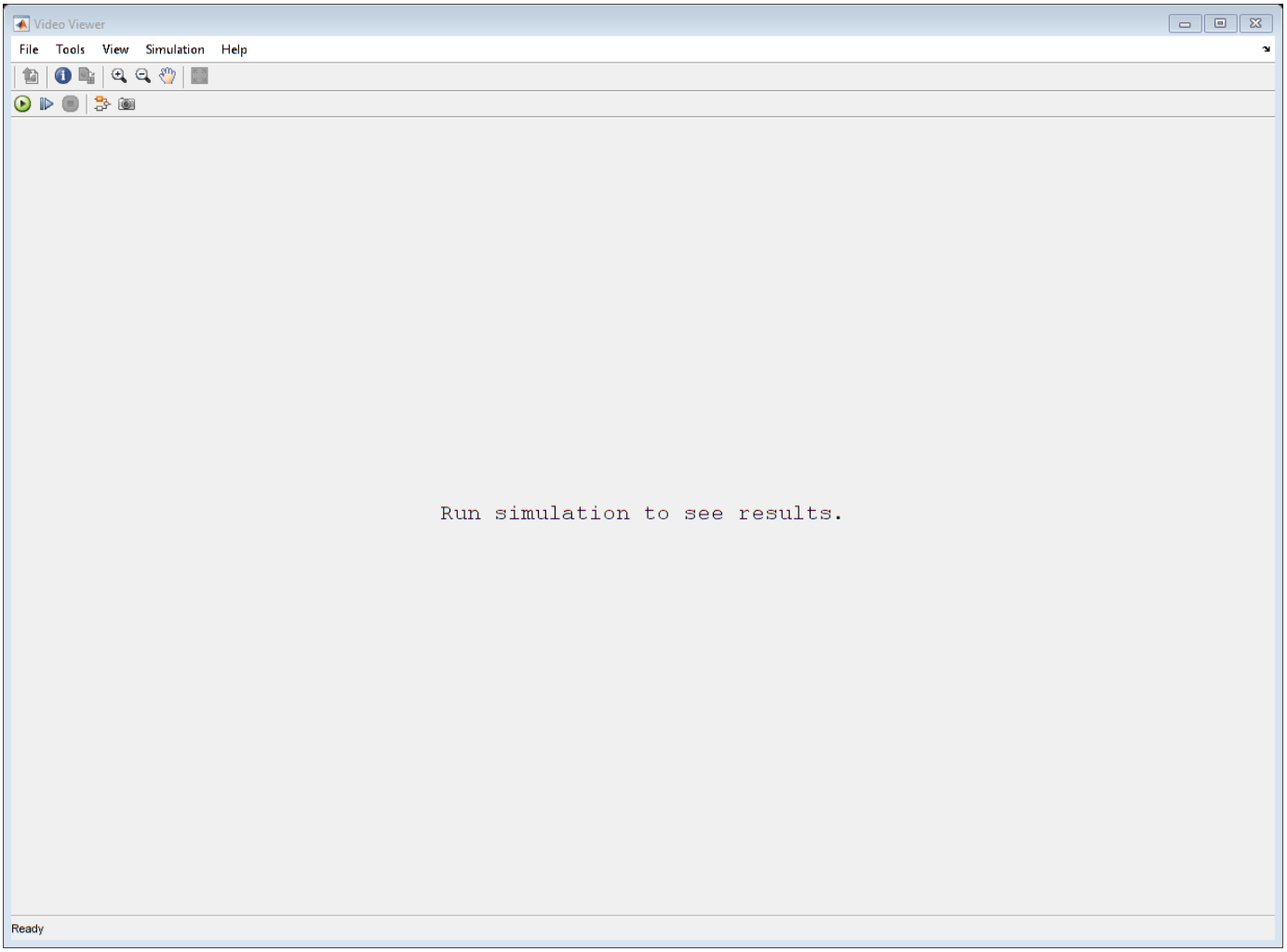
```
% Load reference path
data = load('parkingLotReferenceData.mat');

% Set reference trajectory of the ego vehicle
refPosesX = data.refPosesX;
refPosesY = data.refPosesY;
refPosesT = data.refPosesT;

% Set poses of the parked vehicles
parkedPoses = data.parkedPoses;

% Display the reference path and the parked vehicle locations
sceneName = 'LargeParkingLot';
hScene = figure;
helperShowSceneImage(sceneName);
hold on
plot(refPosesX(:,2), refPosesY(:,2), 'LineWidth', 2, 'DisplayName', 'Reference Path');
scatter(parkedPoses(:,1), parkedPoses(:,2), [], 'filled', 'DisplayName', 'Parked Vehicles');
xlim([-60 40])
ylim([10 60])
```

```
hScene.Position = [100, 100, 1000, 500]; % Resize figure
legend
hold off
```

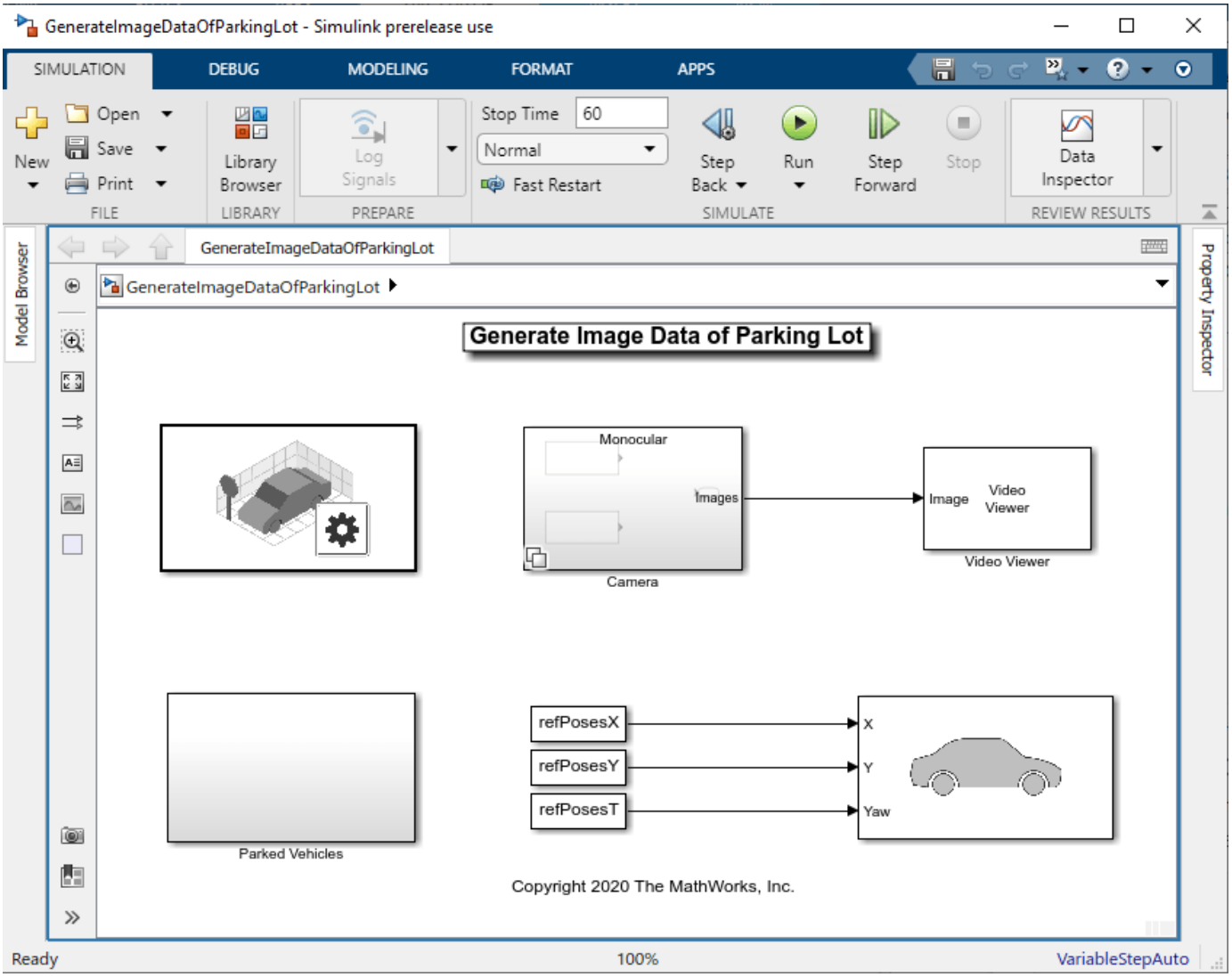


Open the model and add parked vehicles

```
modelName = 'GenerateImageDataOfParkingLot';
open_system(modelName);
```




```
helperAddParkedVehicles(modelName, parkedPoses);
```



Set Up Ego Vehicle and Camera Sensor

Set up the ego vehicle moving along the specified reference path by using the Simulation 3D Vehicle with Ground Following block. The Camera Variant Subsystem contains two configurations of camera sensors: monocular and stereo. In both configurations, the camera is mounted on the vehicle roof center. You can use the Camera Calibrator or Stereo Camera Calibrator app to estimate intrinsics of the actual camera that you want to simulate. This example shows the monocular camera workflow first followed by the stereo camera workflow.

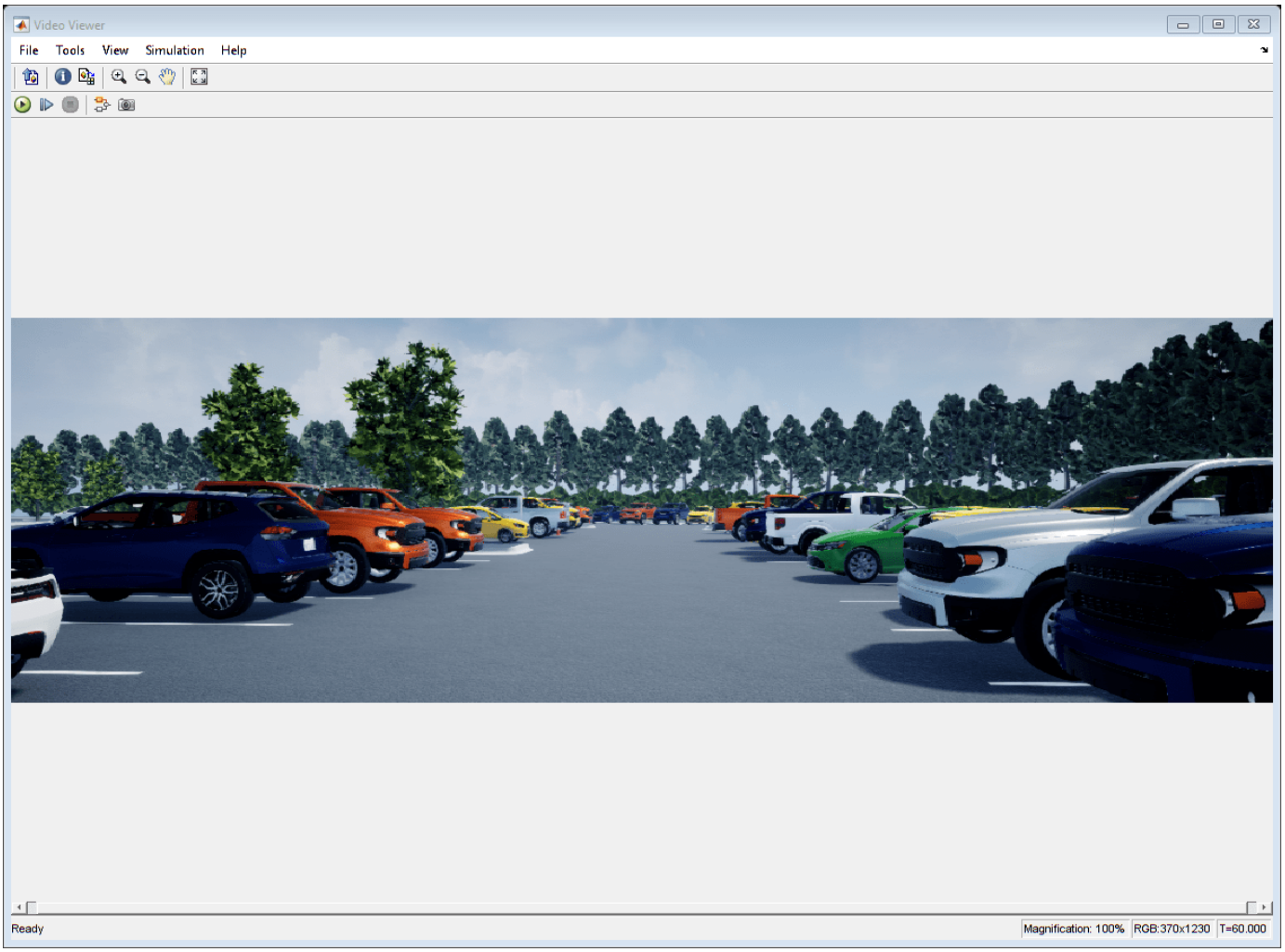
```
% Select monocular camera
useMonoCamera = 1;

% Inspect the monocular camera
open_system([modelName, '/Camera/Monocular']);

% Camera intrinsics
focalLength    = [700, 700];  % specified in units of pixels
principalPoint = [600, 180];  % in pixels [x, y]
```

```
imageSize      = [370, 1230]; % in pixels [mrows, ncols]
intrinsics     = cameraIntrinsics(focalLength, principalPoint, imageSize);
```

Visualize and Record Sensor Data

Run the simulation to visualize and record sensor data. Use the Video Viewer block to visualize the image output from the camera sensor. Use the To Workspace block to record the ground truth location and orientation of the camera sensor.

```
close(hScene)

if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" + char(174) + " Windows" + cl
end

% Run simulation
simOut = sim(modelName);
```




```
% Extract camera images as an imageDatastore
imds = helperGetCameraImages(simOut);
```

```
% Extract ground truth as an array of rigid3d objects
gTruth = helperGetSensorGroundTruth(simOut);
```

Develop Monocular Visual SLAM Algorithm Using Recorded Data

Use the images to evaluate the monocular visual SLAM algorithm. The function `helperVisualSLAM` implements the monocular ORB-SLAM pipeline:

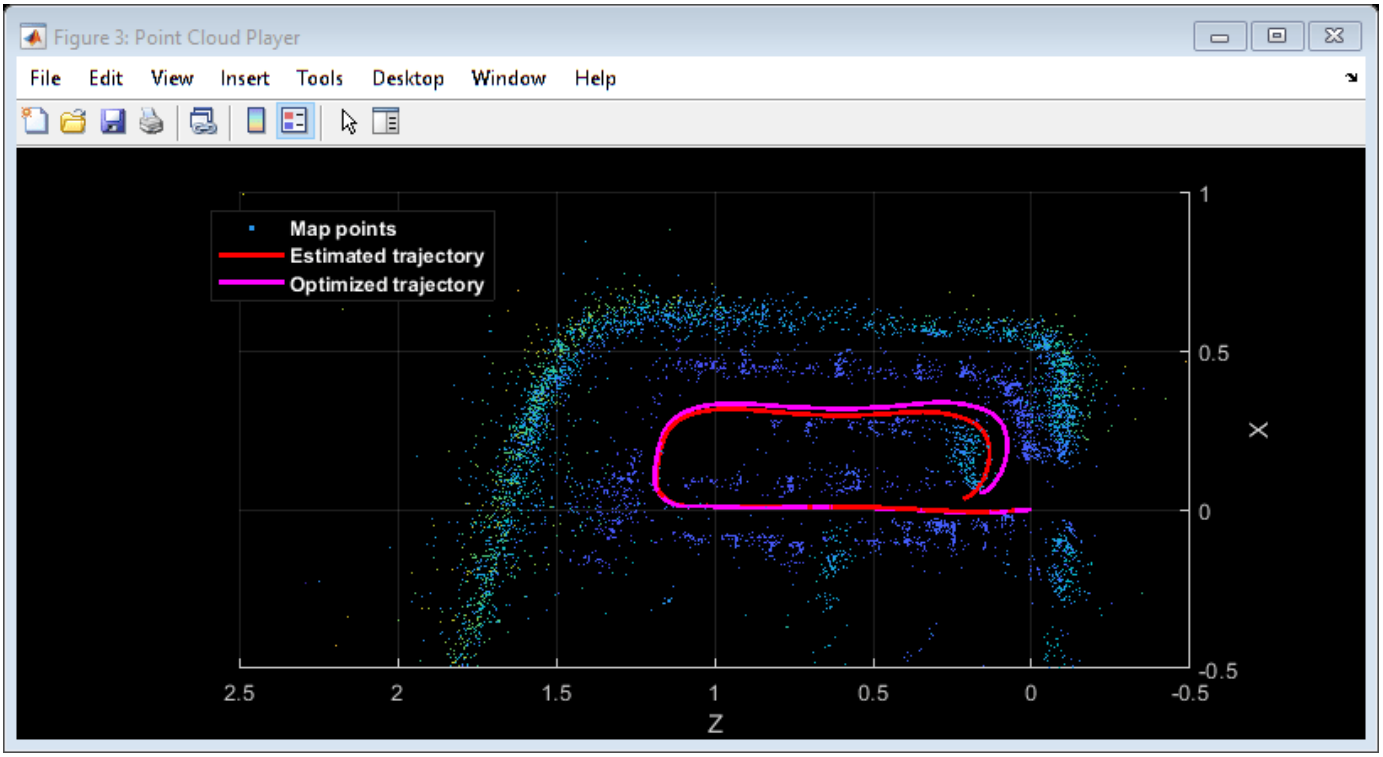
- **Map Initialization**: ORB-SLAM starts by initializing the map of 3-D points from two images. Use `relativeCameraPose` to compute the relative pose based on 2-D ORB feature correspondences and `triangulate` to compute the 3-D map points. The two frames are stored in an `imageviewset` object as key frames. The 3-D map points and their correspondences to the key frames are stored in a `worldpointset` object.
- **Tracking**: Once a map is initialized, for each new image, the function `helperTrackLastKeyFrame` estimates the camera pose by matching features in the current frame to features in the last key frame. The function `helperTrackLocalMap` refines the estimated camera pose 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, `bundleAdjustment` 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 using the `optimizePoseGraph` (Navigation Toolbox) function.

For the implementation details of the algorithm, see the “Monocular Visual Simultaneous Localization and Mapping” on page 1-72 example.

```
[mapPlot, optimizedPoses, addedFramesIdx] = helperVisualSLAM(imds, intrinsics);
```

```
Map initialized with frame 1 and frame 3
```

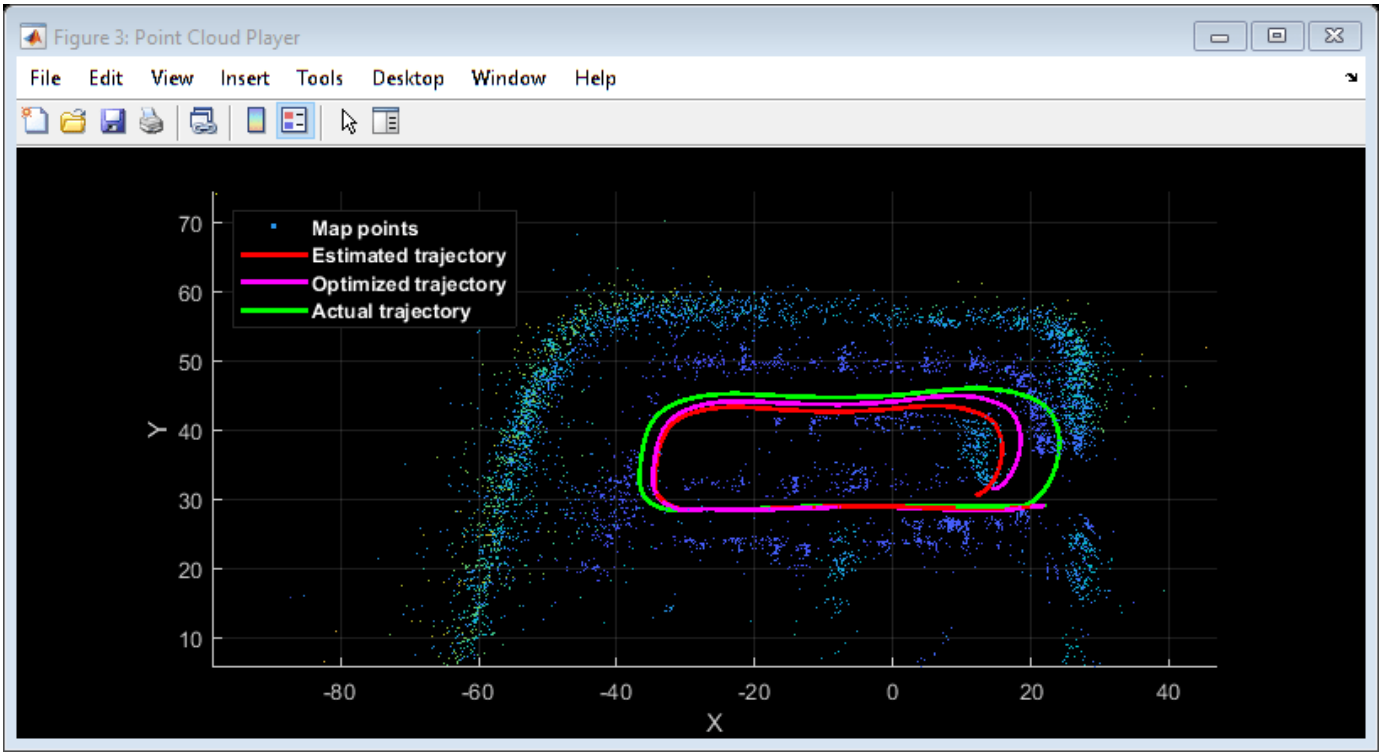
```
Loop edge added between keyframe: 5 and 104
Loop edge added between keyframe: 4 and 104
```



Evaluate Against Ground Truth

You can evaluate the optimized camera trajectory against the ground truth obtained from the simulation. Since the images are generated from a monocular camera, the trajectory of the camera can only be recovered up to an unknown scale factor. You can approximately compute the scale factor from the ground truth, thus simulating what you would normally obtain from an external sensor.

```
% Plot the camera ground truth trajectory
scaledTrajectory = plotActualTrajectory(mapPlot, gTruth(addedFramesIdx), optimizedPoses);

% Show legend
showLegend(mapPlot);
```



You can also calculate the root mean square error (RMSE) of trajectory estimates.

```
helperEstimateTrajectoryError(gTruth(addedFramesIdx), scaledTrajectory);
```

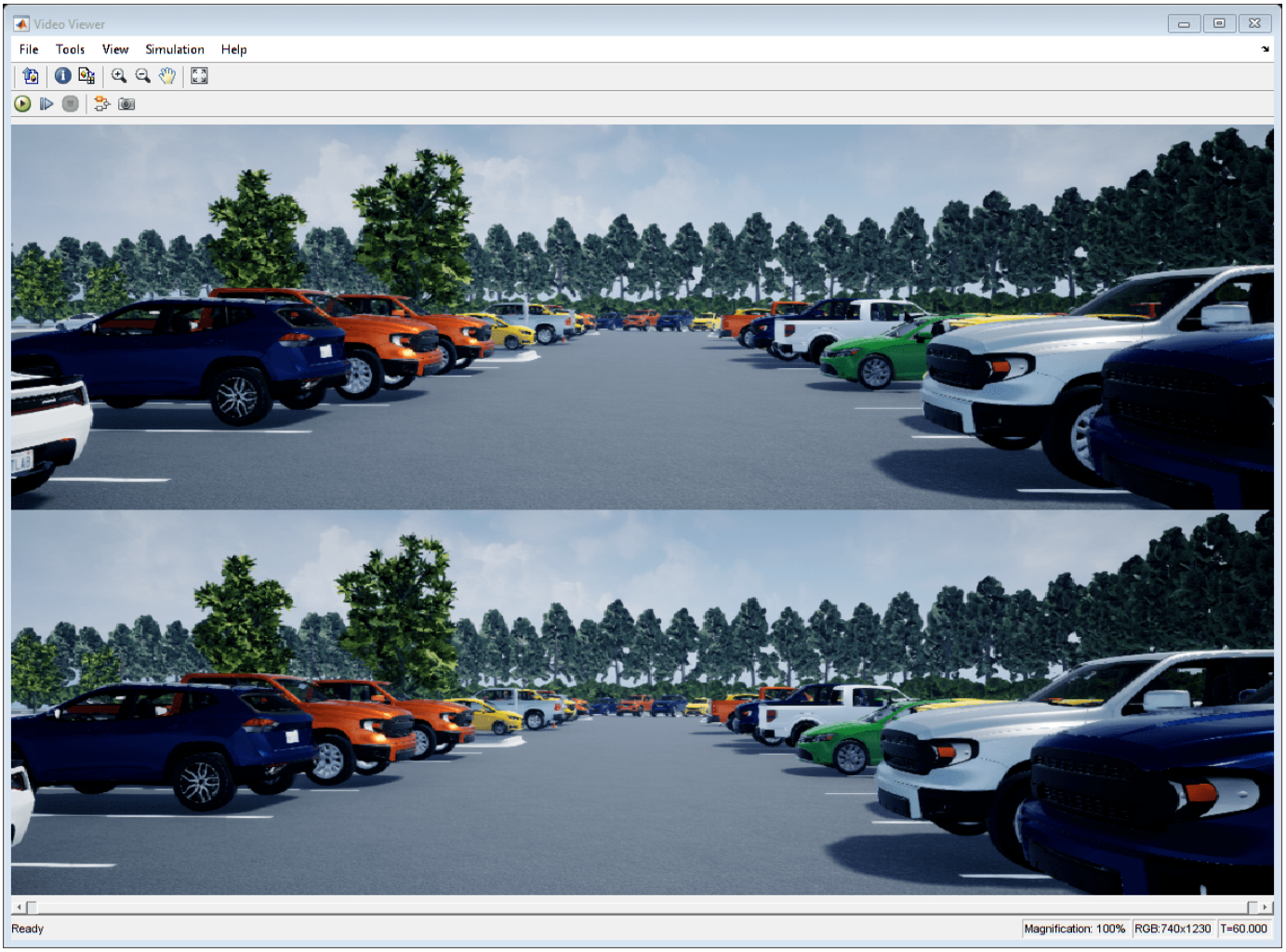
```
Absolute RMSE for key frame trajectory (m): 2.758
```

Stereo Visual SLAM Algorithm

In a monocular visual SLAM algorithm, depth cannot be accurately determined using a single camera. The scale of the map and of the estimated trajectory is unknown and drifts over time. Additionally, because map points often cannot be triangulated from the first frame, bootstrapping the system requires multiple views to produce an initial map. Using a stereo camera solves these problems and provides a more reliable visual SLAM solution. The function `helperVisualSLAMStereo` implements the stereo visual SLAM pipeline. The key difference from the monocular pipeline is that at the map initialization stage, the stereo pipeline creates 3-D map points from a pair of stereo images of the same frame, instead of creating them from two images of different frames. For the implementation details of the algorithm, see the “Stereo Visual Simultaneous Localization and Mapping” on page 1-102 example.

```
% Select stereo camera
useMonoCamera = 0;

% Inspect the stereo camera
open_system([modelName, '/Camera/Stereo']);
snapnow;

% Set stereo camera baseline
baseline = 0.5; % In meters

% Construct the reprojection matrix for 3-D reconstruction
```

```
reprojectionMatrix = [1, 0, 0, -principalPoint(1);
    0, 1, 0, -principalPoint(2);
    0, 0, 0, focalLength(1);
    0, 0, 1/baseline, 0];

% Maximum disparity in stereo images
maxDisparity = 48;

% Run simulation
simOut = sim(modelName);
```



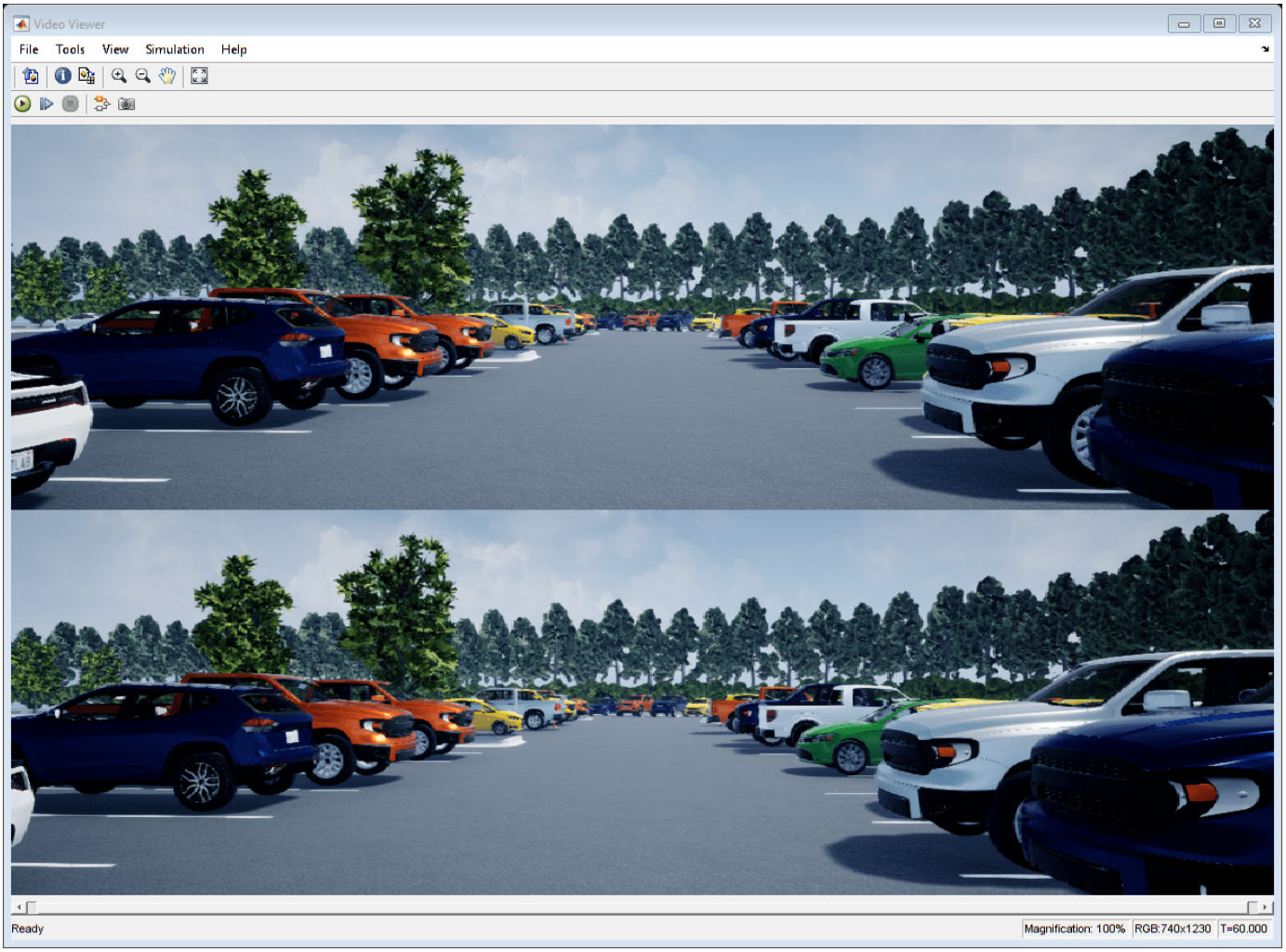
```
snapnow;
```

Extract Stereo Images

```
[imdsLeft, imdsRight] = helperGetCameraImagesStereo(simOut);

% Extract ground truth as an array of rigid3d objects
gTruth = helperGetSensorGroundTruth(simOut);
```

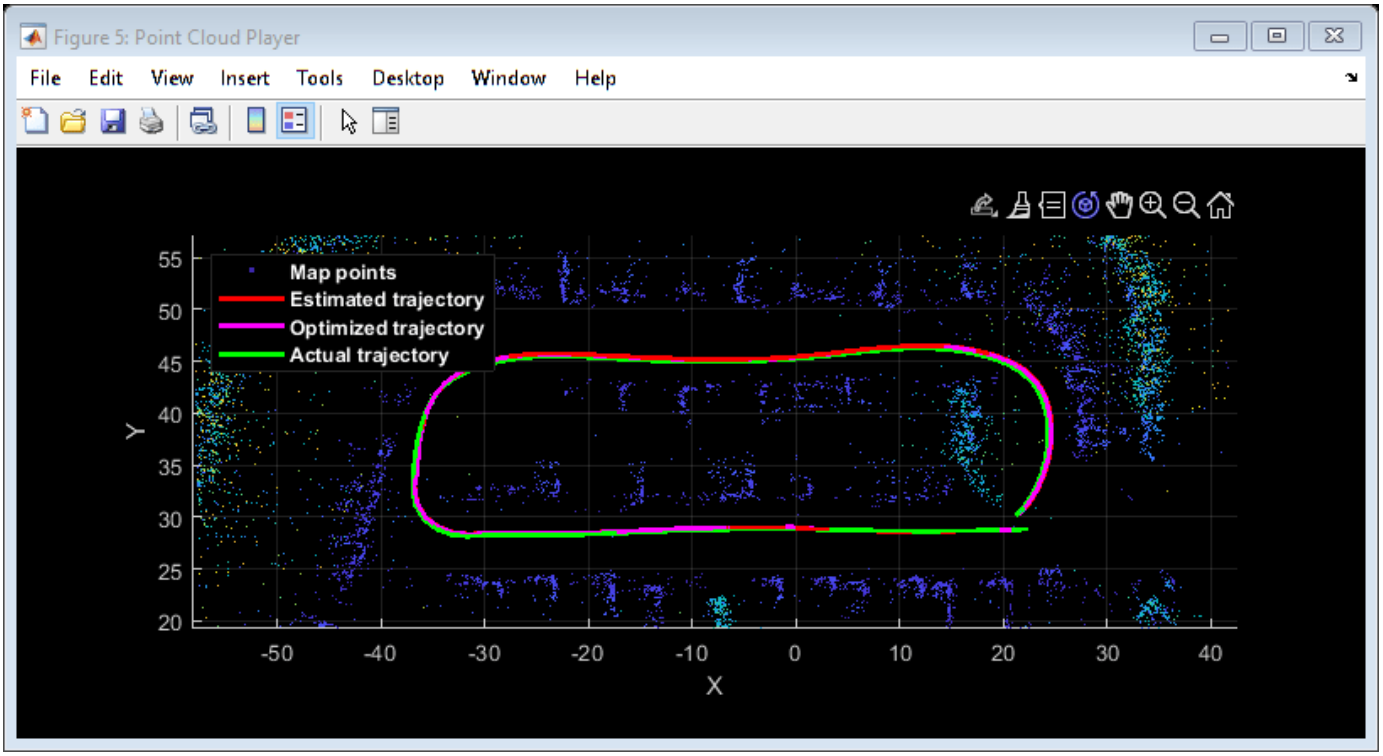
Run the stereo visual SLAM algorithm

```
[mapPlot, optimizedPoses, addedFramesIdx] = helperVisualSLAMStereo(imdsLeft, imdsRight, intrinsic
```



```
Loop edge added between keyframe: 3 and 98
Loop edge added between keyframe: 2 and 98
```

```
% Plot the camera ground truth trajectory
optimizedTrajectory = plotActualTrajectory(mapPlot, gTruth(addedFramesIdx));

% Show legend
showLegend(mapPlot);
```




```
% Calculate the root mean square error (RMSE) of trajectory estimates
helperEstimateTrajectoryError(gTruth(addedFramesIdx), optimizedTrajectory);
```

```
Absolute RMSE for key frame trajectory (m): 0.30292
```

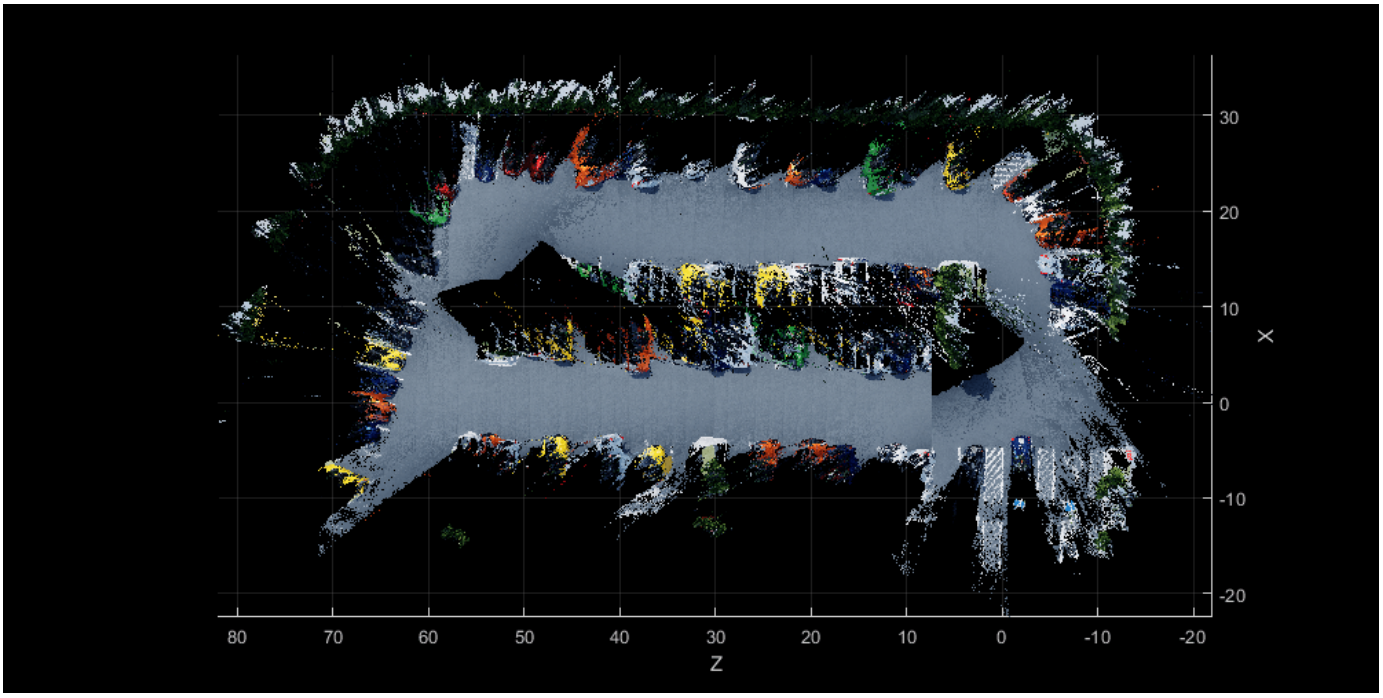
Compared with the monocular visual SLAM algorithm, the stereo visual SLAM algorithm produces a more accurate estimation of the camera trajectory.

Dense Reconstruction from Stereo Images

Given the refined camera poses, you can perform dense reconstruction from the stereo images corresponding to the key frames.

```
pointCloudsAll = helperDenseReconstructFromStereo(imdsLeft, imdsRight, ...
    imageSize, addedFramesIdx, optimizedPoses, maxDisparity, reprojectionMatrix);

% Visualize the scene
figure('Position', [1100 600 1000 500]);
ax = pcshow(pointCloudsAll,'VerticalAxis', 'y', 'VerticalAxisDir', 'down');
xlabel('X')
```

```
ylabel('Y')
zlabel('Z')

% Display bird's eye view of the parking lot
view(ax, [0 -1 0]);
camroll(ax, 90);
```



Close model and figures.

```
close_system(modelName, 0);
close all
```

Supporting Functions

helperGetCameraImages Get camera output

```
function imds = helperGetCameraImages(simOut)
% Save image data to a temporary folder
dataFolder   = fullfile(tempdir, 'parkingLotImages', filesep);
folderExists = exist(dataFolder, 'dir');
if ~folderExists
    mkdir(dataFolder);
end

files = dir(dataFolder);
if numel(files) < 3
    numFrames = numel(simOut.images.Time);
    for i = 3:numFrames % Ignore the first two frames
        img = squeeze(simOut.images.Data(:,:,:,i));
        imwrite(img, [dataFolder, sprintf('%04d', i-2), '.png'])
    end
end
```

```
% Create an imageDatastore object to store all the images
imds = imageDatastore(dataFolder);
end
```

helperGetCameraImagesStereo Get camera output

```
function [imdsLeft, imdsRight] = helperGetCameraImagesStereo(simOut)
% Save image data to a temporary folder
dataFolderLeft   = fullfile(tempdir, 'parkingLotStereoImages', filesep, 'left', filesep);
dataFolderRight  = fullfile(tempdir, 'parkingLotStereoImages', filesep, 'right', filesep);
folderExists     = exist(dataFolderLeft, 'dir');
if ~folderExists
    mkdir(dataFolderLeft);
    mkdir(dataFolderRight);
end

files = dir(dataFolderLeft);
if numel(files) < 3
    numFrames = numel(simOut.imagesLeft.Time);
    for i = 3:numFrames % Ignore the first two frames
        imgLeft = squeeze(simOut.imagesLeft.Data(:,:,:,i));
        imwrite(imgLeft, [dataFolderLeft, sprintf('%04d', i-2), '.png'])

        imgRight = squeeze(simOut.imagesRight.Data(:,:,:,i));
        imwrite(imgRight, [dataFolderRight, sprintf('%04d', i-2), '.png'])
    end
end

% Use imageDatastore objects to store the stereo images
imdsLeft  = imageDatastore(dataFolderLeft);
imdsRight = imageDatastore(dataFolderRight);
end
```

helperGetSensorGroundTruth Save the sensor ground truth

```
function gTruth = helperGetSensorGroundTruth(simOut)
numFrames = numel(simOut.location.Time);
gTruth = repmat(rigid3d, numFrames-2, 1);
for i = 1:numFrames-2 % Ignore the first two frames
    gTruth(i).Translation = squeeze(simOut.location.Data(:, :, i+2));
    % Ignore the roll and the pitch rotations since the ground is flat
    yaw = double(simOut.orientation.Data(:, 3, i+2));
    gTruth(i).Rotation = [cos(yaw), sin(yaw), 0; ...
        -sin(yaw), cos(yaw), 0; ...
        0, 0, 1];
end
end
```

helperEstimateTrajectoryError Calculate the tracking error

```
function rmse = helperEstimateTrajectoryError(gTruth, scaledLocations)
gLocations      = vertcat(gTruth.Translation);

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

helperDenseReconstructFromStereo Perform dense reconstruction from stereo images with known camera poses

```
function pointCloudsAll = helperDenseReconstructFromStereo(imdsLeft, imdsRight, ...
    imageSize, addedFramesIdx, optimizedPoses, maxDisparity, reprojectionMatrix)

ptClouds =  repmat(pointCloud(zeros(1, 3)), numel(addedFramesIdx), 1);

for i = 1: numel(addedFramesIdx)
    I1 = readimage(imdsLeft, addedFramesIdx(i));
    I2 = readimage(imdsRight, addedFramesIdx(i));
    disparityMap = disparitySGM(im2gray(I1), im2gray(I2), 'DisparityRange', [0, maxDisparity],'U
    xyzPoints  = reconstructScene(disparityMap, reprojectionMatrix);

    % Ignore the upper half of the images which mainly show the sky
    xyzPoints(1:100, :, :) = NaN;

    xyzPoints  = reshape(xyzPoints, [imageSize(1)*imageSize(2), 3]);

    validIndex = xyzPoints(:, 3) > 0 & xyzPoints(:, 3) < 40/reprojectionMatrix(4, 3);

    xyzPoints  = xyzPoints(validIndex, :);
    colors = reshape(I1, [imageSize(1)*imageSize(2), 3]);
    colors = colors(validIndex, :);

    currPose = optimizedPoses.AbsolutePose(i);
    xyzPoints  = xyzPoints * currPose.Rotation + currPose.Translation;
    ptCloud = pointCloud(xyzPoints, 'Color', colors);
    ptClouds(i) = pcdownsample(ptCloud, 'random', 0.2);
end

% Concatenate the point clouds
pointCloudsAll = pccat(ptClouds);
end
```

Visual Localization in a Parking Lot

This example shows how to develop a visual localization system using synthetic image data from the Unreal Engine® simulation environment.

It is a challenging task to obtain ground truth for evaluating the performance of a localization algorithm in different conditions. Virtual simulation in different scenarios is a cost-effective method to obtain the ground truth in comparison with more expensive approaches such as using high-precision inertial navigation systems or differential GPS. The use of simulation enables testing under a variety of scenarios and sensor configurations. It also enables a rapid algorithm development, and provides precise ground truth.

This example uses the Unreal Engine simulation environment from Epic Games® to develop and evaluate a visual localization algorithm in a parking lot scenario.

Overview

Visual localization is the process of estimating the camera pose for a captured image relative to a visual representation of a known scene. It is a key technology for applications such as augmented reality, robotics, and automated driving. Compared with a “Implement Visual SLAM in MATLAB” on page 11-8, visual localization assumes that a map of the environment is known and does not require 3-D reconstruction or loop closure detection. The pipeline of visual localization includes the following:

- **Map Loading:** Load the pre-built map 3-D map containing world point positions and the 3-D to 2-D correspondences between the map points and the key frames. Additionally. for each key frame, load the feature descriptors corresponding to the 3-D map points.
- **Global Initialization:** Extract features from the first image frame and match them with the features corresponding to all the 3-D map points. After getting the 3-D to 2-D correspondences, estimate the camera pose of the first frame in the world coordinate by solving a Perspective-n-Point (PnP) problem. Refine the pose using motion-only bundle adjustment. The key frame that shares the most covisible 3-D map points with the first frame is identified as the reference key frame.
- **Tracking:** Once the first frame is localized, for each new frame, match features in the new frame with features in the reference key frame that have known 3-D world points. Estimate and refine the camera pose using the same approach as in Global Initialization step. The camera pose can be further refined by tracking the features associated with nearby key frames.

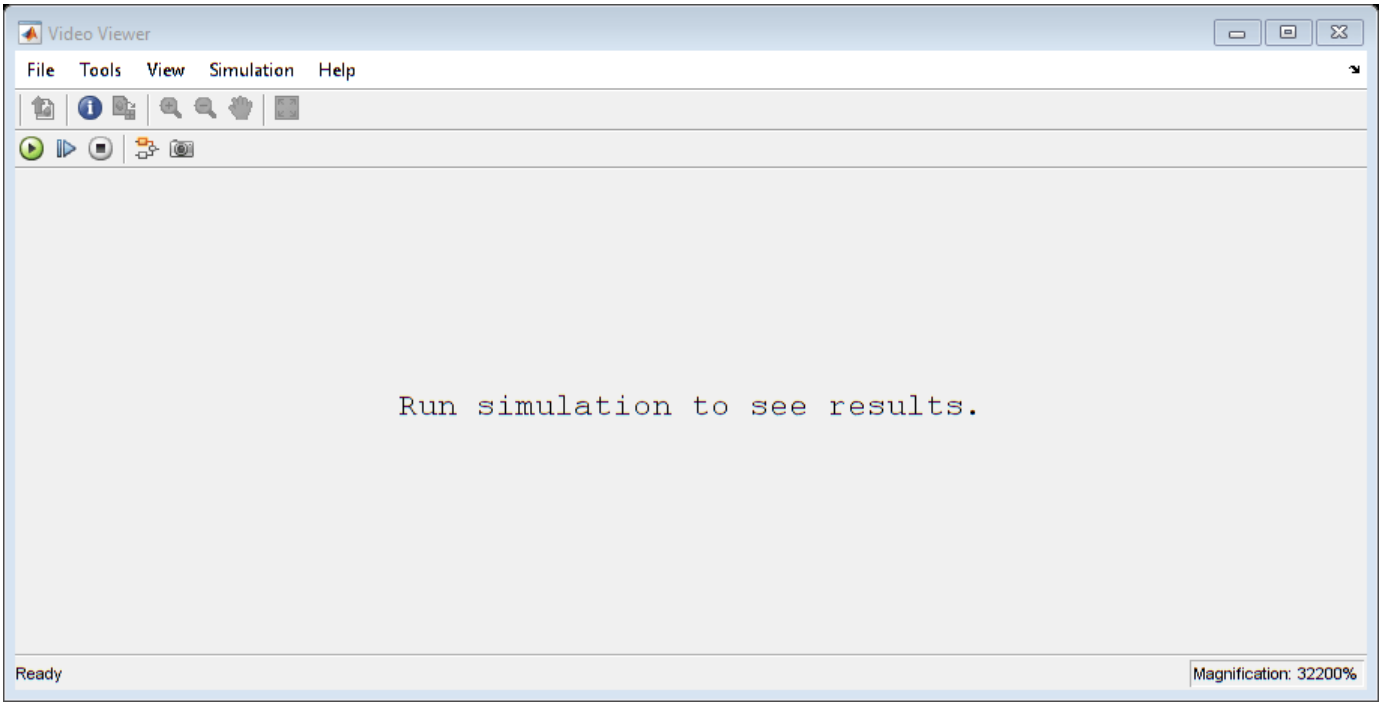
Create Scene

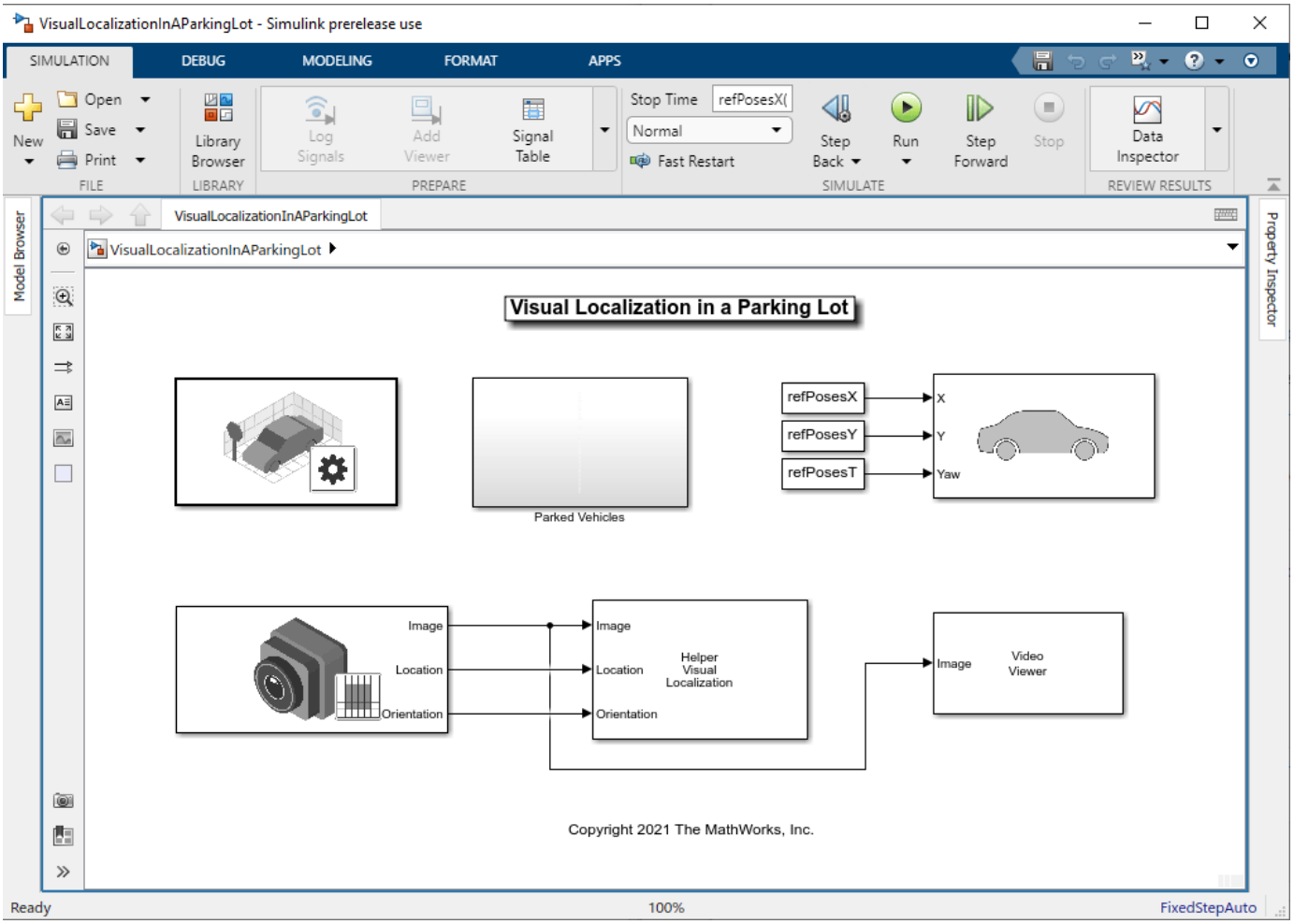
Guiding a vehicle into a parking spot is a challenging maneuver that relies on accurate localization. The `VisualLocalizationInAParkingLot` model simulates a visual localization system in the parking lot scenario used in the “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” (Automated Driving Toolbox) example.

- The Simulation 3D Scene Configuration (Automated Driving Toolbox) block sets up the Large Parking Lot scene. The `Parked Vehicles` subsystem adds parked cars into the parking lot.
- The Simulation 3D Vehicle with Ground Following (Automated Driving Toolbox) block controls the motion of the ego vehicle.
- The Simulation 3D Camera (Automated Driving Toolbox)block models a monocular camera fixed at the center of the vehicle's roof. You can use the Camera Calibrator app to estimate intrinsics of the actual camera that you want to simulate.

- The `Helper Visual Localization` MATLAB System block implements the visual localization algorithm. The initial camera pose with respect to the map is estimated using the `helperGlobalInitialization` function. The subsequent camera poses are estimated using the `helperTrackingRefKeyFrame` function and refined using the `helperTrackLocalKeyFrames` function. This block also provides a visualization of the estimated camera trajectory in the pre-built map. You can specify the pre-built map data and the camera intrinsic parameters in the block dialog.

```
% Open the model
modelName = 'VisualLocalizationInAParkingLot';
open_system(modelName);
```





Load Map Data

The pre-built map data is generated using the stereo camera in the “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” (Automated Driving Toolbox) example. The data consists of three objects that are commonly used to manage image and map data for visual SLAM:

- **`vSetKeyFrame`:** an `imageviewset` object storing the camera poses of key frames and the associated feature points for each 3-D map point in `mapPointSet`.
- **`mapPointSet`**: a `worldpointset` object storing the 3-D map point locations and the correspondences between the 3-D points and 2-D feature points across key frames. The 3-D map points provide a sparse representation of the environment.
- **`directionAndDepth`:** a `helperViewDirectionAndDepth` object storing view direction and depth of each map point in `mapPointSet`.

```
% Load pre-built map data
mapData = load("prebuiltMapData.mat")
```

```
mapData = struct with fields:
        vSetKeyFrames: [1×1 imageviewset]
          mapPointSet: [1×1 worldpointset]
    directionAndDepth: [1×1 helperViewDirectionAndDepth]
```

Set Up Ego Vehicle and Camera Sensor

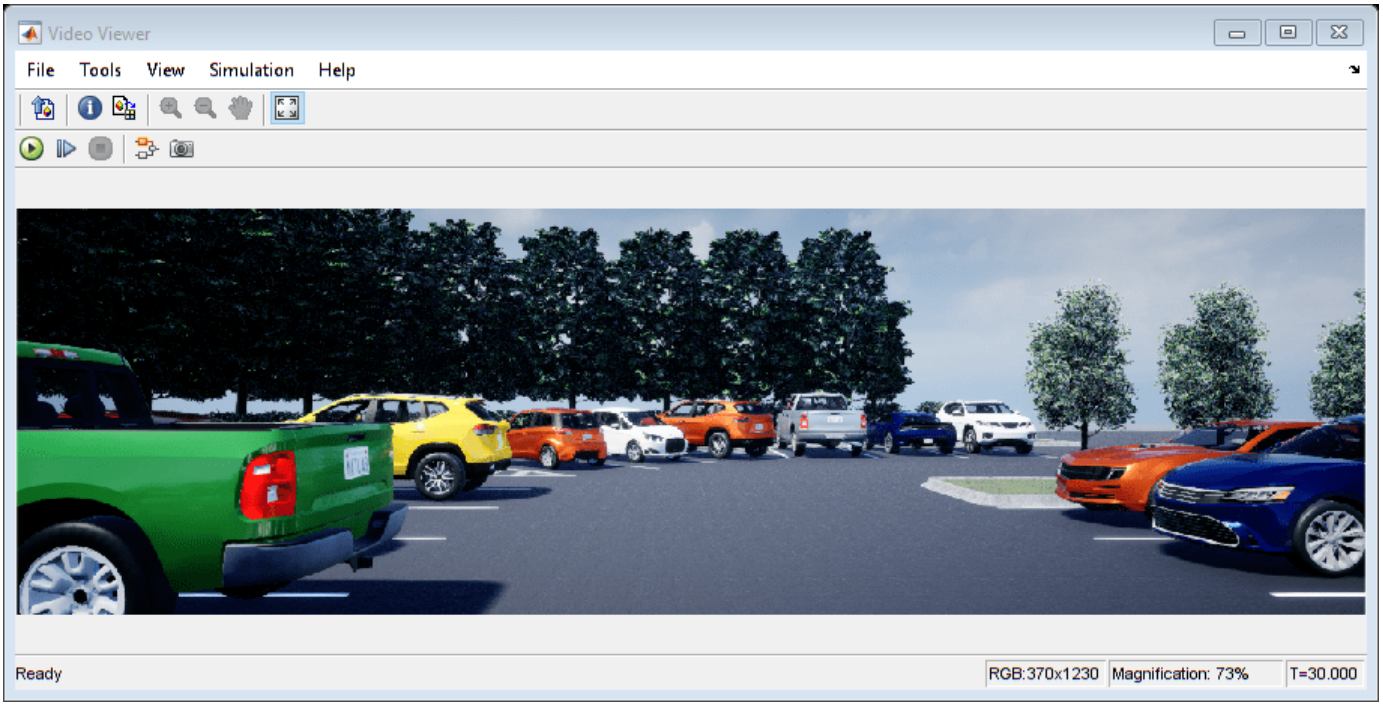
You can follow the “Select Waypoints for Unreal Engine Simulation” (Automated Driving Toolbox) example to select a sequence of waypoints and generate a reference trajectory for the ego vehicle. This example uses a recorded reference trajectory.

```
% Load reference path
refPosesData = load('parkingLotLocalizationData.mat');

% Set reference trajectory of the ego vehicle
refPosesX = refPosesData.refPosesX;
refPosesY = refPosesData.refPosesY;
refPosesT = refPosesData.refPosesT;

% Set camera intrinsics
focalLength    = [700, 700];  % specified in units of pixels
principalPoint = [600, 180];  % in pixels [x, y]
imageSize      = [370, 1230]; % in pixels [mrows, ncols]
```

Run Simulation

Run the simulation and visualize the estimated camera trajectory in the pre-built map. The white points represent the tracked 3-D map points in the current frame. You can compare the estimated trajectory with the ground truth provided by the Simulation 3D Camera block to evaluate the localization accuracy.

```
if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" + char(174) + " Windows" + cl
end

% Open video viewer to examine camera images
open_system([modelName, '/Video Viewer']);

% Run simulation
sim(modelName);
```


Video Viewer
File Tools View Simulation Help
Ready
RGB:370x1230
Magnification: 73%
T=30.000

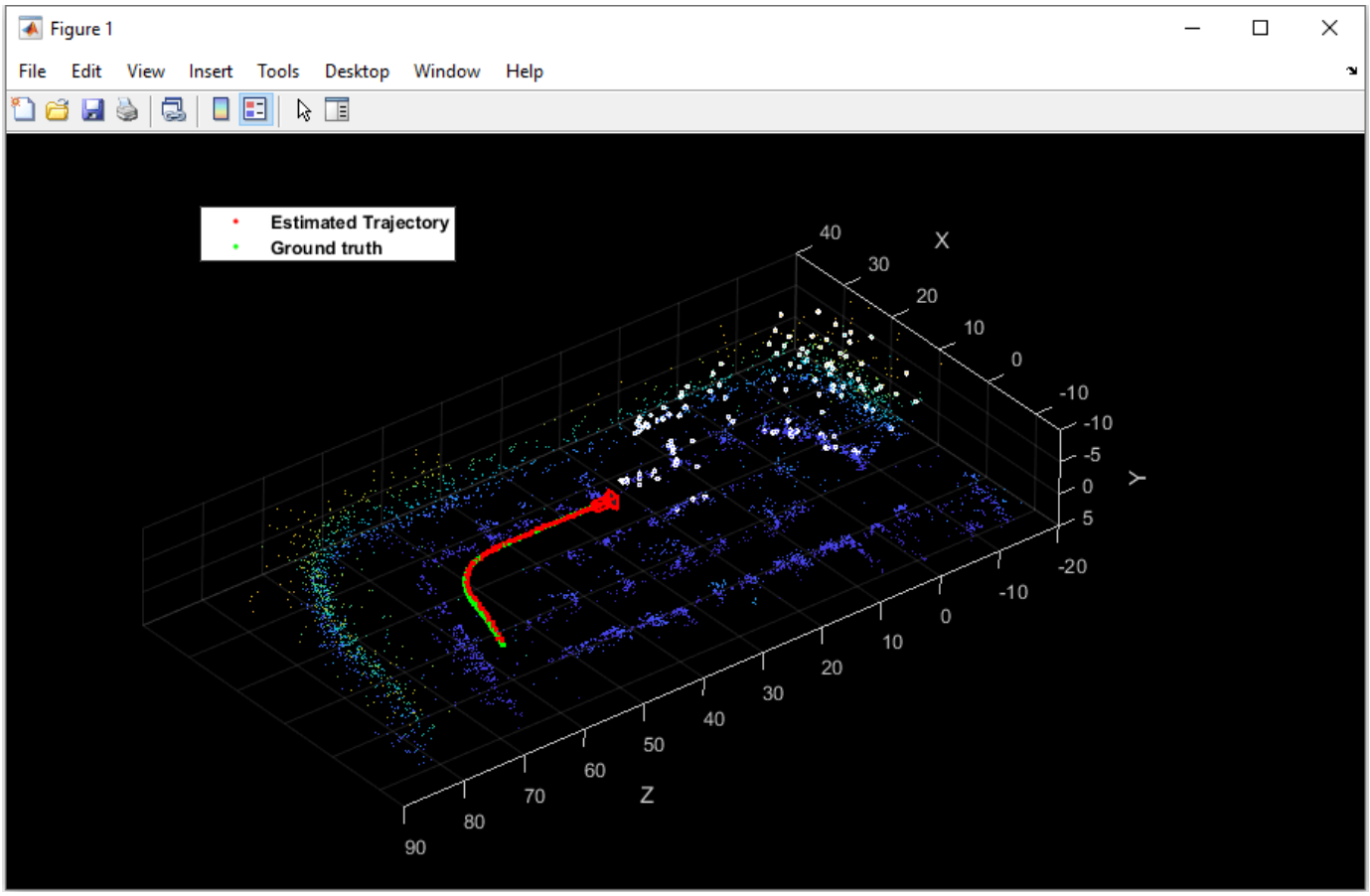


Figure 1
File Edit View Insert Tools Desktop Window Help
Estimated Trajectory
Ground truth
X
Y
Z

Close the model.

```
close_system([modelName, '/Video Viewer']);
close_system(modelName, 0);
```

Conclusion

With this setup, you can rapidly iterate over different scenarios, sensor configurations, or reference trajectories and refine the visual localization algorithm before moving to real-world testing.

- To select a different scenario, use the Simulation 3D Scene Configuration block. Choose from the existing prebuilt scenes or create a custom scene in the Unreal® Editor.
- To create a different reference trajectory, use the `helperSelectSceneWaypoints` tool, as shown in the Select Waypoints for Unreal Engine Simulation example.
- To alter the sensor configuration use the Simulation 3D Camera (Automated Driving Toolbox)block. The **Mounting** tab provides options for specifying different sensor mounting placements. The **Parameters** tab provides options for modifying sensor parameters such as detection range, field of view, and resolution. You can also use the Simulation 3D Fisheye Camera (Automated Driving Toolbox) block which provides a larger field of view.

Stereo Visual SLAM for UAV Navigation in 3D Simulation

Visual SLAM is the process of calculating the position and orientation of a camera with respect to its surroundings while simultaneously mapping the environment. Developing a visual SLAM algorithm and evaluating its performance in varying conditions is a challenging task. One of the biggest challenges is generating the ground truth of the camera sensor, especially in outdoor environments. The use of simulation enables testing under a variety of scenarios and camera configurations while providing precise ground truth.

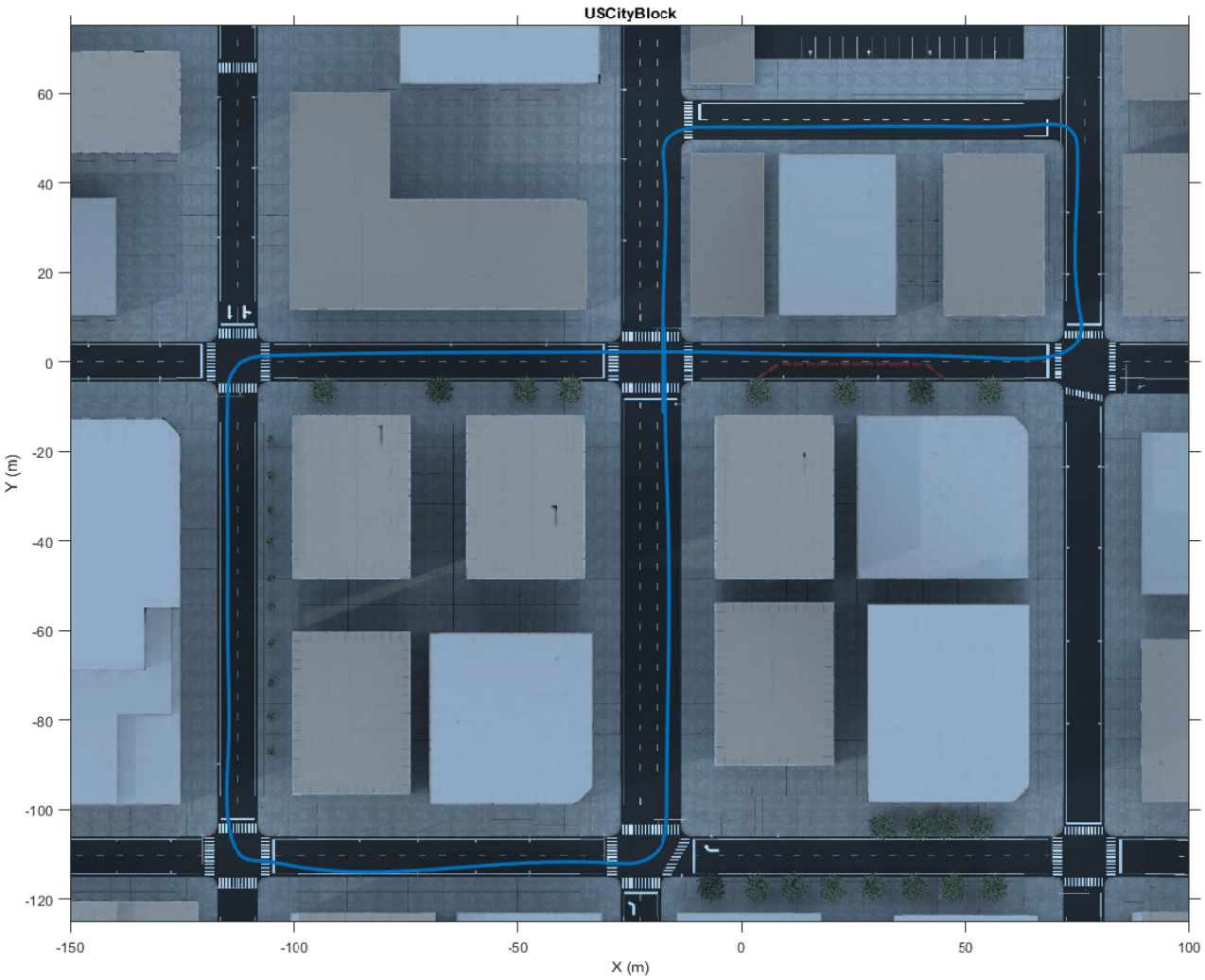
This example demonstrates the use of Unreal Engine® simulation to develop a visual SLAM algorithm for a UAV equipped with a stereo camera in a city block scenario. For more information about the implementation of the visual SLAM pipeline for a stereo camera [1] on page 1-0 , see the "Stereo Visual Simultaneous Localization and Mapping" on page 1-102 example.

Set Up Simulation Environment

First, set up a scenario in the simulation environment that can be used to test the visual SLAM algorithm. Use a scene depicting a typical city block with a UAV as the vehicle under test.

Next, select a trajectory for the UAV to follow in the scene. You can follow the "Select Waypoints for Unreal Engine Simulation" (Automated Driving Toolbox) example to interactively select a sequence of waypoints and then use the `helperSelectSceneWaypoints` function to generate a reference trajectory for the UAV. This example uses a recorded reference trajectory as shown below:

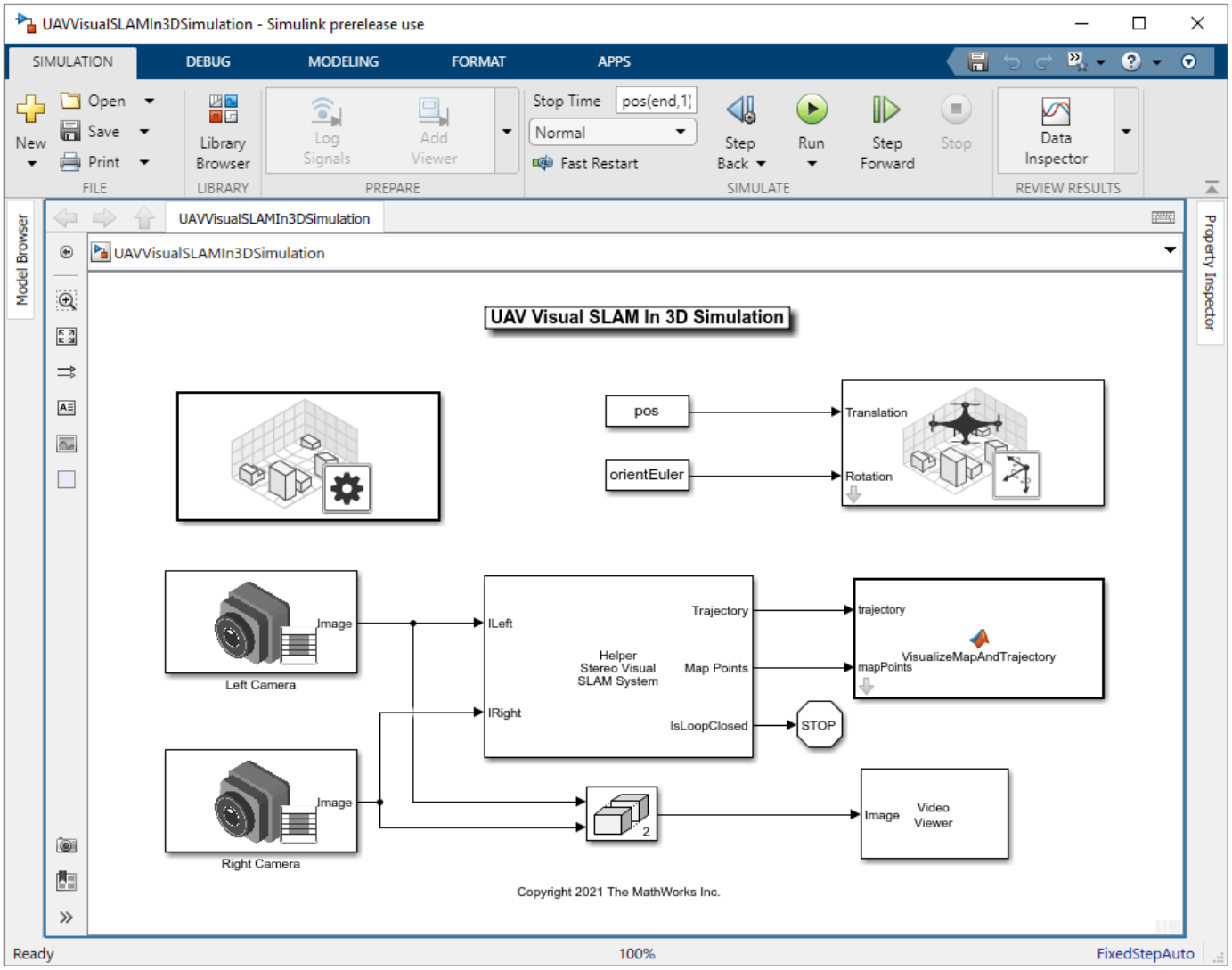
```
% Load reference path
data        = load('uavStereoSLAMData.mat');

pos         = data.pos;          % Position
orientEuler = data.orientEuler;  % Orientation
```



The UAVVisual`SLAMIn3DSimulation` Simulink® model is configured with the US City Block scene using the Simulation 3D Scene Configuration (UAV Toolbox) block. The model places a UAV on the scene using the Simulation 3D UAV Vehicle (UAV Toolbox) block. A stereo camera consisting of two Simulation 3D Camera (UAV Toolbox) blocks is attached to the UAV. In the dialog box of the Simulation 3D Camera (UAV Toolbox) block, use the **Mounting** tab to adjust the placement of the camera. Use the **Parameters** tab to configure properties of the camera to simulate different cameras. To estimate the intrinsics of the stereo camera that you want to simulate, use the “Using the Stereo Camera Calibrator App” on page 16-31 app.

```
% Stereo camera parameters
focalLength    = [1109, 1109]; % In pixels
```

```
principalPoint = [640, 360];   % In pixels [x, y]
imageSize      = [720, 1280];  % In pixels [mrows, ncols]
baseline       = 0.5;          % In meters

% Open the model
modelName = 'UAVVisualSLAMIn3DSimulation';
open_system(modelName);
```



Implement the Stereo Visual SLAM Algorithm

The Helper Stereo Visual SLAM System block implements the stereo visual SLAM pipeline, consisting of the following steps:

- **Map Initialization**: The pipeline starts by initializing the map of 3-D points from a pair of images generated from the stereo camera using the disparity map. The left image is stored as the first key frame.

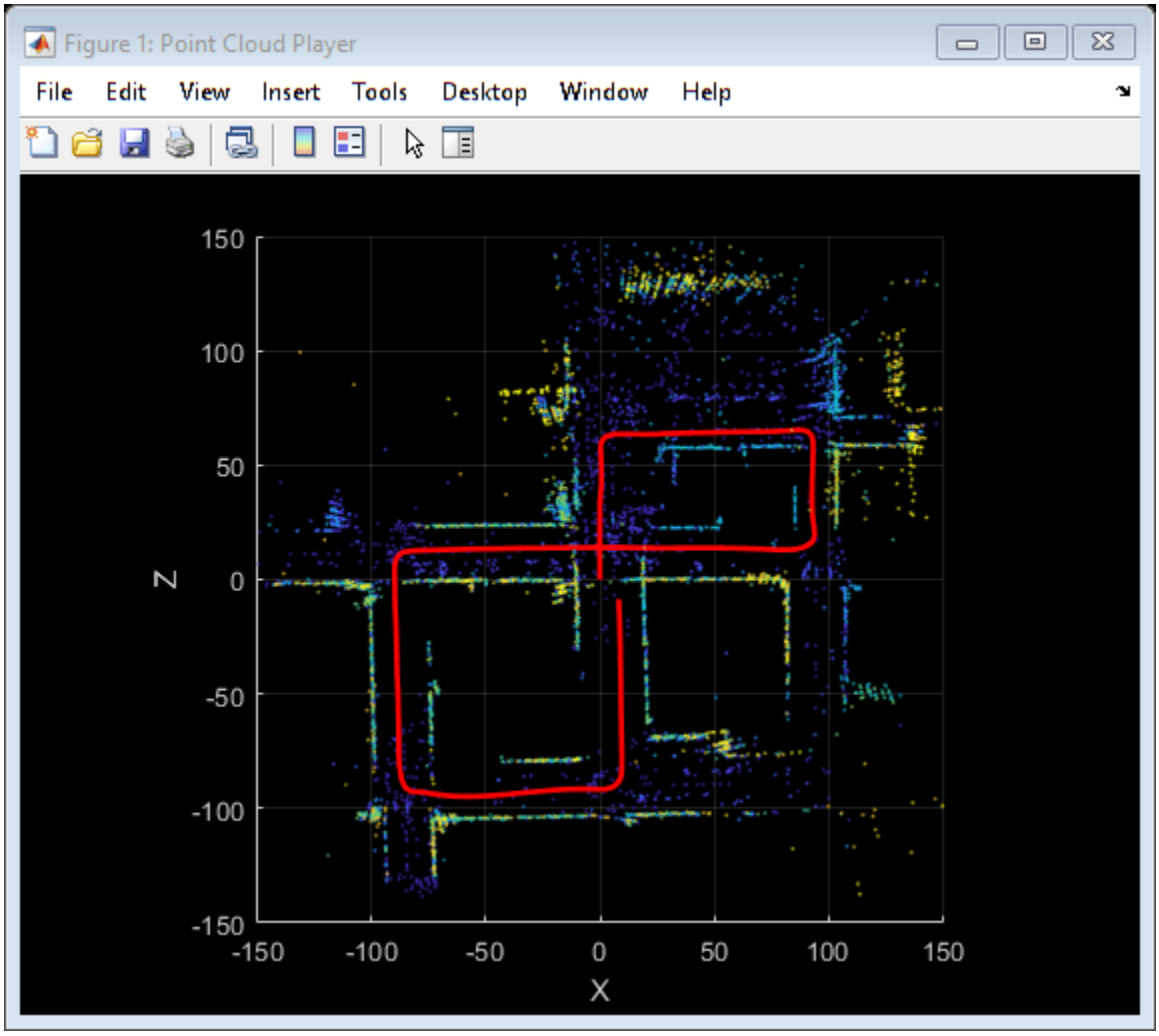
- **Tracking**: Once a map is initialized, for each new stereo pair, the pose of the camera is estimated by matching features in the left image to features in the last key frame. The estimated camera pose is refined by tracking the local map.
- **Local Mapping**: If the current left image is identified as a key frame, new 3-D map points are computed from the disparity of the stereo pair. 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.

For the implementation details of the algorithm, see the “Stereo Visual Simultaneous Localization and Mapping” on page 1-102 example.

Run Stereo Visual SLAM Simulation

Simulate the model and visualize the results. The **Video Viewer** block displays the stereo image output. The **Point Cloud Player** displays the reconstructed 3-D map with the estimated camera trajectory.

```
if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" + char(174) + " Windows" + cl
end

% Run simulation
sim(modelName);
```




```
Loop edge added between keyframe: 2 and 164
```

Close the model.

```
close_system(modelName);
```

References

[1] Mur-Artal, Raul, and Juan D. Tardós. "ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras." *IEEE Transactions on Robotics* 33, no. 5 (2017): 1255-1262.

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 like 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.

The advantages of using AprilTags as a calibration pattern include greater feature point detection, and consistent, repeatable detections. 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 “Using the Single Camera Calibrator App” on page 16-18.

This example shows how to calibrate a camera using AprilTags programmatically, and by using the Camera Calibrator app:

- Functional Interface on page 1-0
- Camera Calibration App on page 1-0

Camera Calibration Using Functional Interface

Step 1: Generate the calibration pattern

Download and prepare tag images

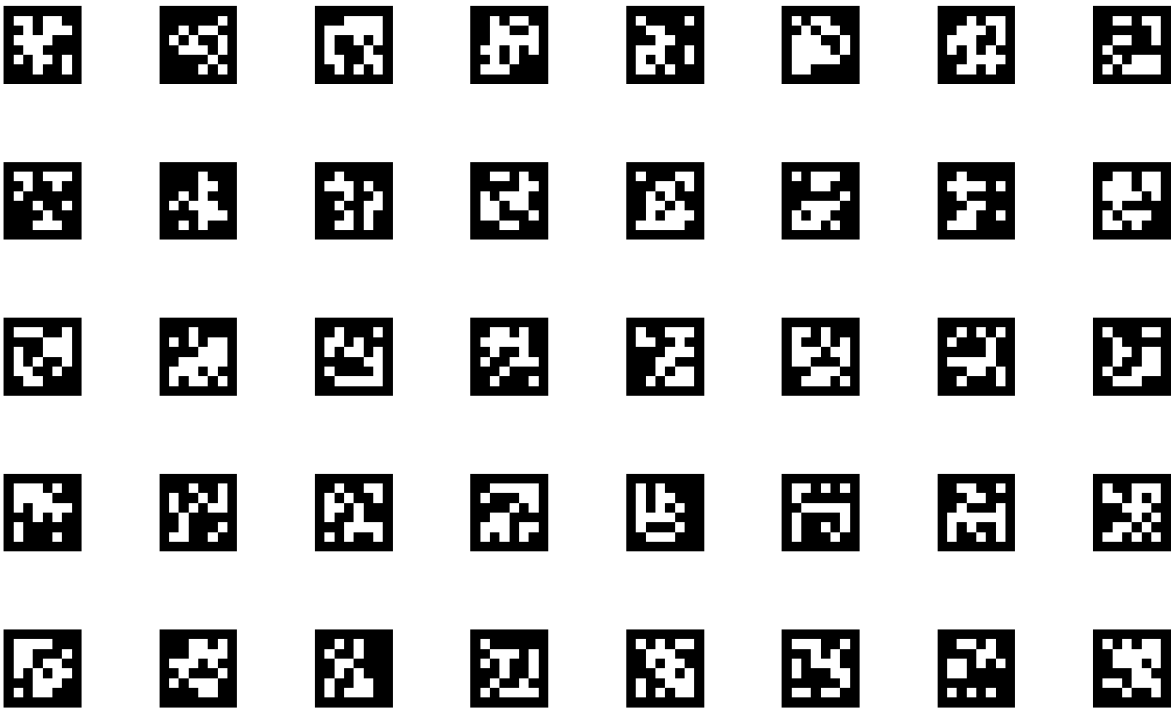
Pre-generated tags for all the supported families can be downloaded from here using a web browser or by running the following code:

```
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 on page 1-0 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.

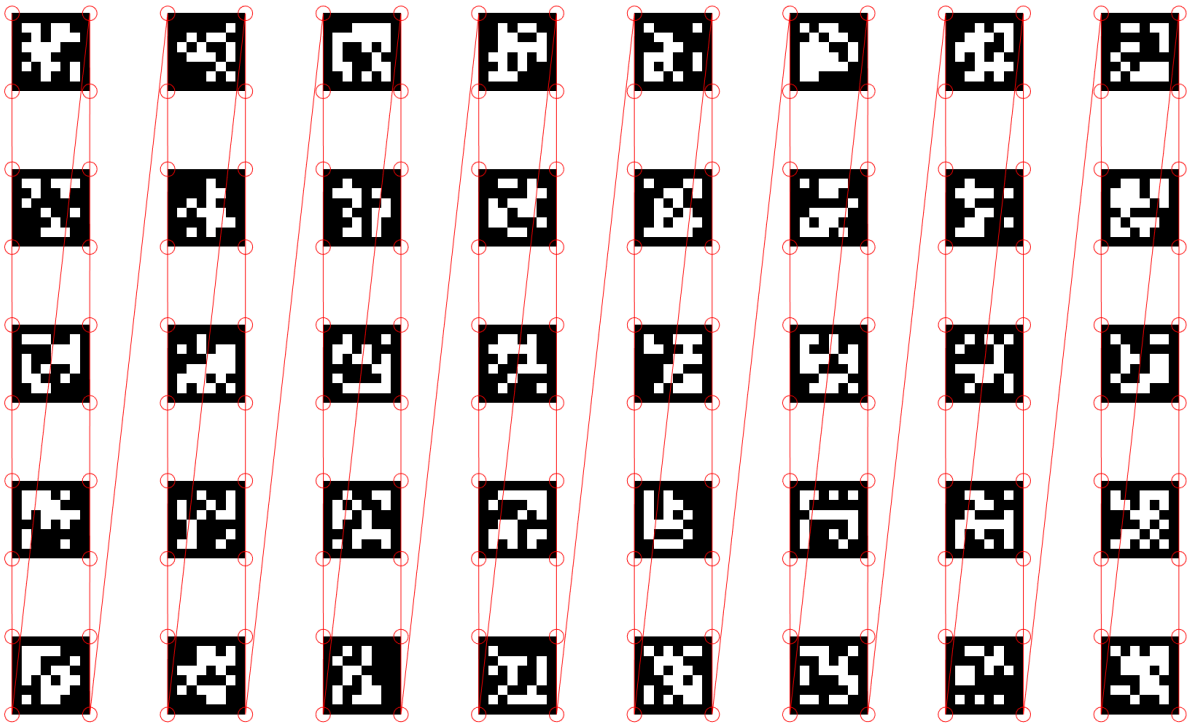
```
% Set the properties of the calibration pattern.
tagArrangement = [5,8];
tagFamily = 'tag36h11';
```

```
% 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, such as a checkerboard.

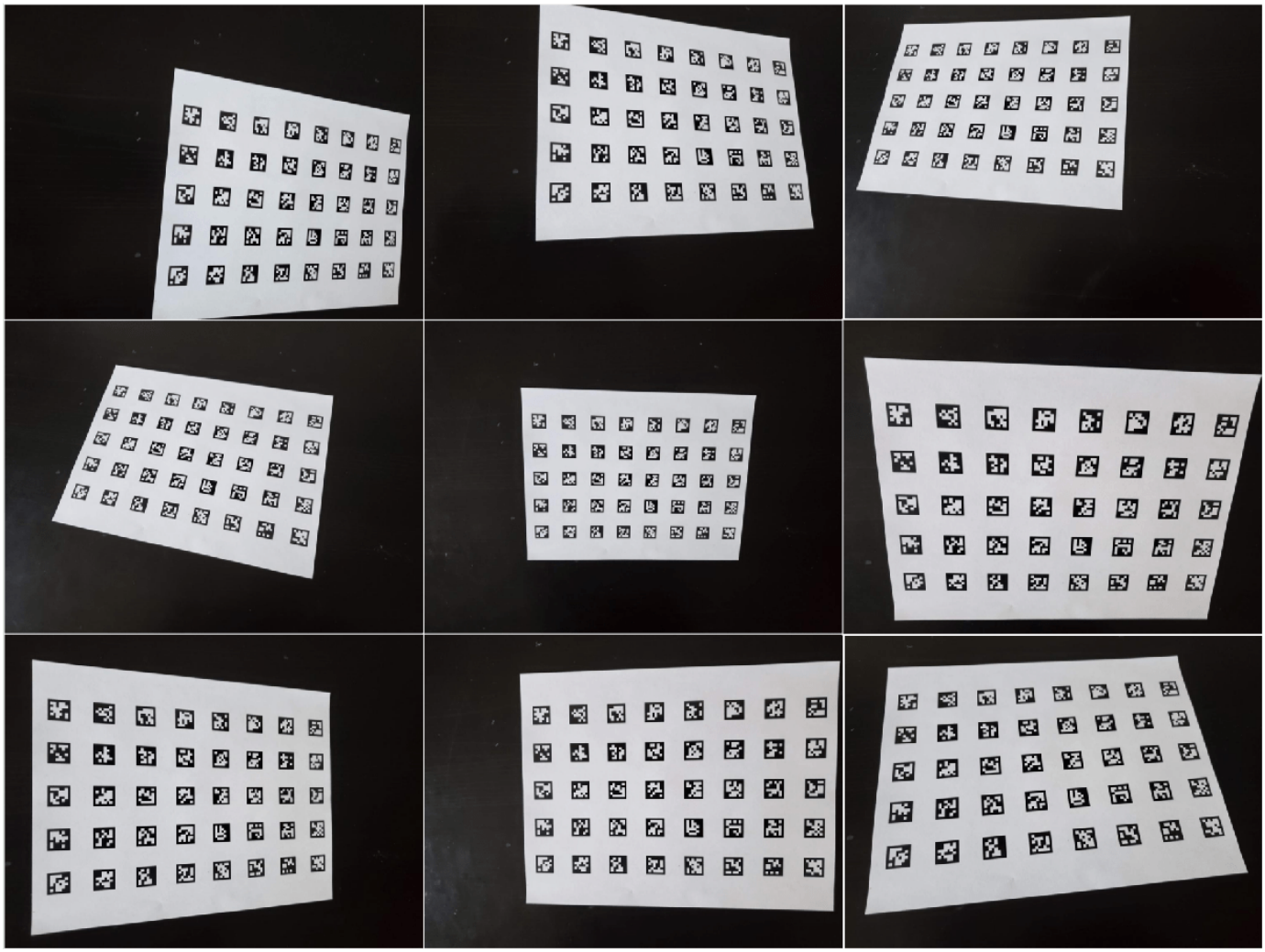
```
% Read and localize the tags in the calibration pattern.
[tagIds, tagLocs] = readAprilTag(calibPattern,tagFamily);

% 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

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 (e.g. 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, see “Prepare Camera and Capture Images” on page 16-4.



Step 2: Detect and localize the AprilTags

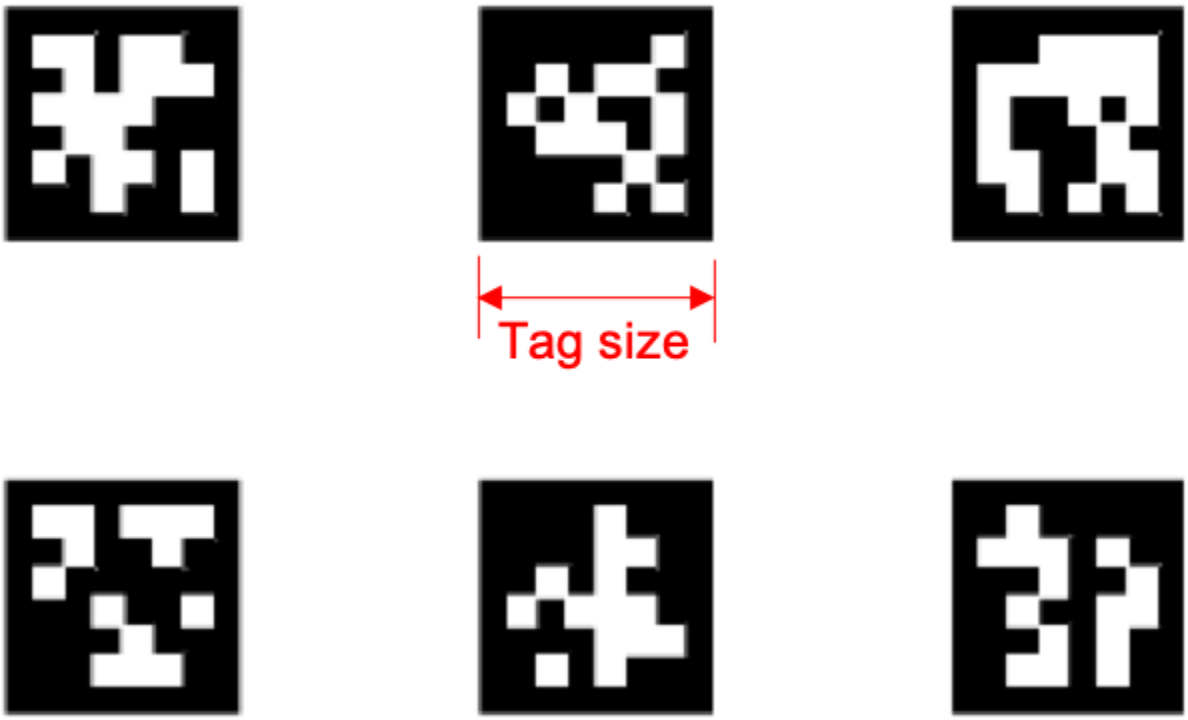
The helperDetectAprilTagCorners on page 1-0 function, included at the end of the example, is used to detect, and localize the tags from the captured images and arrange them in a checkerboard fashion to be used as key points in the calibration procedure.

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

% 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 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);
```

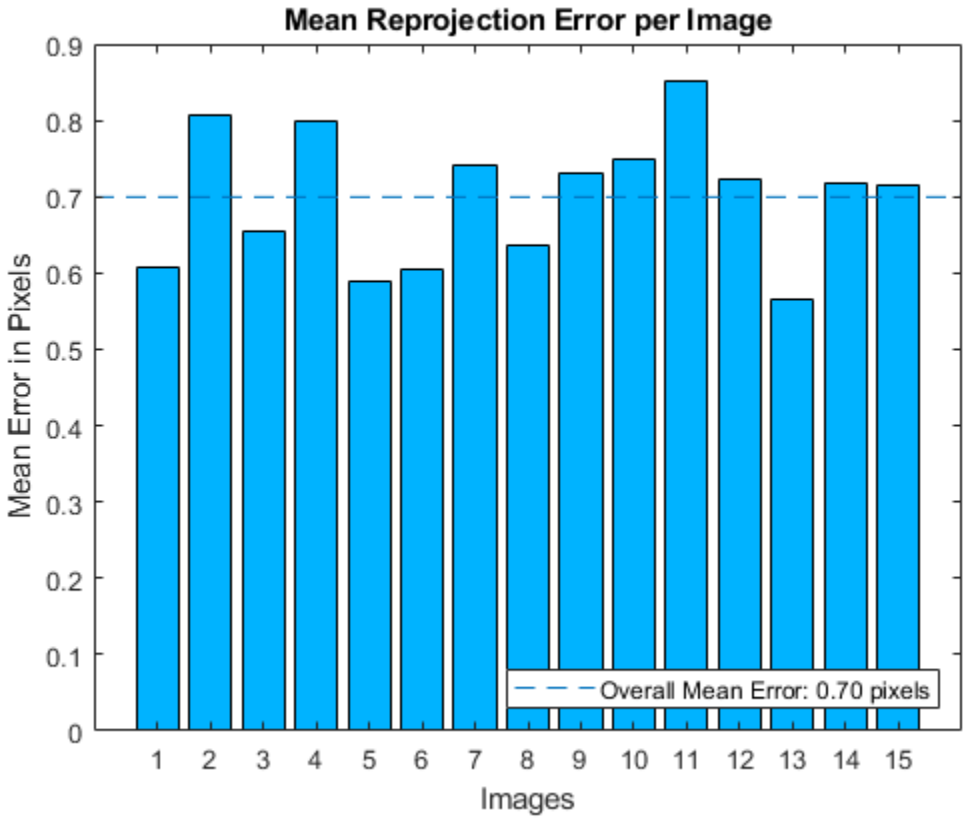
Step 4: Estimate camera parameters

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

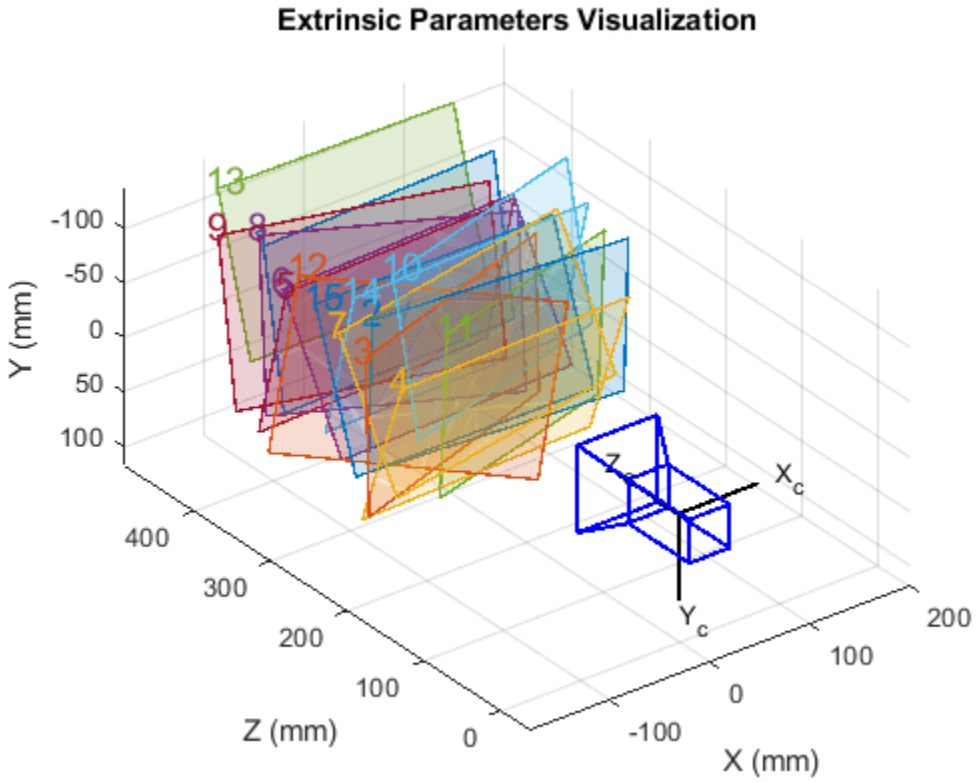
```
% Determine the size of the images.
I = readimage(imdsCalib, 1);
imageSize = [size(I,1), size(I,2)];

% Estimate the camera parameters.
params = estimateCameraParameters(imagePoints,worldPoints,ImageSize=imageSize);
```

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

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

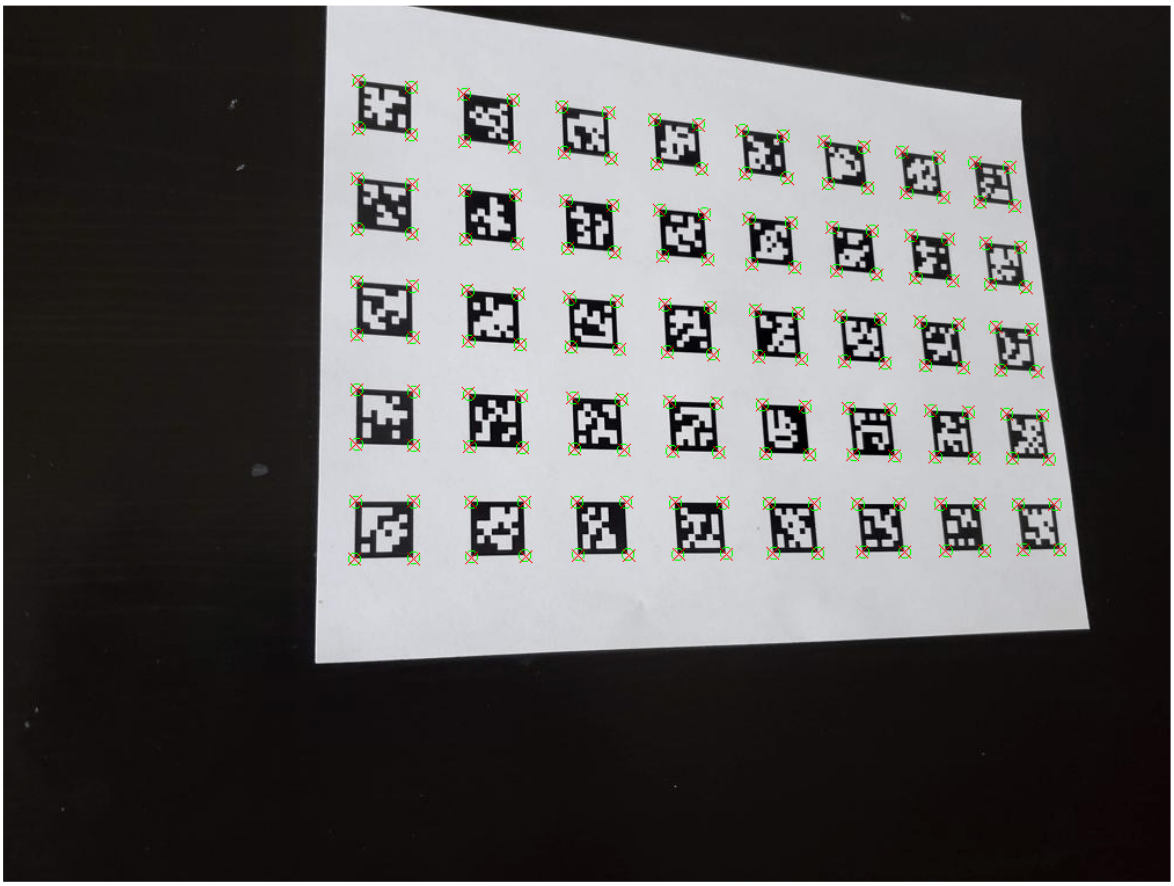



```
% Display the extrinsics.
figure
showExtrinsics(params)
```



Inspect the locations of the detected image points and the reprojected points, which were 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.

Integrating AprilTag Calibration Pattern Support into Camera Calibrator App

For convenience of use, the above workflow can also be integrated into the Camera Calibrator app. The overall workflow remains the same and the steps are:

1. Add images with AprilTags.

2. Import a custom pattern detector class for AprilTags. The detector must do the following:

- Detect and localize the AprilTags
- Generate world points for the calibration pattern

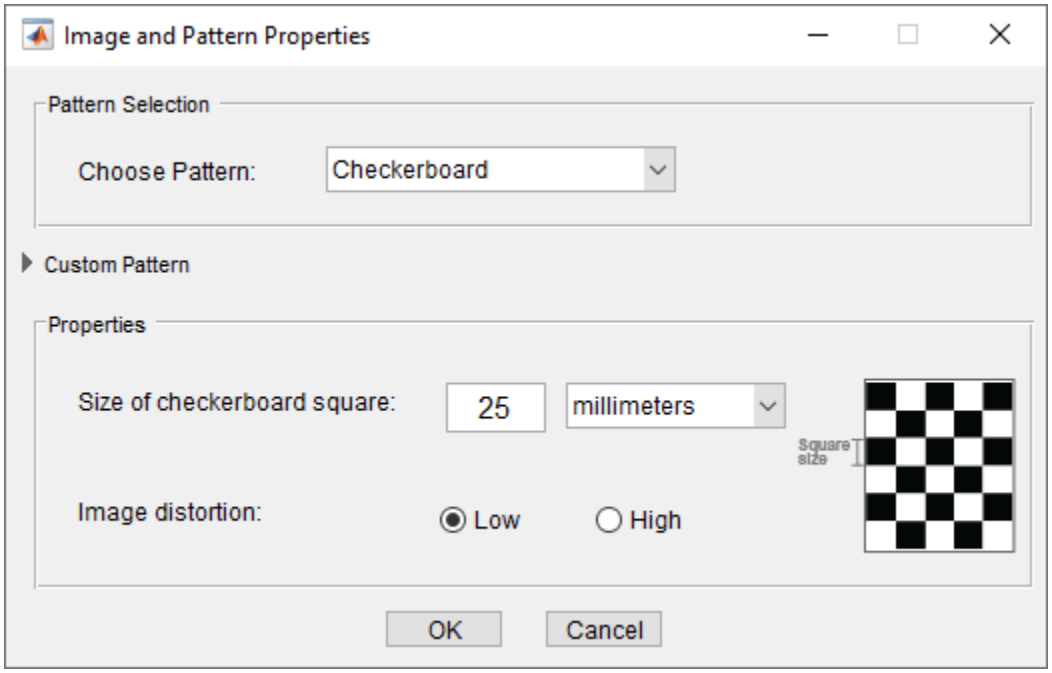
3. Estimate camera parameters.

Add Images with AprilTags

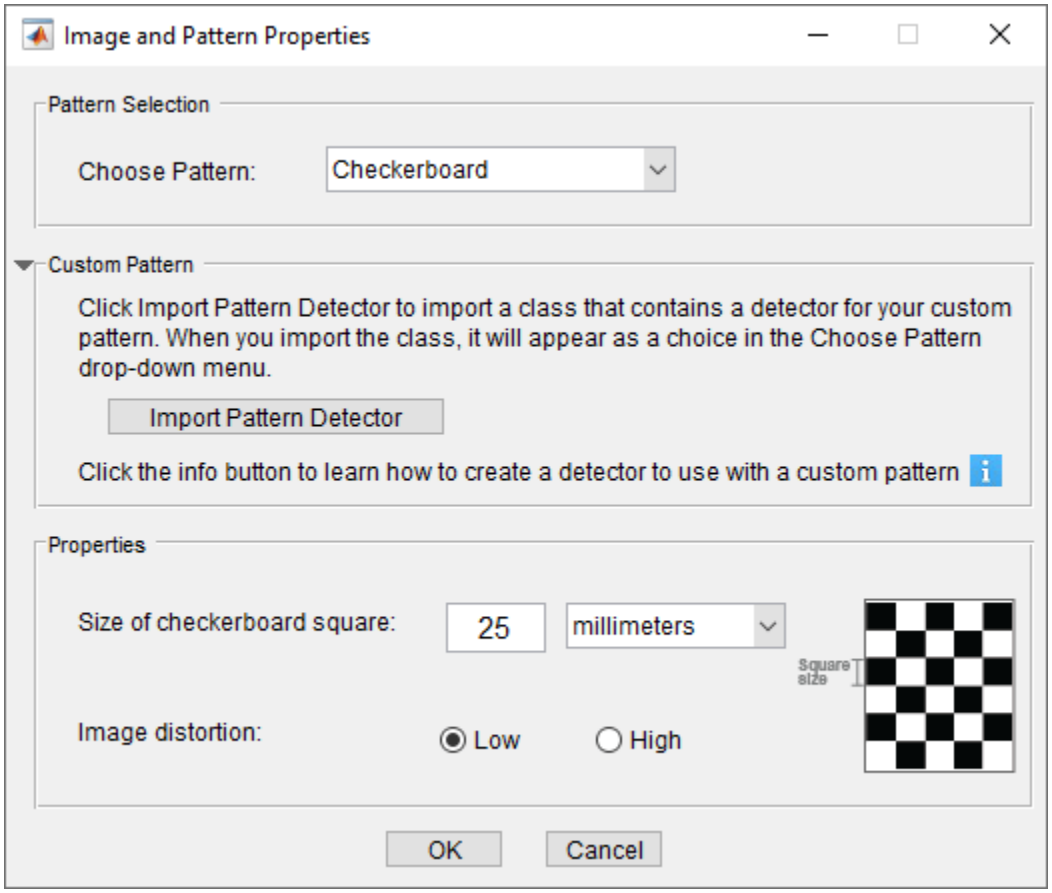
Open Camera Calibrator App:

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

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. We will reuse the same images as above on page 1-0 . You will need at least 2 images for camera calibration. Once you add images, the following UI will appear:



Expand the **Custom Pattern** panel, to see more options.



Import Custom Pattern Detector Class

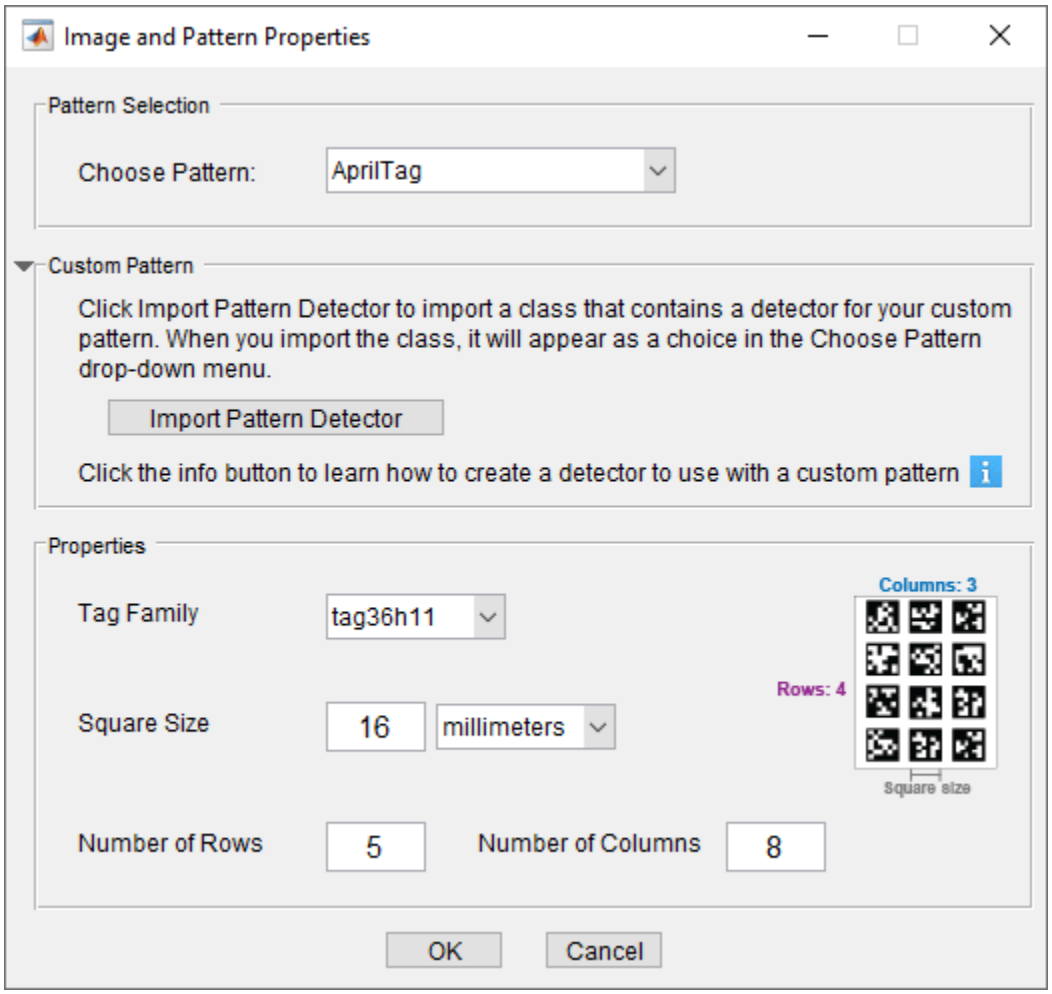
The above UI shows a drop-down list for pattern selection. By default, the app does not include a pattern detector for AprilTags. You can create a custom pattern detector class and then add it to the list to use in the app. For more information on how to create a custom pattern, click on the information icon ( ). A custom pattern detector class for AprilTags has been provided in MyCustomAprilTagPatternDetector.m file. This class contains UI code for parameters needed by the detector and functions for detecting and processing the custom AprilTags calibration pattern.

The example uses the **configureUIComponents()** function to configure the UI component and the **initializePropertyValues()** to initialize it. The **helperDrawImageAxesLabels** on page 1-0 function, included at the end of the example, is used to render the origin, X-axis and Y-axis labels in the calibration images displayed in the Camera Calibrator app dialog.

The main calibrations functions are:

- **detectPatternPoints()** - Detects and localizes the AprilTags from the captured images and sorts them for use as key points in the calibration procedure. This function is implemented using **helperDetectAprilTagCorners** on page 1-0 function, given at the end of the example.
- **generateWorldPoints()** - Computes world coordinates for the corresponding image coordinates in the AprilTag pattern. This function is implemented using **helperGenerateAprilTagPattern** on page 1-0 function, given at the end of the example.

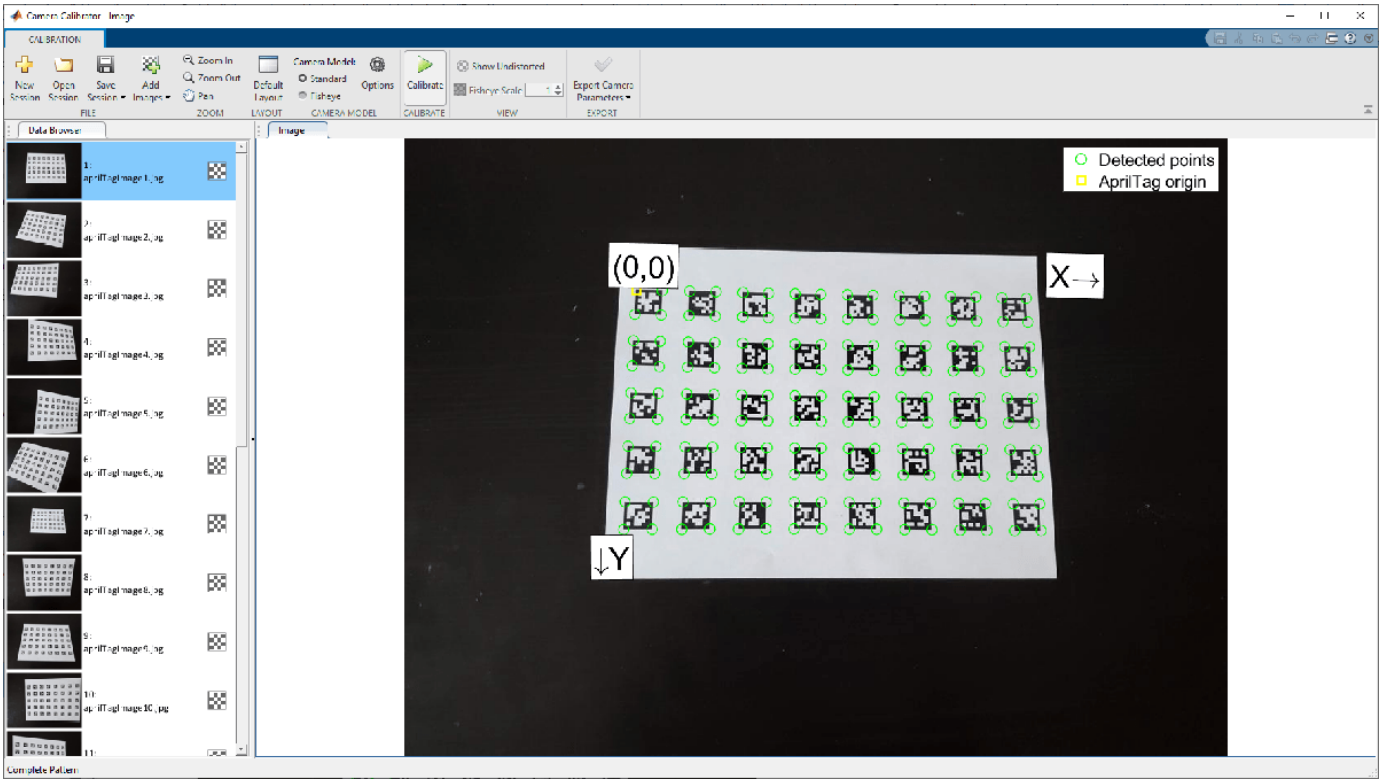
Import the custom pattern detector class by clicking on the **Import Pattern Detector** button under **Custom Pattern** panel. Choose the class file **MyCustomAprilTagPatternDetector.m**. If there are no errors in the class, then you will see the following view:



For this example, all the fields in the **Properties** panel have correct values. But you can customize these values to fit your needs. Note that **Square Size** represents the width of the tag in world units and is also assumed to be equal to the spacing between each tag in the image.

Click **OK** and the **Data Browser** pane displays a list of images with IDs, as shown below:

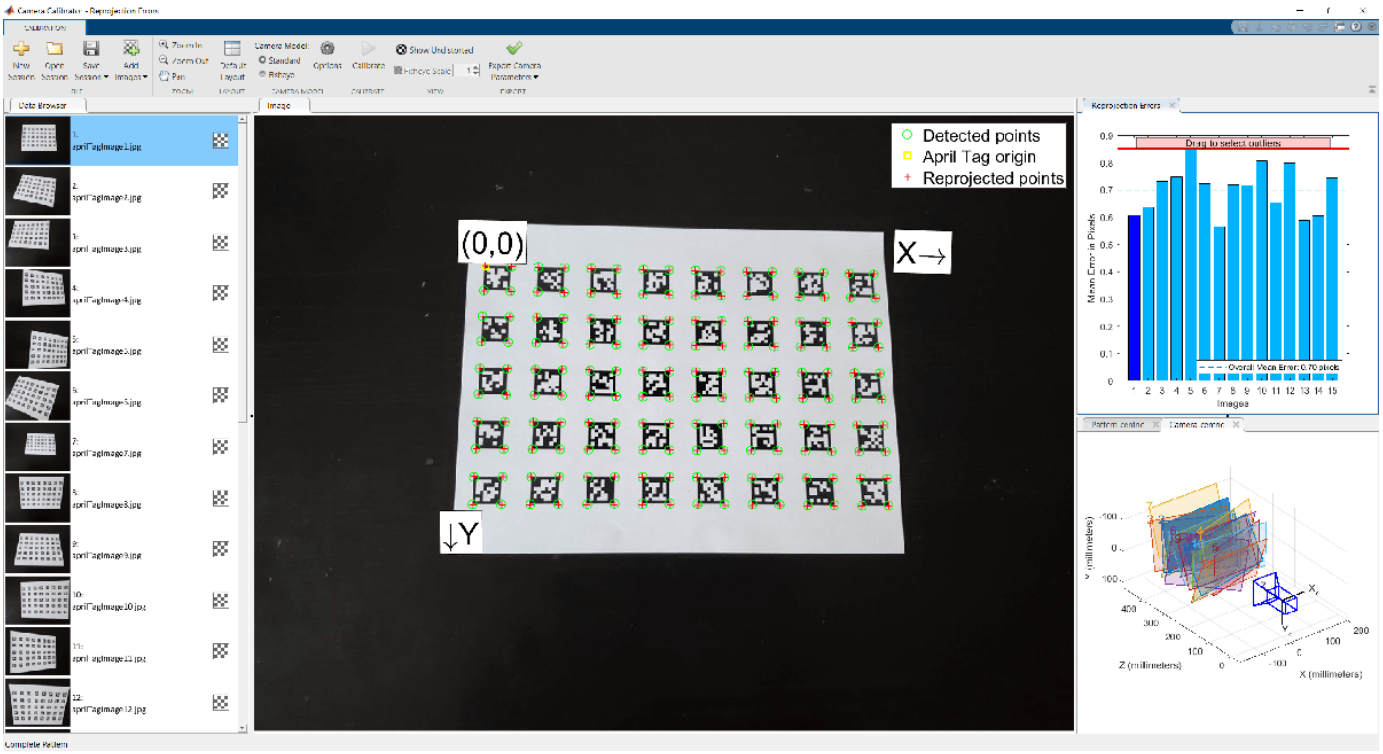
These images will contain the detected pattern. To view an image, select it from the **Data Browser** pane.



Estimate Camera Parameters

At this point, camera calibration process is the same as given in “Using the Single Camera Calibrator App” on page 16-18.

With the default calibration settings, click the **Calibrate** button on the **Calibration** tab. Visualize the accuracy of the calibration by inspecting the **Reprojection Errors** pane and then visualize estimates of the extrinsic camera parameters in the **Camera-centric** pane which shows the patterns positioned with respect to the camera.



Supporting Functions and Classes

helperGenerateAprilTagPattern generates an AprilTag based calibration pattern.

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

numTags = tagArragement(1)*tagArragement(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, tagArragement)';
tagImages = tagImages(:,:,columnMajIdx(:));

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

end
```

helperDetectAprilTagCorners detects AprilTag calibration pattern in images.

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

% 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
```

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
```

helperDrawImageAxesLabels renders the origin, X-axis and Y-axis labels in the calibration images displayed in the calibrator app.

```
function [originLabel,xLabel,yLabel] = helperDrawImageAxesLabels(boardSize,imagePoints)

    numBoardRows = boardSize(1)-1;
    numBoardCols = boardSize(2)-1;

    % Reshape checkerboard corners to boardSize shaped array
    boardCoordsX = reshape(imagePoints(:,1), [numBoardRows,numBoardCols]);
    boardCoordsY = reshape(imagePoints(:,2), [numBoardRows,numBoardCols]);
    boardCoords = cat(3, boardCoordsX,boardCoordsY);

    % Origin label (check if origin location is inside the image)
    if ~isnan(boardCoordsX(1,1))
        p1 = boardCoords(1,1,:);

        refPointIdx = find(~isnan(boardCoordsX(:,1)),2);
        p2 = boardCoords(refPointIdx(2),1,:);
```

```
    refPointIdx = find(~isnan(boardCoordsX(1,:)),2);
    p3 = boardCoords(1,refPointIdx(2),:);

    [loc, theta] = getAxesLabelPosition(p1,p2,p3);

    originLabel.Location = loc;
    originLabel.Orientation = theta;
else
    originLabel = struct;
end

% X-axis label
firstRowIdx = numBoardCols:-1:1;
refPointIdx13 = find(~isnan(boardCoordsX(1,firstRowIdx)),2);
refPointIdx13 = firstRowIdx(refPointIdx13);

p1 = boardCoords(1,refPointIdx13(1),:);
p3 = boardCoords(1,refPointIdx13(2),:);

refPointIdx2 = find(~isnan(boardCoordsX(:,refPointIdx13(1))),2);
p2 = boardCoords(refPointIdx2(2),refPointIdx13(1),:);

[loc, theta] = getAxesLabelPosition(p1,p2,p3);
theta = 180 + theta;

xLabel.Location    = loc;
xLabel.Orientation = theta;

% Y-axis label
firstColIdx = numBoardRows:-1:1;
refPointIdx12 = find(~isnan(boardCoordsX(firstColIdx,1)),2);
refPointIdx12 = firstColIdx(refPointIdx12);

p1 = boardCoords(refPointIdx12(1),1,:);
p2 = boardCoords(refPointIdx12(2),1,:);

refPointIdx3 = find(~isnan(boardCoordsX(refPointIdx12(1),:)), 2);
p3 = boardCoords(refPointIdx12(1),refPointIdx3(2),:);

[loc,theta] = getAxesLabelPosition(p1,p2,p3);

yLabel.Location = loc;
yLabel.Orientation = theta;

%-------------------------------------------------------------
% p1+v
%  \
%   \     v1
%    p1 ------ p2
%    |
% v2 |
%    |
%    p3
function [loc,theta] = getAxesLabelPosition(p1,p2,p3)
    v1 = p3 - p1;
    theta = -atan2d(v1(2),v1(1));

    v2 = p2 - p1;
```

```
        v = -v1 - v2;
        d = hypot(v(1),v(2));
        minDist = 40;
        if d < minDist
            v = (v / d) * minDist;
        end
        loc = p1 + v;
    end
    %--------------------------------------------------------------

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.

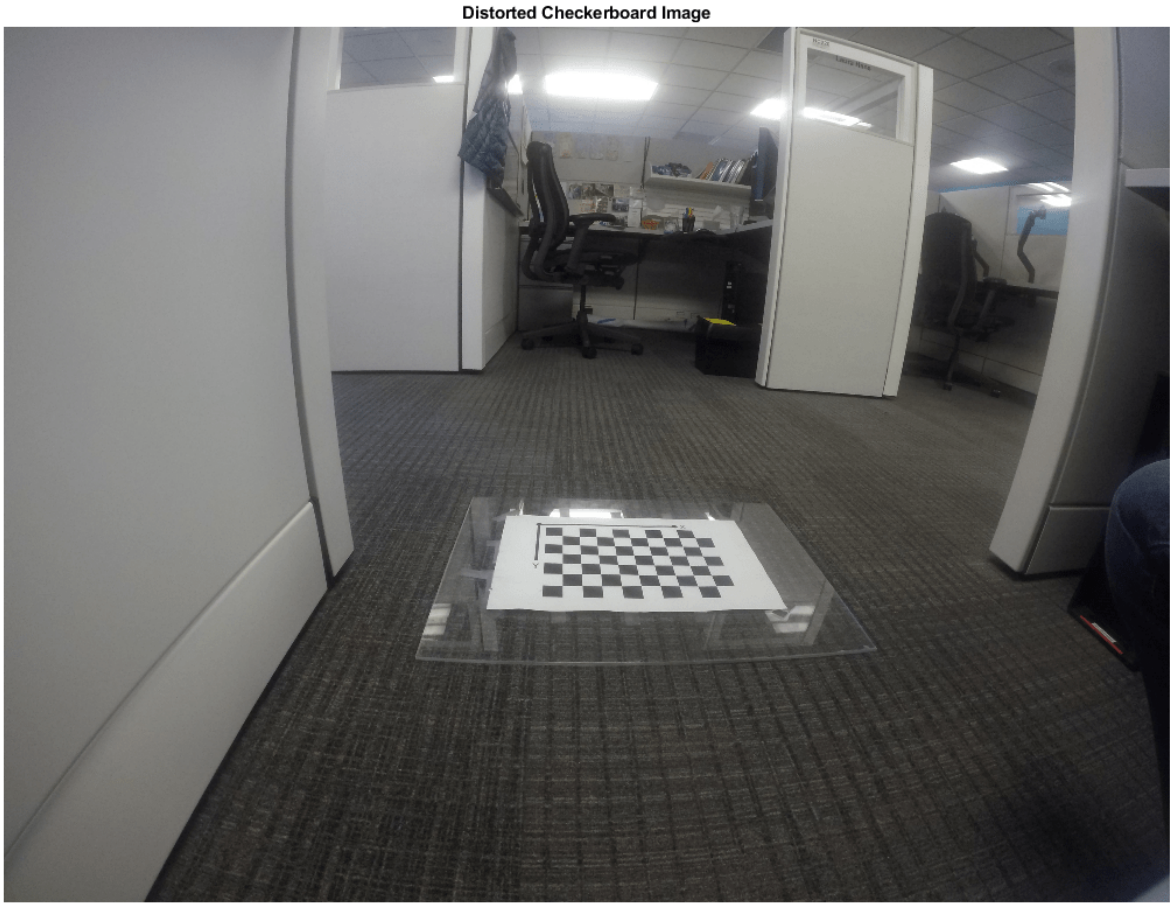
```
 % Gather a set of calibration images.
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', ...
      'calibration', 'gopro'));
imageFileNames = images.Files;

% 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
```

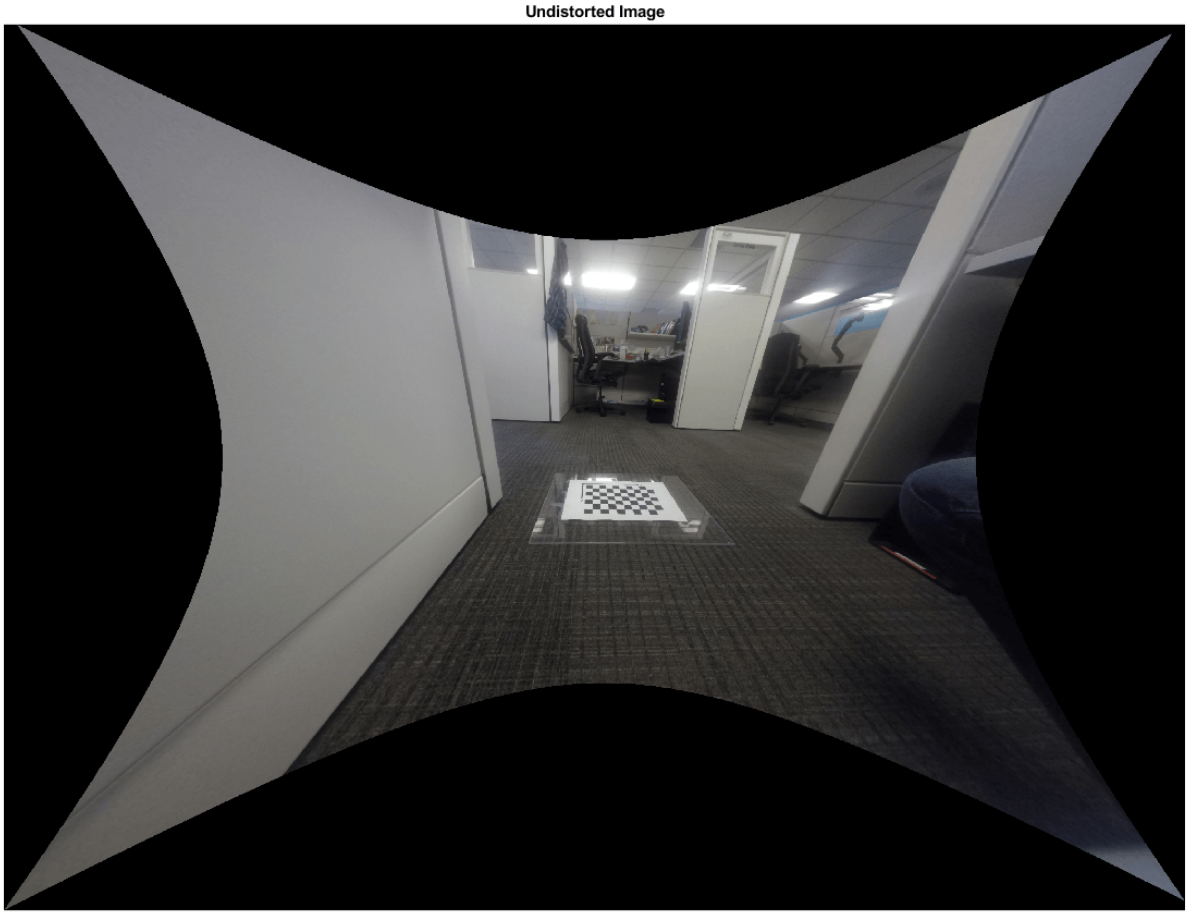
```
% noticeable distortion, such as along the wall next to the checkerboard.

imageFileName = fullfile(toolboxdir('driving'), 'drivingdata', 'checkerboard.png');
I = imread(imageFileName);
imshow(I)
title('Distorted Checkerboard Image');
```




```
[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')
```

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.

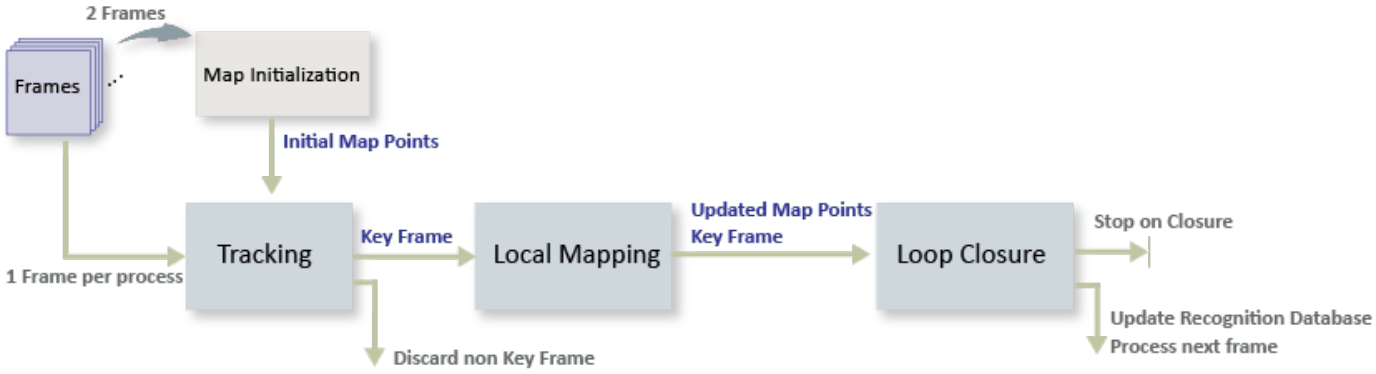
To speed up computations, you can enable parallel computing from the “Computer Vision Toolbox Preferences” dialog box. To open Computer Vision Toolbox™ preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Computer Vision Toolbox.**

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.
- **Place 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_o
dataFolder      = fullfile(tempdir, 'tum_rgbd_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

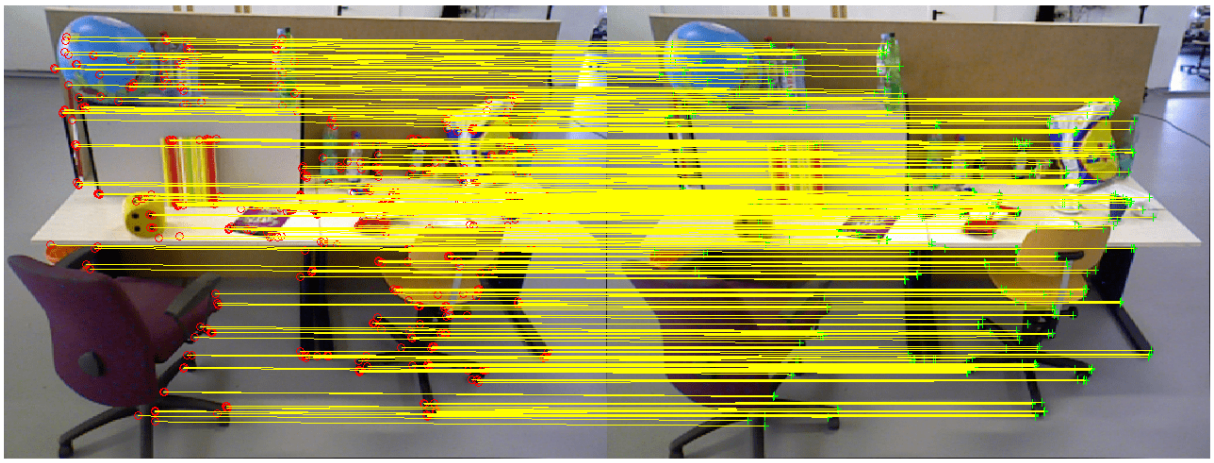
when it is located in the front of both cameras, when its reprojection error is low, and when the parallax of the two views of the point is sufficiently large.

```
% 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;
numPoints   = 1000;
[preFeatures, prePoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels, numPoint

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, nu

    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),:);

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

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

    % 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 = 1; % 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 26
```

```
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. It also builds and updates a pose graph. The absolute camera poses and relative camera poses of odometry edges are stored as `rigid3d` objects. The relative camera poses of loop-closure edges are stored as `affine3d` objects.
- `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', indexPai

% 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));
```

Initialize Place Recognition Database

Loop detection is performed using the bags-of-words approach. A visual vocabulary represented as a `bagOfFeatures` object is created offline with the ORB descriptors extracted from a large set of images in the dataset by calling:

```
bag = bagOfFeatures(imds,'CustomExtractor',
@helperORBFeatureExtractorFunction, 'TreeProperties', [5, 10],
'StrongestFeatures', 1);
```

where `imds` is an `imageDatastore` object storing the training images and `helperORBFeatureExtractorFunction` is the ORB feature extractor function. See “Image Retrieval with Bag of Visual Words” on page 17-110 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 ORB features.

```
% Load the bag of features data created offline
bofData         = load('bagOfFeaturesDataSLAM.mat');

% Initialize the place recognition database
loopDatabase    = invertedImageIndex(bofData.bof,"SaveFeatureLocations", false);

% Add features of the first two key frames to the database
addImageFeatures(loopDatabase, preFeatures, preViewId);
addImageFeatures(loopDatabase, currFeatures, currViewId);
```

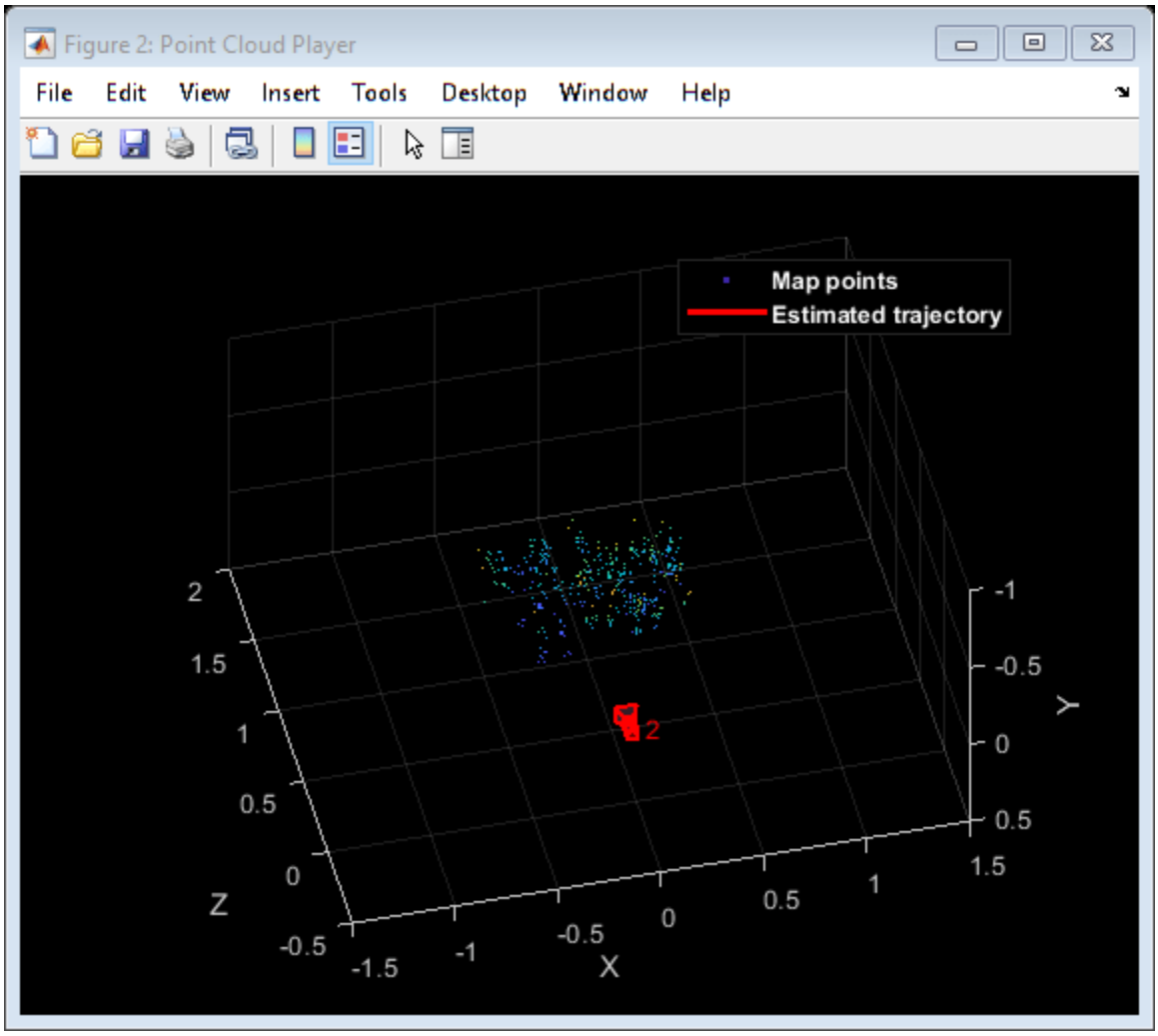
Refine and Visualize the Initial Reconstruction

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', 20, ...
    'Solver', 'preconditioned-conjugate-gradient');

% 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, preViewId, 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;

% 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 `matchFeaturesInRadius`.
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 `matchFeaturesInRadius` 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.

If tracking is lost because not enough number of feature points could be matched, try inserting new key frames more frequently.

```
% Main loop
isLastFrameKeyFrame = true;
while ~isLoopClosed && currFrameIdx < numel(imds.Files)
    currI = readimage(imds, currFrameIdx);

    [currFeatures, currPoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels, nu

    % 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 and 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 100 map points.
    % 2. The map points tracked by the current frame are fewer than 90% of
    %    points tracked by the reference key frame.
    %
    % Tracking performance is sensitive to the value of numPointsKeyFrame.
    % If tracking is lost, try a larger value.
    %
    % localKeyFrameIds:   ViewId of the connected key frames of the current frame
    numSkipFrames     = 20;
    numPointsKeyFrame = 100;
    [localKeyFrameIds, currPose, mapPointsIdx, featureIdx, isKeyFrame] = ...
        helperTrackLocalMap(mapPointSet, directionAndDepth, vSetKeyFrames, mapPointsIdx, ...
        featureIdx, currPose, currFeatures, currPoints, intrinsics, scaleFactor, numLevels, ...
        isLastFrameKeyFrame, lastKeyFrameIdx, currFrameIdx, numSkipFrames, numPointsKeyFrame);

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

    if ~isKeyFrame
```

```
    currFrameIdx        = currFrameIdx + 1;
    isLastFrameKeyFrame = false;
    continue
else
    isLastFrameKeyFrame = true;
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.

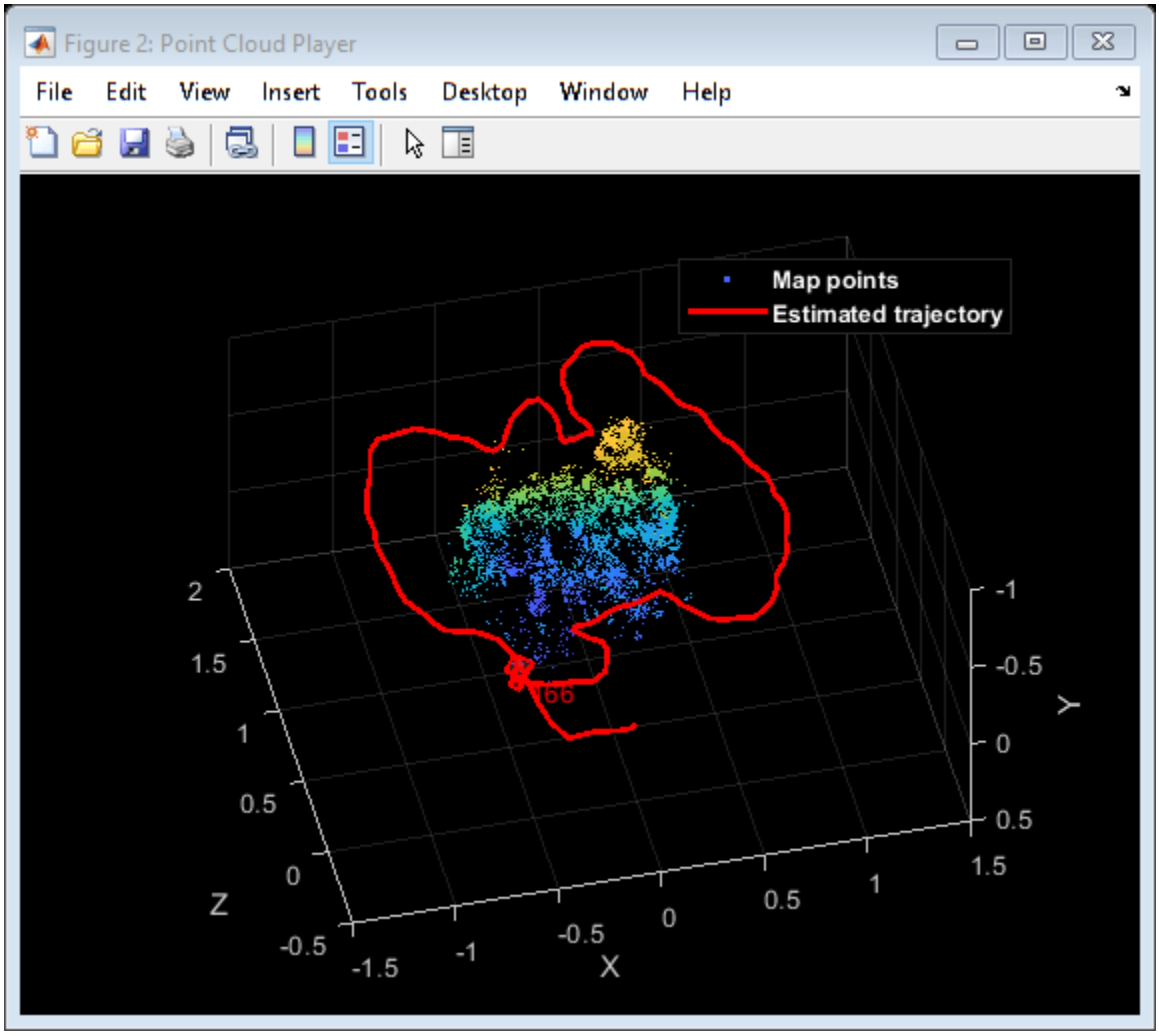
```
% Add the new key frame
[mapPointSet, vSetKeyFrames] = helperAddNewKeyFrame(mapPointSet, vSetKeyFrames, ...
    currPose, currFeatures, currPoints, mapPointsIdx, featureIdx, localKeyFrameIds);

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

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

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

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

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

Loop Closure

The loop closure detection step takes the current key frame processed by the local mapping process and tries to detect and close the loop. 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, use `estimateGeometricTransform3D` to compute the relative pose between the loop candidate frame and the current key frame. The relative pose represents a 3-D similarity transformation stored in an `affine3d` object. Then add the loop connection with the relative pose and update `mapPointSet` and `vSetKeyFrames`.

```
    % Check loop closure after some key frames have been created
    if 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, loopEdgeNumMatches);

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

    % If no loop closure is detected, add current features into the database
    if ~isLoopClosed
        addImageFeatures(loopDatabase,  currFeatures, currKeyFrameId);
    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: 9 and 166
```

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

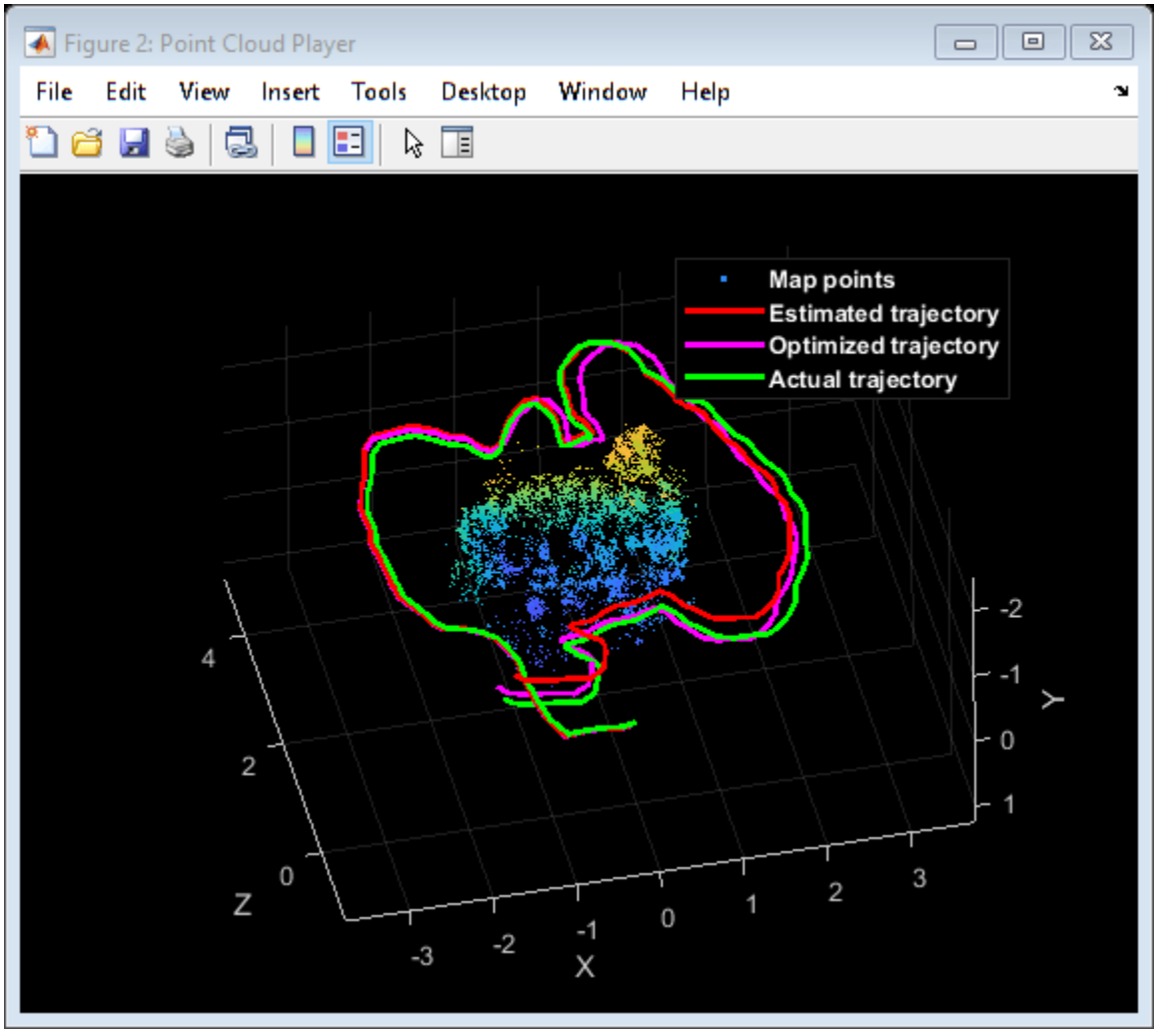
```
% Optimize the poses
minNumMatches      = 30;
[vSetKeyFramesOptim, poseScales] = optimizePoses(vSetKeyFrames, minNumMatches, 'Tolerance', 1e-16

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

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('orbslamGroundTruth.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): 0.20758
```

This concludes an overview of how to build a map of an indoor environment and estimate the trajectory of the camera using ORB-SLAM. You can test the visual SLAM pipeline with a different dataset by tuning the following parameters:

- `numPoints`: For image resolution of 480x640 pixels set `numPoints` to be 1000. For higher resolutions, such as 720 × 1280, set it to 2000. Larger values require more time in feature extraction.
- `numSkipFrames`: For frame rate of 30fps, set `numSkipFrames` to be 20. For a slower frame rate, set it to be a smaller value. Increasing `numSkipFrames` improves the tracking speed, but may result in tracking lost when the camera motion is fast.

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.

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

`helperORBFeatureExtractorFunction` implements the ORB 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, numPoints, varargin)

% In this example, the images are already undistorted. In a general
% workflow, uncomment the following code to undistort the images.
%
```

```
% if nargin > 4
%     intrinsics = varargin{1};
% end
% Irgb  = undistortImage(Irgb, intrinsics);

% Detect ORB features
Igray  = im2gray(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);
error1in2  = sum((locations2 - xy1In2).^2, 2);
error2in1  = sum((locations1 - xy2In1).^2, 2);

outlierThreshold = 6;

score = sum(max(outlierThreshold-error1in2, 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);
error1in2 = (sum([locations2, ones(size(locations2, 1),1)].* lineIn2, 2)).^2 ...
    ./ sum(lineIn2(:,1:2).^2, 2);

outlierThreshold = 4;

score = sum(max(outlierThreshold-error1in2, 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
```

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, poseScales)
%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 = indices
    majorViewIds = directionAndDepth.MajorViewId(i);
    poseNew = posesNew(majorViewIds).T;
    poseNew(1:3, 1:3) = poseNew(1:3, 1:3) * poseScales(majorViewIds);
    tform = posesOld(majorViewIds).T \ poseNew;
    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.

See Also

Related Examples

- “Stereo Visual Simultaneous Localization and Mapping” on page 1-102

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

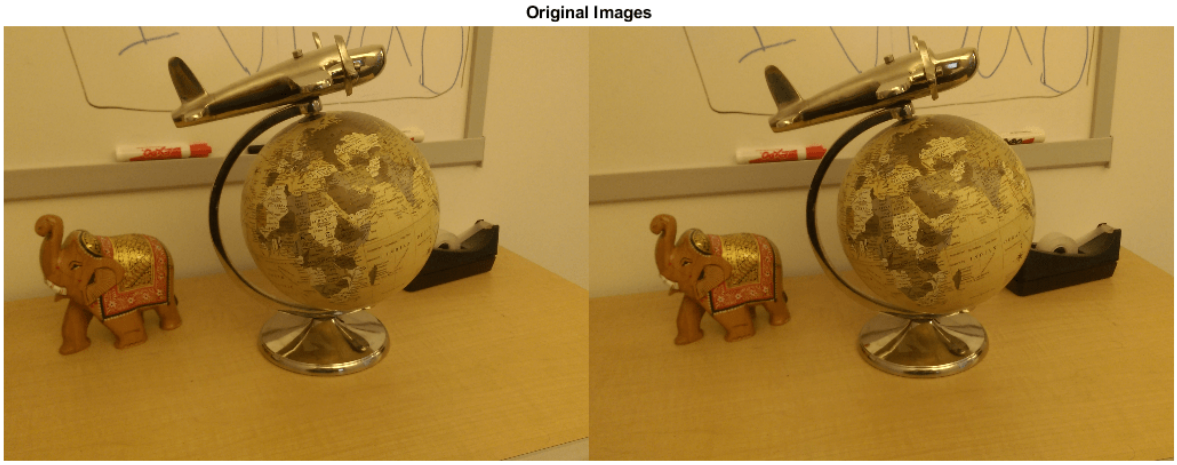
This example shows how to reconstruct a 3-D scene from a pair of 2-D images taken with a camera calibrated using the Camera Calibrator app. The algorithm consists of the following steps:

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 `estimateEssentialMatrix`.
3. Compute the motion of the camera using the `relativeCameraPose` 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');
```



Load Camera Parameters

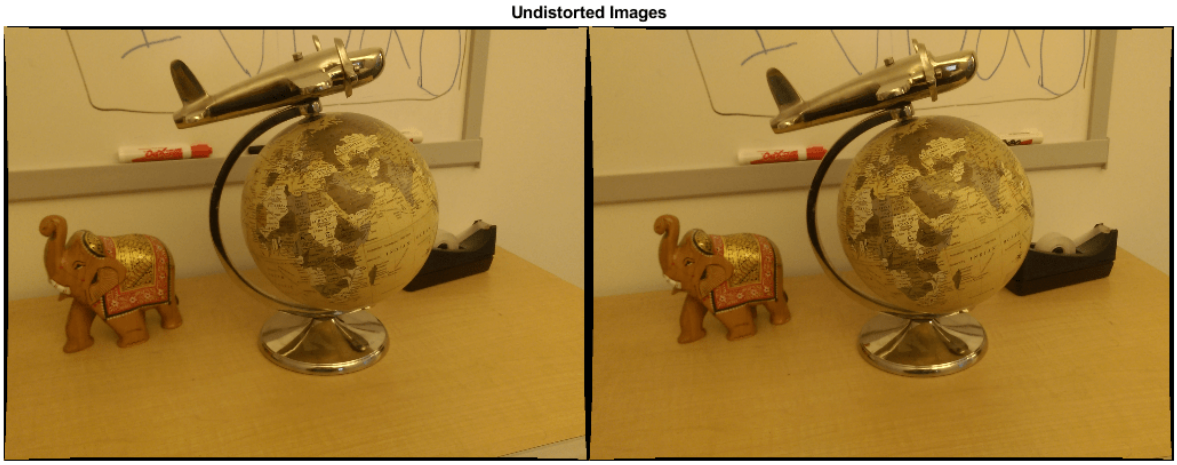
This example uses the camera parameters calculated by the Camera Calibrator app. The parameters are stored in the `cameraParams` object, and include the camera intrinsics and lens distortion coefficients.

```
% 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');
```

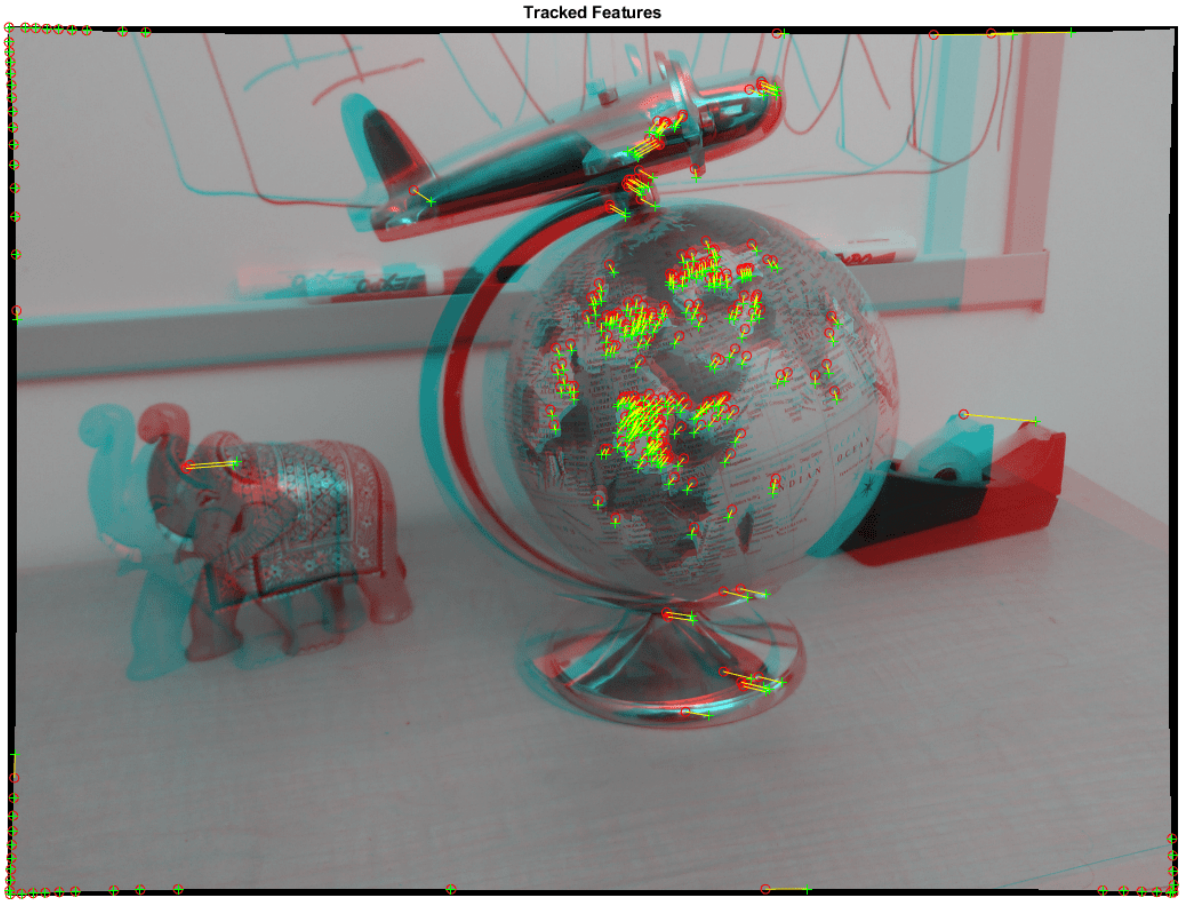


Find Point Correspondences Between The Images

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(im2gray(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));
```

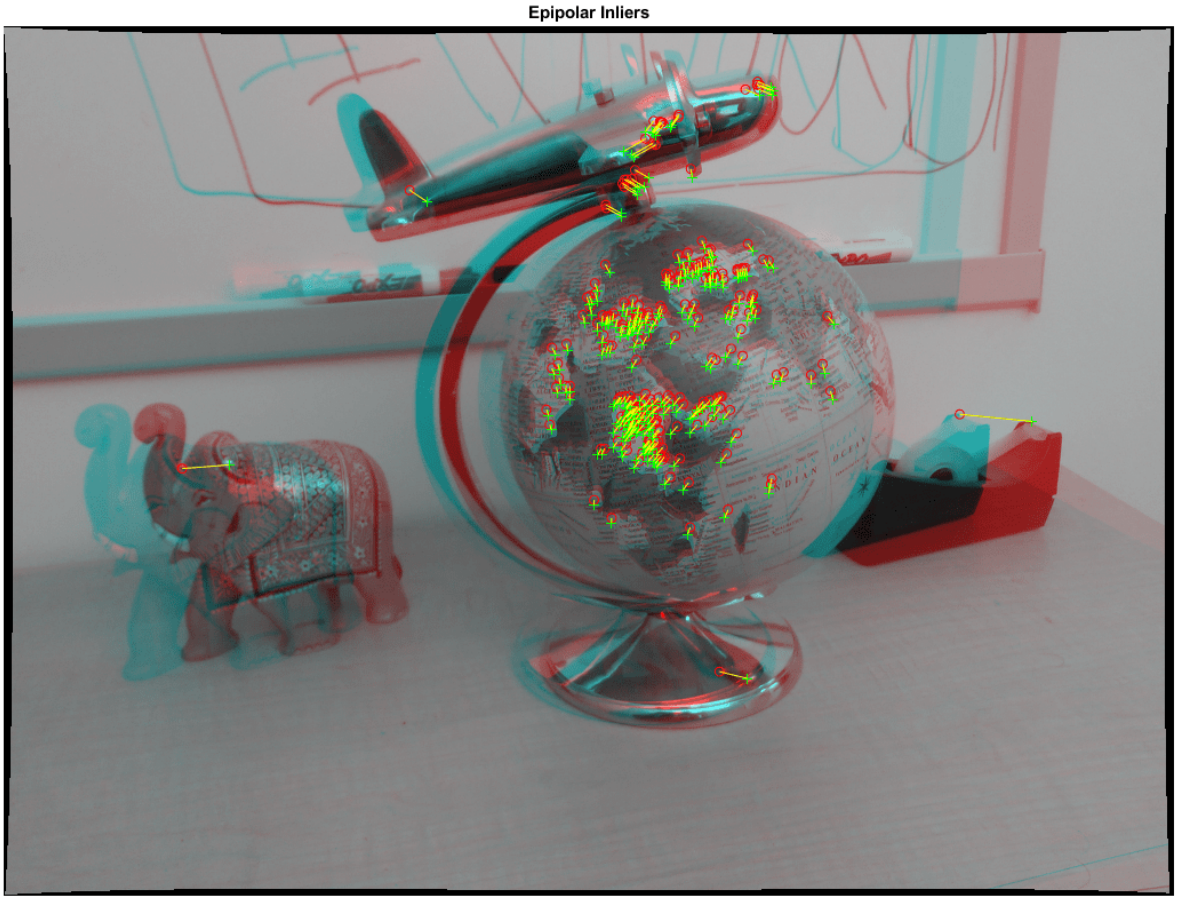

```
% Create the point tracker
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels', 5);

% 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 Essential Matrix

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

```
% Estimate the fundamental matrix
[E, epipolarInliers] = estimateEssentialMatrix(...
    matchedPoints1, matchedPoints2, cameraParams, 'Confidence', 99.99);

% 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 location and orientation of the second camera relative to the first one. Note that `t` is a translation unit vector, because translation can only be computed up to scale.

```
[orient, loc] = relativeCameraPose(E, 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. Use an ROI to exclude points close to the
% image edges.
roi = [30, 30, size(I1, 2) - 30, size(I1, 1) - 30];
imagePoints1 = detectMinEigenFeatures(im2gray(I1), 'ROI', roi, ...
    'MinQuality', 0.001);

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

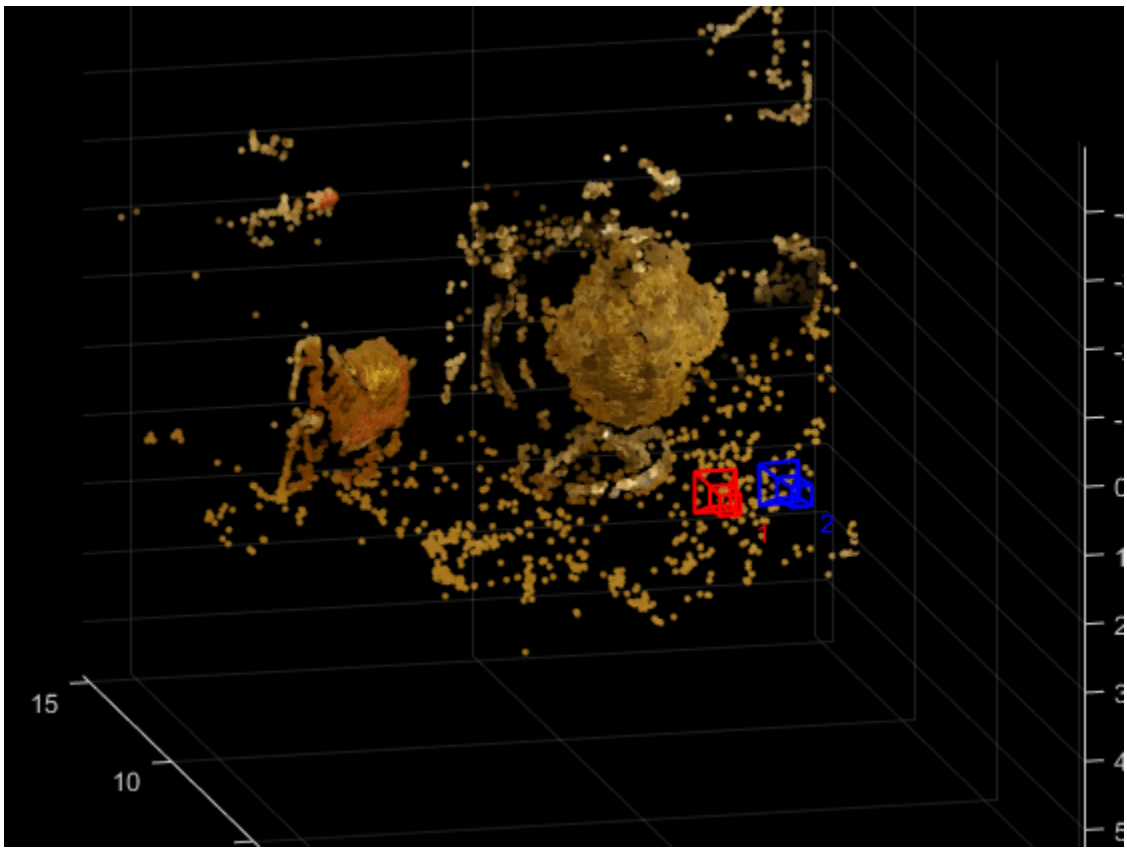
```
% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);

% 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 Z-axis. Thus, its
% transformation is identity.
tform1 = rigid3d;
camMatrix1 = cameraMatrix(cameraParams, tform1);

% Compute extrinsics of the second camera
cameraPose = rigid3d(orient, loc);
tform2 = cameraPoseToExtrinsics(cameraPose);
camMatrix2 = cameraMatrix(cameraParams, tform2);

% 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', loc, 'Orientation', orient, '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');
```

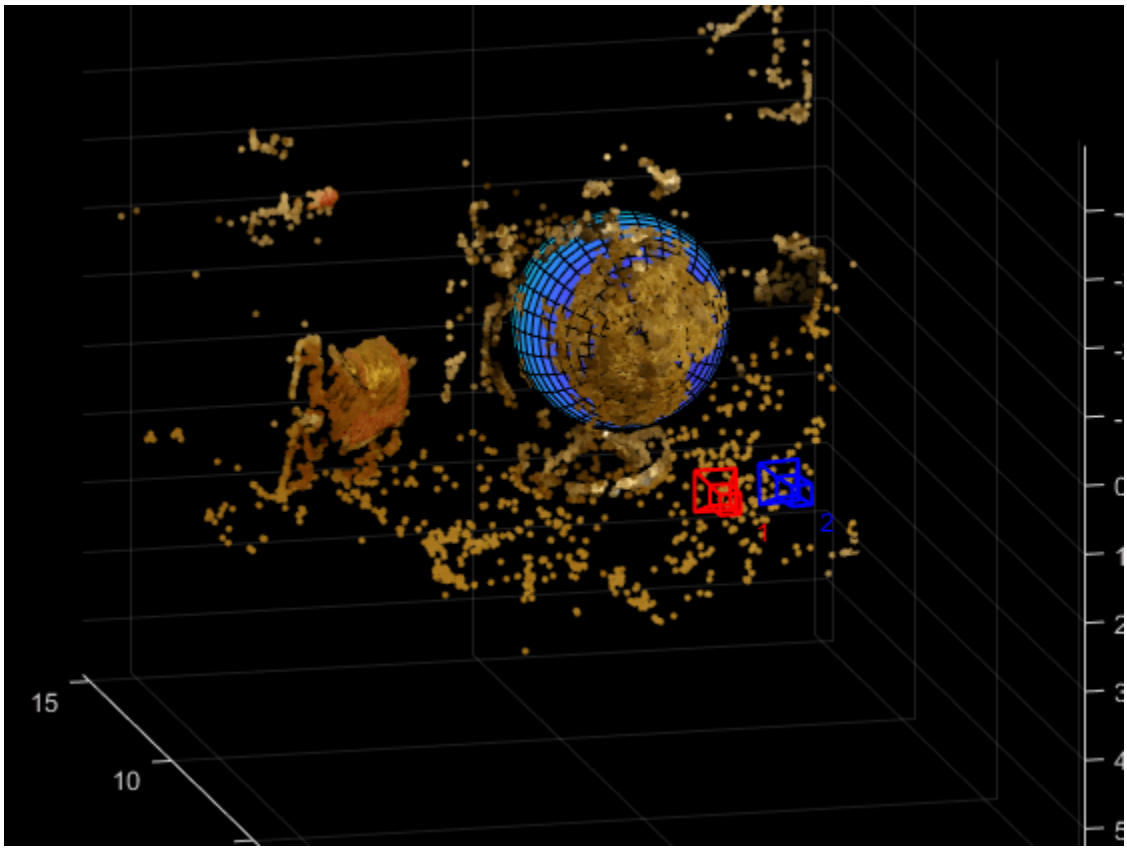
```
zlabel('z-axis')

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
```

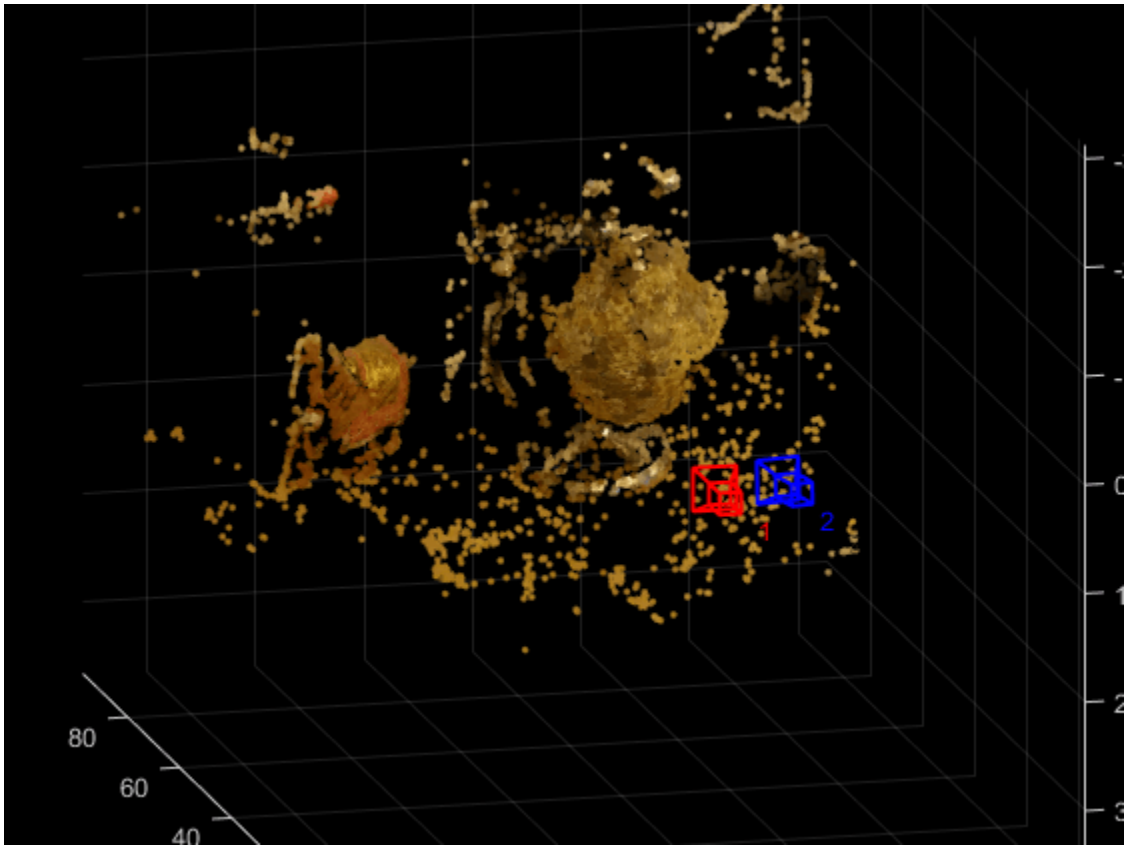


Metric Reconstruction of the Scene

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);
loc = loc * 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', loc, 'Orientation', orient, '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.

Stereo Visual Simultaneous Localization and Mapping

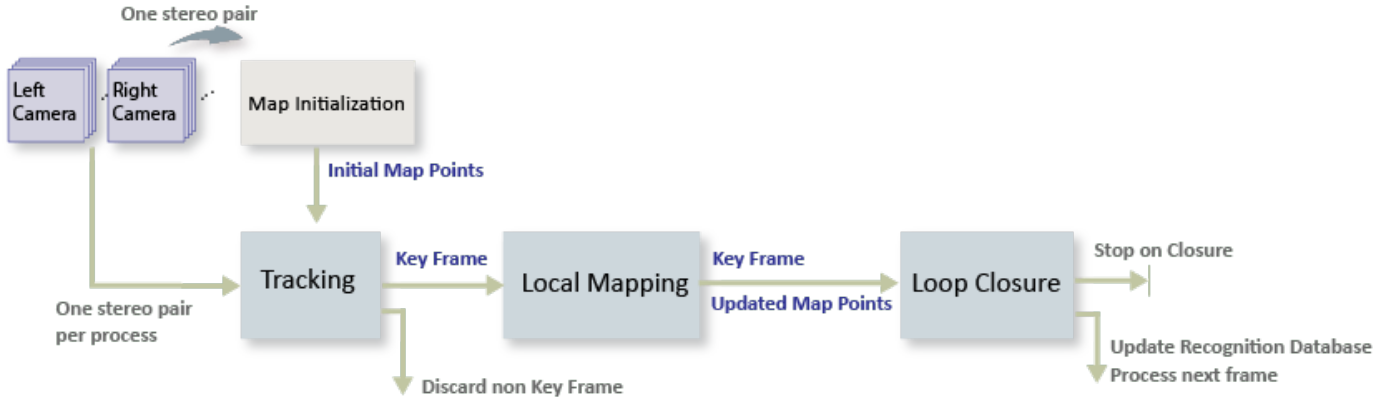
Visual simultaneous localization and mapping (vSLAM), refers to the process of calculating the position and orientation of a camera with respect to its surroundings, while simultaneously mapping the environment. The process uses only visual inputs from the camera. Applications for vSLAM include augmented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

vSLAM can be performed by using just a monocular camera. However, since depth cannot be accurately calculated using a single camera, the scale of the map and the estimated trajectory is unknown and drifts over time. In addition, to bootstrap the system, multiple views are required to produce an initial map as it cannot be triangulated from the first frame. Using a stereo camera solves these problems and provides a more reliable vSLAM solution.

This example shows how to process image data from a stereo camera to build a map of an outdoor environment and estimate the trajectory of the camera. The example uses a version of ORB-SLAM2 [1] on page 1-0 algorithm, which is feature-based and supports stereo cameras.

Overview of Processing Pipeline

The pipeline for stereo vSLAM is very similar to the monocular vSLAM pipeline in the “Monocular Visual Simultaneous Localization and Mapping” on page 1-72 example. The major difference is that in the **Map Initialization** stage 3-D map points are created from a pair of stereo images of the same stereo pair instead of two images of different frames.



- **Map Initialization**: The pipeline starts by initializing the map of 3-D points from a pair of stereo images using the disparity map. The left image is stored as the first key frame.
- **Tracking**: Once a map is initialized, for each new stereo pair, the pose of the camera is estimated by matching features in the left image to features in the last key frame. The estimated camera pose is refined by tracking the local map.
- **Local Mapping**: If the current left image is identified as a key frame, new 3-D map points are computed from the disparity of the stereo pair. 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 Stereo Image Sequence

The data used in this example are from the UTIAS Long-Term Localization and Mapping Dataset provided by University of Toronto Institute for Aerospace Studies. You can download the data to a directory using a web browser or by running the following code:

```
dataFolder   = [fullfile(pwd), filesep, 'stereoImageData'];
zipFileName  = [dataFolder, filesep, 'run_000005.zip'];
folderExists = exist(dataFolder, 'dir');

% Create a folder in the current directory to save the downloaded file
if ~folderExists
    mkdir(dataFolder);
    disp('Downloading run_000005.zip (818 MB). This download can take a few minutes.')
    !wget ftp://asrl3.utias.utoronto.ca/2020-vtr-dataset/UTIAS-In-The-Dark/run_000005.zip -P ./s

    % Extract contents of the downloaded file
    disp('Extracting run_000005.zip (818 MB) ...')
    unzip(zipFileName, dataFolder);
end
```

Use two `imageDatastore` objects to store the stereo images.

```
imgFolderLeft  = [dataFolder,'/images/left/'];
imgFolderRight = [dataFolder,'/images/right/'];
imdsLeft       = imageDatastore(imgFolderLeft);
imdsRight      = imageDatastore(imgFolderRight);

% Inspect the first pair of images
currFrameIdx   = 1;
currILeft      = readimage(imdsLeft, currFrameIdx);
currIRight     = readimage(imdsRight, currFrameIdx);
imshowpair(currILeft, currIRight, 'montage');
```

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 two images of a stereo pair. The matched pairs should satisfy the following constraints:

- The horizontal shift between the two corresponding feature points in the rectified stereo pair image is less than the maximum disparity. You can determine the approximate maximum disparity value from the stereo anaglyph of the stereo pair image. For more information, see Choosing Range of Disparity.
- The vertical shift between the two corresponding feature points in the rectified stereo pair image is less than a threshold.
- The scales of the matched features are nearly identical.

The 3-D world locations corresponding to the matched feature points are determined as follows:

- Use `disparitySGM` to compute the disparity map for each pair of stereo images by using semi-global matching (SGM) method.
- Use `reconstructScene` to compute the 3-D world point coordinates from the disparity map.
- Find the locations in the disparity map that correspond to the feature points and their 3-D world locations.

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

% Load the initial camera pose. The initial camera pose is derived based
% on the transformation between the camera and the vehicle:
% http://asrl.utias.utoronto.ca/datasets/2020-vtr-dataset/text_files/transform_camera_vehicle.tx
initialPoseData = load('initialPose.mat');
initialPose     = initialPoseData.initialPose;

% Construct the reprojection matrix for 3-D reconstruction.
% The intrinsics for the dataset can be found at the following page:
% http://asrl.utias.utoronto.ca/datasets/2020-vtr-dataset/text_files/camera_parameters.txt
focalLength     = [387.777 387.777];     % specified in pixels
principalPoint  = [257.446 197.718];     % specified in pixels [x, y]
baseline        = 0.239965;              % specified in meters
imageSize       = size(currILeft,[1,2]); % in pixels [mrows, ncols]
intrinsics      = cameraIntrinsics(focalLength, principalPoint, imageSize);

reprojectionMatrix = [1, 0, 0, -principalPoint(1);
    0, 1, 0, -principalPoint(2);
    0, 0, 0, focalLength(1);
    0, 0, 1/baseline, 0];

% In this example, the images are already undistorted and rectified. In a general workflow,
% uncomment the following code to undistort and rectify the images.
% currILeft     = undistortImage(currILeft, intrinsics);
% currIRight    = undistortImage(currIRight, intrinsics);
% stereoParams  = stereoParameters(intrinsics, intrinsics, eye(3), [-baseline, 0 0]);
% [currILeft, currIRight] = rectifyStereoImages(currILeft, currIRight, stereoParams, 'OutputView

% Detect and extract ORB features from the rectified stereo images
scaleFactor = 1.2;
numLevels   = 8;
[currFeaturesLeft,  currPointsLeft]   = helperDetectAndExtractFeatures(currILeft, scaleFactor, nu
[currFeaturesRight, currPointsRight]  = helperDetectAndExtractFeatures(currIRight, scaleFactor, n

% Match feature points between the stereo images and get the 3-D world positions
disparityRange = [0 48];  % specified in pixels
[xyzPoints, matchedPairs] = helperReconstructFromStereo(currILeft, currIRight, ...
    currFeaturesLeft, currFeaturesRight, currPointsLeft, currPointsRight, reprojectionMatrix, ini
```

Data Management and Visualization

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

```
% 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(xyzPoints, 1));

% Add the first key frame
currKeyFrameId = 1;
```

```
vSetKeyFrames = addView(vSetKeyFrames, currKeyFrameId, initialPose, 'Points', currPointsLeft,...
    'Features', currFeaturesLeft.Features);

% Add 3-D map points
[mapPointSet, stereoMapPointsIdx] = addWorldPoints(mapPointSet, xyzPoints);

% Add observations of the map points
mapPointSet = addCorrespondences(mapPointSet, currKeyFrameId, stereoMapPointsIdx, matchedPairs(:

% Visualize matched features in the first key frame
featurePlot = helperVisualizeMatchedFeaturesStereo(currILeft, currIRight, currPointsLeft, ...
    currPointsRight, matchedPairs);
```

```
% Visualize initial map points and camera trajectory
mapPlot     = helperVisualizeMotionAndStructureStereo(vSetKeyFrames, mapPointSet);

% Show legend
showLegend(mapPlot);
```

Initialize Place Recognition Database

Loop detection is performed using the bags-of-words approach. A visual vocabulary represented as a `bagOfFeatures` object is created offline with the ORB descriptors extracted from a large set of images in the dataset by calling:

```
bag = bagOfFeatures(imds,'CustomExtractor',
@helperORBFeatureExtractorFunction,'TreeProperties',[5, 10],
'StrongestFeatures',1);
```

where `imds` is an `imageDatastore` object storing the training images and `helperORBFeatureExtractorFunction` is the ORB feature extractor function. See “Image Retrieval with Bag of Visual Words” on page 17-110 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 ORB features.

```
% Load the bag of features data created offline
bofData         = load('bagOfFeaturesDataSLAM.mat');

% Initialize the place recognition database
loopDatabase    = invertedImageIndex(bofData.bof,"SaveFeatureLocations", false);

% Add features of the first key frame to the database
addImageFeatures(loopDatabase, currFeaturesLeft, currKeyFrameId);
```

Tracking

The tracking process is performed using every pair and determines when to insert a new key frame.

```
% ViewId of the last key frame
lastKeyFrameId    = currKeyFrameId;

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

% Indices of all the key frames in the input image sequence
addedFramesIdx    = lastKeyFrameIdx;

currFrameIdx      = 2;
isLoopClosed      = false;
```

Each frame is processed as follows:

1. ORB features are extracted for each new stereo pair of images 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`.

Given the camera pose, project the map points observed by the last key frame into the current frame and search for feature correspondences using `matchFeaturesInRadius`.

1. With 3-D to 2-D correspondences in the current frame, refine the camera pose by performing a motion-only bundle adjustment using `bundleAdjustmentMotion`.
2. Project the local map points into the current frame to search for more feature correspondences using `matchFeaturesInRadius` and refine the camera pose again using `bundleAdjustmentMotion`.
3. The last step of tracking is to decide if the current frame should be a new key frame. A frame is a key frame if both of the following conditions are satisfied:

- At least 5 frames have passed since the last key frame or the current frame tracks fewer than 100 map points.
- The map points tracked by the current frame are fewer than 90% of points tracked by the reference key frame.

If the current frame is to become a key frame, continue to the **Local Mapping** process. Otherwise, start **Tracking** for the next frame.

```
% Main loop
isLastFrameKeyFrame = true;
while ~isLoopClosed && currFrameIdx <= numel(imdsLeft.Files)
```

```
currILeft  = readimage(imdsLeft, currFrameIdx);
currIRight = readimage(imdsRight, currFrameIdx);

[currFeaturesLeft, currPointsLeft]    = helperDetectAndExtractFeatures(currILeft, scaleFacto
[currFeaturesRight, currPointsRight]  = helperDetectAndExtractFeatures(currIRight, scaleFact

% Track the last key frame
% trackedMapPointsIdx:  Indices of the map points observed in the current left frame
% trackedFeatureIdx:    Indices of the corresponding feature points in the current left frame
[currPose, trackedMapPointsIdx, trackedFeatureIdx] = helperTrackLastKeyFrame(mapPointSet, ..
    vSetKeyFrames.Views, currFeaturesLeft, currPointsLeft, lastKeyFrameId, intrinsics, scaleF

if isempty(currPose) || numel(trackedMapPointsIdx) < 30
    currFrameIdx = currFrameIdx + 1;
    continue
end

% Track the local map  and check if the current frame is a key frame.
% localKeyFrameIds:   ViewId of the connected key frames of the current frame
numSkipFrames     = 5;
numPointsKeyFrame = 100;
[localKeyFrameIds, currPose, trackedMapPointsIdx, trackedFeatureIdx, isKeyFrame] = ...
    helperTrackLocalMap(mapPointSet, directionAndDepth, vSetKeyFrames, trackedMapPointsIdx,
    trackedFeatureIdx, currPose, currFeaturesLeft, currPointsLeft, intrinsics, scaleFactor, n
    isLastFrameKeyFrame, lastKeyFrameIdx, currFrameIdx, numSkipFrames, numPointsKeyFrame);

% Match feature points between the stereo images and get the 3-D world positions
[xyzPoints, matchedPairs] = helperReconstructFromStereo(currILeft, currIRight, currFeaturesLe
    currFeaturesRight, currPointsLeft, currPointsRight, reprojectionMatrix, currPose, dispari

% Visualize matched features in the stereo image
updatePlot(featurePlot, currILeft, currIRight, currPointsLeft, currPointsRight, trackedFeatur

if ~isKeyFrame && currFrameIdx < numel(imdsLeft.Files)
    currFrameIdx = currFrameIdx + 1;
    isLastFrameKeyFrame = false;
    continue
else
    [untrackedFeatureIdx, ia] = setdiff(matchedPairs(:, 1), trackedFeatureIdx);
    xyzPoints = xyzPoints(ia, :);
    isLastFrameKeyFrame = true;
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, currFeaturesLeft, currPointsLeft, trackedMapPointsIdx, trackedFeatureIdx, loca

% Remove outlier map points that are observed in fewer than 3 key frames
if currKeyFrameId == 2
    triangulatedMapPointsIdx = [];
end

[mapPointSet, directionAndDepth, trackedMapPointsIdx] = ...
    helperCullRecentMapPoints(mapPointSet, directionAndDepth, trackedMapPointsIdx, triangula
    stereoMapPointsIdx);

% Add new map points computed from disparity
[mapPointSet, stereoMapPointsIdx] = addWorldPoints(mapPointSet, xyzPoints);
mapPointSet = addCorrespondences(mapPointSet, currKeyFrameId, stereoMapPointsIdx, ...
    untrackedFeatureIdx);

% Create new map points by triangulation
minNumMatches = 20;
minParallax   = 0.35;
[mapPointSet, vSetKeyFrames, triangulatedMapPointsIdx, stereoMapPointsIdx] = helperCreateNewM
    mapPointSet, vSetKeyFrames, currKeyFrameId, intrinsics, scaleFactor, minNumMatches, minPa
    untrackedFeatureIdx, stereoMapPointsIdx);

% Update view direction and depth
directionAndDepth = update(directionAndDepth, mapPointSet, vSetKeyFrames.Views, ...
    [trackedMapPointsIdx; triangulatedMapPointsIdx; stereoMapPointsIdx], true);

% Local bundle adjustment
[mapPointSet, directionAndDepth, vSetKeyFrames, triangulatedMapPointsIdx, stereoMapPointsIdx]
    helperLocalBundleAdjustmentStereo(mapPointSet, directionAndDepth, vSetKeyFrames, ...
    currKeyFrameId, intrinsics, triangulatedMapPointsIdx, stereoMapPointsIdx);

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

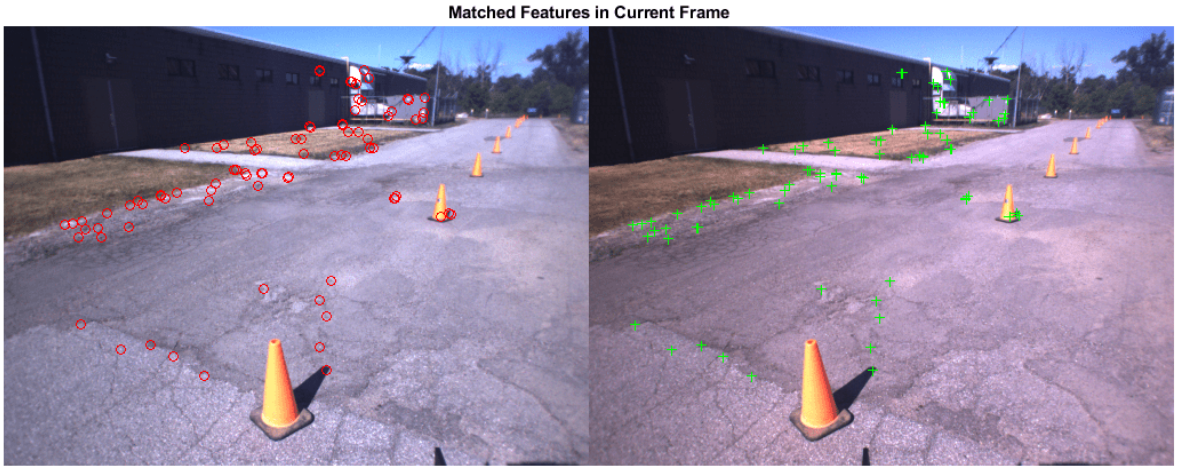
Loop Closure

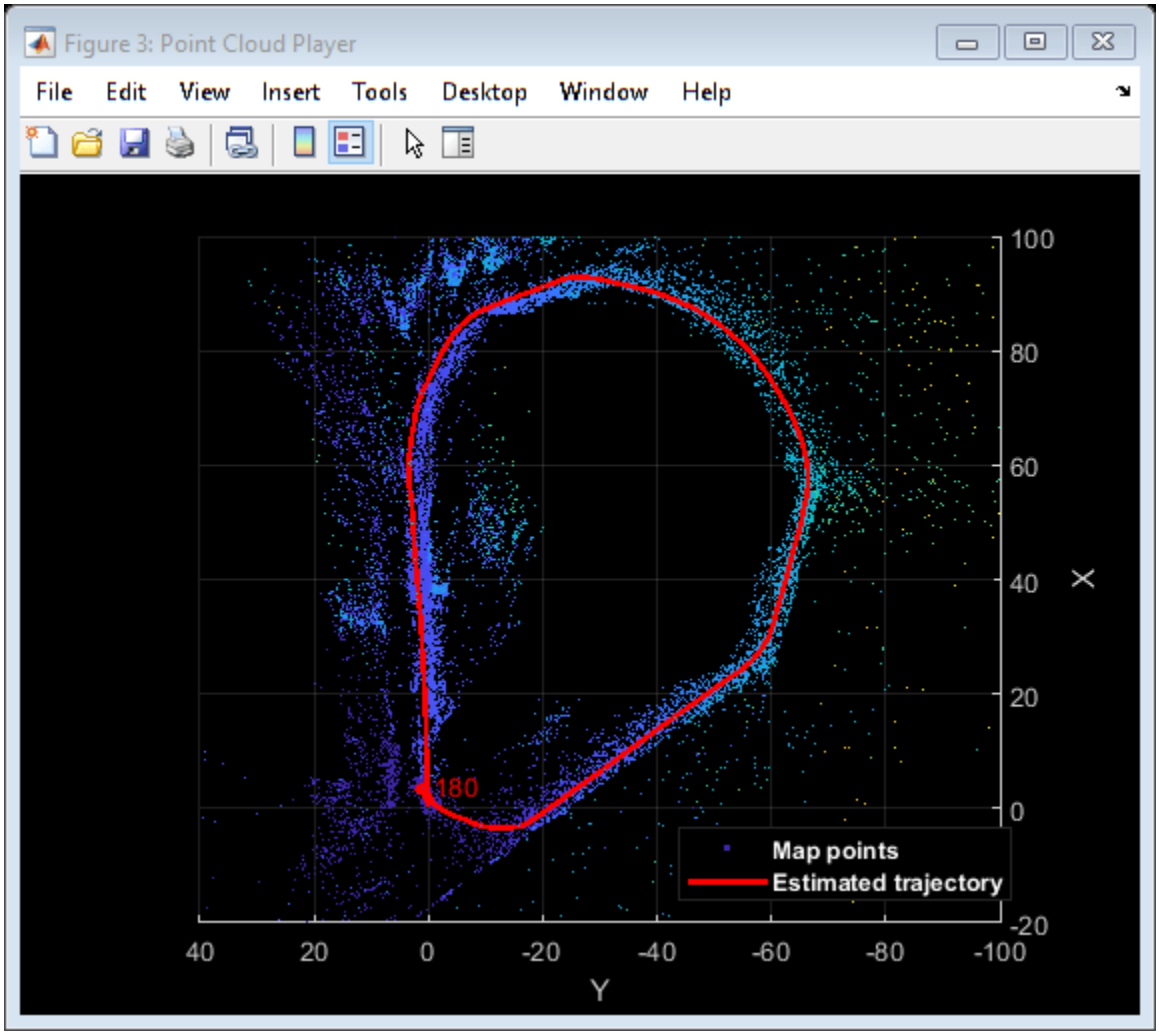
The loop closure step takes the current key frame processed by the local mapping process and tries to detect and close the loop. Loop candidates are identified by querying images in the database that are visually similar to the current key frame using `evaluateImageRetrieval`. A candidate key frame is 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 closure candidate is found, compute the relative pose between the loop closure candidate frame and the current key frame using `estimateGeometricTransform3D`. Then add the loop connection with the relative pose and update `mapPointSet` and `vSetKeyFrames`.

```
% Check loop closure after some key frames have been created
if currKeyFrameId > 50

    % Minimum number of feature matches of loop edges
```

```
            loopEdgeNumMatches = 50;

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

            isTooCloseView = currKeyFrameId - max(validLoopCandidates) < 100;
            if isDetected && ~isTooCloseView
                % Add loop closure connections
                [isLoopClosed, mapPointSet, vSetKeyFrames] = helperAddLoopConnectionsStereo(...
                    mapPointSet, vSetKeyFrames, validLoopCandidates, currKeyFrameId, ...
                    currFeaturesLeft, loopEdgeNumMatches);
            end
        end

        % If no loop closure is detected, add current features into the database
        if ~isLoopClosed
            addImageFeatures(loopDatabase,  currFeaturesLeft, currKeyFrameId);
        end

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






```
Loop edge added between keyframe: 4 and 180
Loop edge added between keyframe: 3 and 180
Loop edge added between keyframe: 6 and 180
```

Finally, apply pose graph optimization over the essential graph in `vSetKeyFrames` to correct the drift. 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.

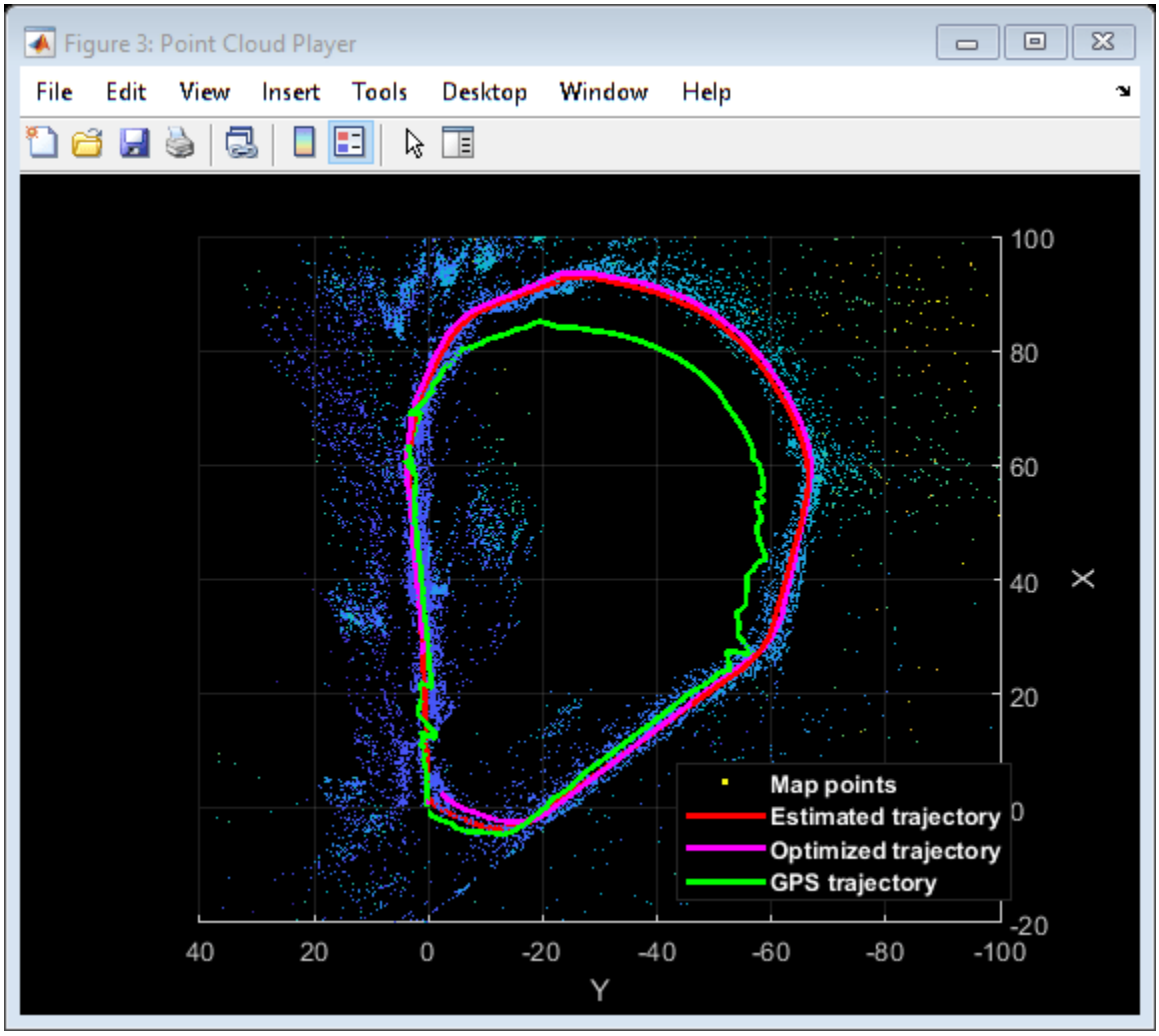
```
% Optimize the poses
vSetKeyFramesOptim = optimizePoses(vSetKeyFrames, minNumMatches, 'Tolerance', 1e-16);

% Update map points after optimizing the poses
mapPointSet = helperUpdateGlobalMap(mapPointSet, directionAndDepth, vSetKeyFrames, vSetKeyFramesO

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 the solution. The downloaded data contains a `gps.txt` file that stores the GPS location for each frame. You can convert the GPS location from geographic to local Cartesian coordinates using `latlon2local` (Automated Driving Toolbox) from Automated Driving Toolbox or `geodetic2enu` (Mapping Toolbox) from Mapping Toolbox. In this example, you can simply load the converted GPS data from an M-file.

```
% Load GPS data
gpsData     = load('gpsLocation.mat');
gpsLocation = gpsData.gpsLocation;

% Transform GPS locations to the reference coordinate system
gTruth = helperTransformGPSLocations(gpsLocation, optimizedPoses);

% Plot the GPS locations
plotActualTrajectory(mapPlot, gTruth(addedFramesIdx, :));

% Show legend
showLegend(mapPlot);
```

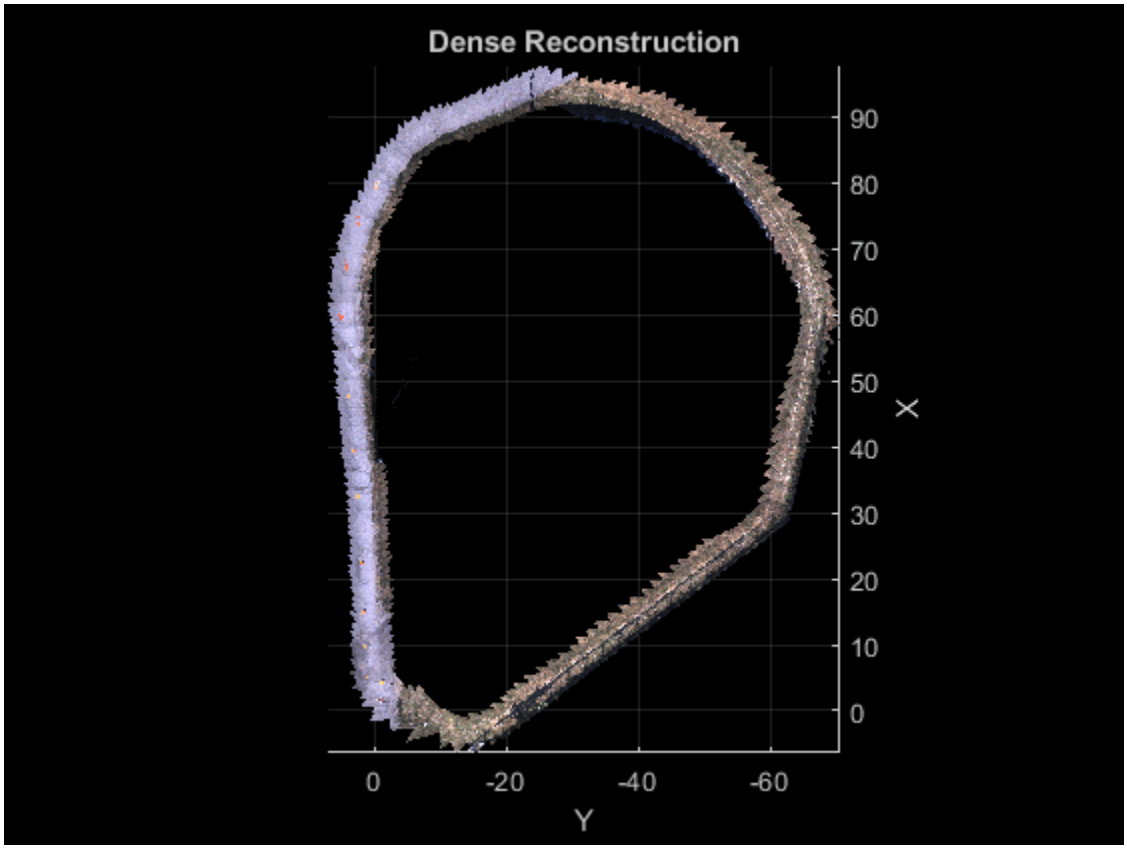


Dense Reconstruction from Stereo Images

Given the refined camera poses, you can perform dense reconstruction from the stereo images corresponding to the key frames.

```
% Create an array of pointCloud objects to store the 3-D world points
% associated with the key frames
ptClouds =  repmat(pointCloud(zeros(1, 3)), numel(addedFramesIdx), 1);

for i = 1: numel(addedFramesIdx)
    ILeft  = readimage(imdsLeft, addedFramesIdx(i));
    IRight = readimage(imdsRight, addedFramesIdx(i));

    % Reconstruct scene from disparity
    disparityMap = disparitySGM(im2gray(ILeft), im2gray(IRight), 'DisparityRange', disparityRange
    xyzPoints    = reconstructScene(disparityMap, reprojectionMatrix);

    % Ignore the upper half of the images which mainly show the sky
    xyzPoints(1:floor(imageSize(1)/2), :, :) = NaN;

    % Ignore the lower part of the images which show the vehicle
```

```
    xyzPoints(imageSize(1)-50:end, :, :) = NaN;

    xyzPoints  = reshape(xyzPoints, [imageSize(1)*imageSize(2), 3]);

    % Get color from the color image
    colors = reshape(ILeft, [imageSize(1)*imageSize(2), 3]);

    % Remove world points that are too far away from the camera
    validIndex = xyzPoints(:, 3) > 0 & xyzPoints(:, 3) < 100/reprojectionMatrix(4, 3);
    xyzPoints  = xyzPoints(validIndex, :);
    colors     = colors(validIndex, :);

    % Transform world points to the world coordinates
    currPose   = optimizedPoses.AbsolutePose(i);
    xyzPoints  = xyzPoints * currPose.Rotation + currPose.Translation;
    ptCloud    = pointCloud(xyzPoints, 'Color', colors);

    % Downsample the point cloud
    ptClouds(i) = pcdownsample(ptCloud, 'random', 0.5);
end

% Concatenate the point clouds
pointCloudsAll = pccat(ptClouds);

% Visualize the point cloud
figure
ax = pcshow(pointCloudsAll,'VerticalAxis', 'y', 'VerticalAxisDir', 'down');
xlabel('X')
ylabel('Y')
zlabel('Z')
title('Dense Reconstruction')

% Display the bird's eye view of the scene
view(ax, [0 0 1]);
camroll(ax, 90);
```



Supporting Functions

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

`helperDetectAndExtractFeatures` detect and extract and ORB features from the image.

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

numPoints = 1500;

% Detect ORB features
Igray  = im2gray(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
```

`helperReconstructFromStereo` reconstruct scene from stereo image using the disparity map

```
function [xyzPoints, indexPairs] = helperReconstructFromStereo(I1, I2, ...
    features1, features2, points1, points2, reprojectionMatrix, currPose, disparityRange)

indexPairs     = helperFindValidFeaturePairs(features1, features2, points1, points2, disparityRa
```

```
% Compute disparity for all pixels in the left image. In practice, it is more
% common to compute disparity just for the matched feature points.
disparityMap   = disparitySGM(rgb2gray(I1), rgb2gray(I2), 'DisparityRange', disparityRange);
xyzPointsAll   = reconstructScene(disparityMap, reprojectionMatrix);

% Find the corresponding world point of the matched feature points
locations      = floor(points1.Location(indexPairs(:, 1), [2 1]));
xyzPoints      = [];
isPointFound   = false(size(points1));

for i = 1:size(locations, 1)
    point3d = squeeze(xyzPointsAll(locations(i,1), locations(i, 2), :))';
    isPointValid   = point3d(3) > 0 & point3d(3) < 200/reprojectionMatrix(4, 3);
    if isPointValid
        xyzPoints       = [xyzPoints; point3d]; %#ok<*AGROW>
        isPointFound(i) = true;
    end
end
indexPairs = indexPairs(isPointFound, :);
xyzPoints  = xyzPoints * currPose.Rotation + currPose.Translation;
end
```

`helperFindValidFeaturePairs` match features between a pair of stereo images

```
function indexPairs = helperFindValidFeaturePairs(features1, features2, points1, points2, dispari
indexPairs  = matchFeatures(features1, features2,...
    'Unique', true, 'MaxRatio', 1, 'MatchThreshold', 40);

matchedPoints1 = points1.Location(indexPairs(:,1), :);
matchedPoints2 = points2.Location(indexPairs(:,2), :);
scales1        = points1.Scale(indexPairs(:,1), :);
scales2        = points2.Scale(indexPairs(:,2), :);

dist2EpipolarLine = abs(matchedPoints2(:, 2) - matchedPoints1(:, 2));
shiftDist = matchedPoints1(:, 1) - matchedPoints2(:, 1);

isCloseToEpipolarline = dist2EpipolarLine < 2*scales2;
isDisparityValid      = shiftDist > 0 & shiftDist < disparityRange(2);
isScaleIdentical      = scales1 == scales2;
indexPairs = indexPairs(isCloseToEpipolarline & isDisparityValid & isScaleIdentical, :);
end
```

`helperCullRecentMapPoints` cull recently added map points.

```
function [mapPointSet, directionAndDepth, mapPointsIdx] = ...
    helperCullRecentMapPoints(mapPointSet, directionAndDepth, mapPointsIdx, newPointIdx, stereoMa

outlierIdx = setdiff([newPointIdx; stereoMapPointsIndices], mapPointsIdx);

if ~isempty(outlierIdx)
    mapPointSet   = removeWorldPoints(mapPointSet, outlierIdx);
    directionAndDepth = remove(directionAndDepth, outlierIdx);
    mapPointsIdx  = mapPointsIdx - arrayfun(@(x) nnz(x>outlierIdx), mapPointsIdx);
end
end
```

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

```
function [mapPointSet, directionAndDepth] = helperUpdateGlobalMap(...
    mapPointSet, directionAndDepth, vSetKeyFrames, vSetKeyFramesOptim)

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
```

`helperTransformGPSLocations` transform the GPS locations to the reference coordinate system

```
function gTruth = helperTransformGPSLocations(gpsLocations, optimizedPoses)

initialYawGPS  = atan( (gpsLocations(100, 2) - gpsLocations(1, 2)) / ...
    (gpsLocations(100, 1) - gpsLocations(1, 1)));
initialYawSLAM = atan((optimizedPoses.AbsolutePose(50).Translation(2) - ...
    optimizedPoses.AbsolutePose(1).Translation(2)) / ...
    (optimizedPoses.AbsolutePose(59).Translation(1) - ...
    optimizedPoses.AbsolutePose(1).Translation(1)));

relYaw = initialYawGPS - initialYawSLAM;
relTranslation = optimizedPoses.AbsolutePose(1).Translation;

initialTform = rotationVectorToMatrix([0 0 relYaw]);
for i = 1:size(gpsLocations, 1)
    gTruth(i, :) =  initialTform * gpsLocations(i, :)' + relTranslation';
end
end
```

References

[1] Mur-Artal, Raul, and Juan D. Tardós. "ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras." *IEEE Transactions on Robotics* 33, no. 5 (2017): 1255-1262.

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

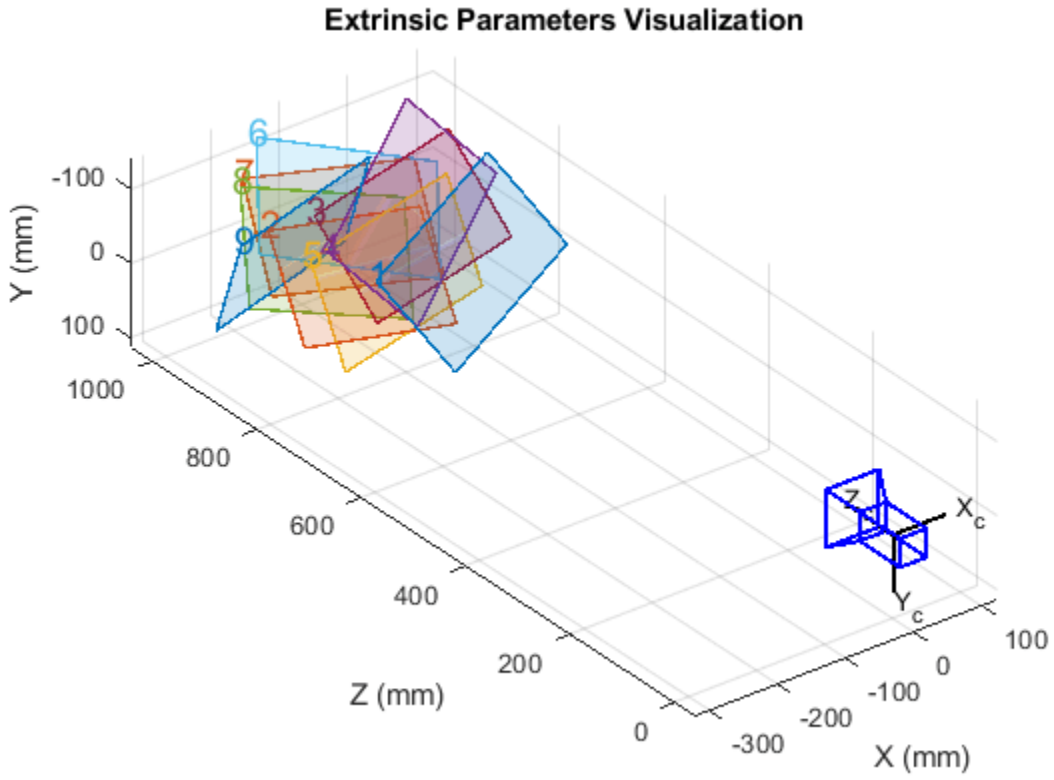
Estimate camera parameters using a set of images of a checkerboard calibration pattern.

```
% 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);
```

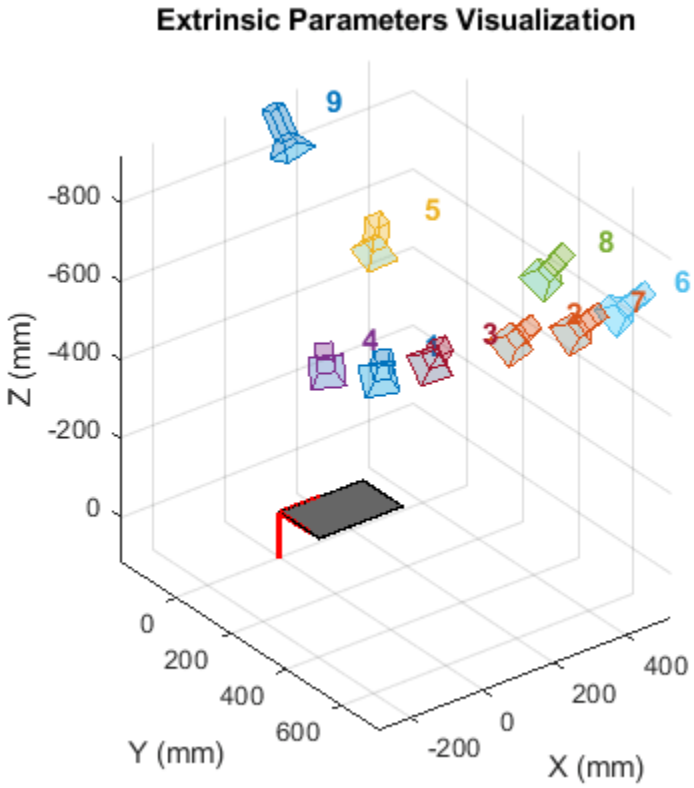
Extrinsics

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');
```



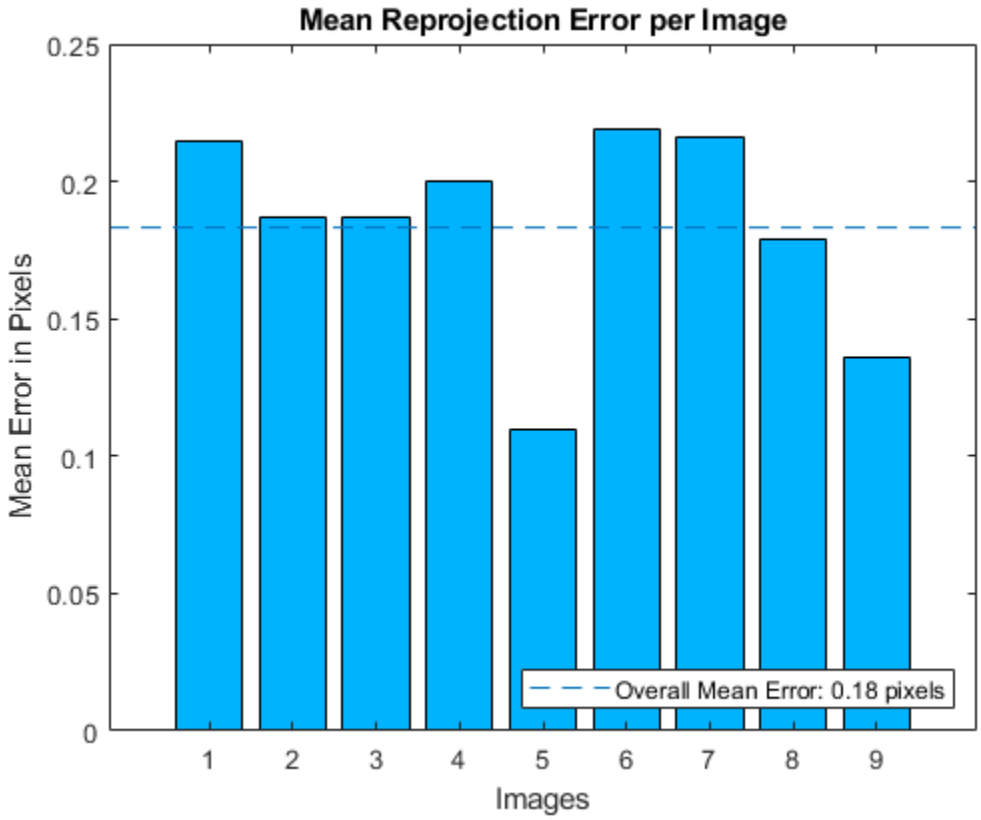

```
figure;
showExtrinsics(params, 'PatternCentric');
```



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 +/- 1.96σ 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      710.3785 +/- 4.0579  ]
Principal point (pixels):[  563.6481 +/- 5.3967      355.7252 +/- 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 +/- 0.0024
                         [   -0.7283 +/- 0.0050       -0.0996 +/- 0.0072        0.1964 +/- 0.0027
                         [   -0.6722 +/- 0.0051       -0.1444 +/- 0.0074       -0.1329 +/- 0.0026
                         [   -0.5836 +/- 0.0056       -0.2901 +/- 0.0074       -0.5622 +/- 0.0025
```

```
                          [   -0.3157 +/- 0.0065       -0.1441 +/- 0.0075       -0.1067 +/- 0.0011
                          [   -0.7581 +/- 0.0052        0.1947 +/- 0.0072        0.4324 +/- 0.0030
                          [   -0.7515 +/- 0.0051        0.0767 +/- 0.0072        0.2070 +/- 0.0029
                          [   -0.6223 +/- 0.0053        0.0231 +/- 0.0073        0.3663 +/- 0.0024
                          [    0.3443 +/- 0.0063       -0.2226 +/- 0.0073       -0.0437 +/- 0.0014

Translation vectors (mm):
                          [ -146.0517 +/- 6.0391      -26.8685 +/- 3.7318      797.9026 +/- 3.9002
                          [ -209.4358 +/- 6.9637      -59.4565 +/- 4.3578      921.8198 +/- 4.6295
                          [ -129.3825 +/- 7.0907      -44.1030 +/- 4.3751      937.6831 +/- 4.4913
                          [ -151.0049 +/- 6.6905      -27.3253 +/- 4.1339      884.2788 +/- 4.3925
                          [ -174.9500 +/- 6.7056      -24.3499 +/- 4.1606      886.4961 +/- 4.6686
                          [ -134.3097 +/- 7.8887     -103.4981 +/- 4.8925     1042.4553 +/- 4.8184
                          [ -173.9846 +/- 7.6891      -73.1691 +/- 4.7812     1017.2385 +/- 4.8126
                          [ -202.9448 +/- 7.4327      -87.9091 +/- 4.6482      983.6957 +/- 4.9072
                          [ -319.8862 +/- 6.3213     -119.8898 +/- 4.0922      829.4581 +/- 4.9591
```

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.

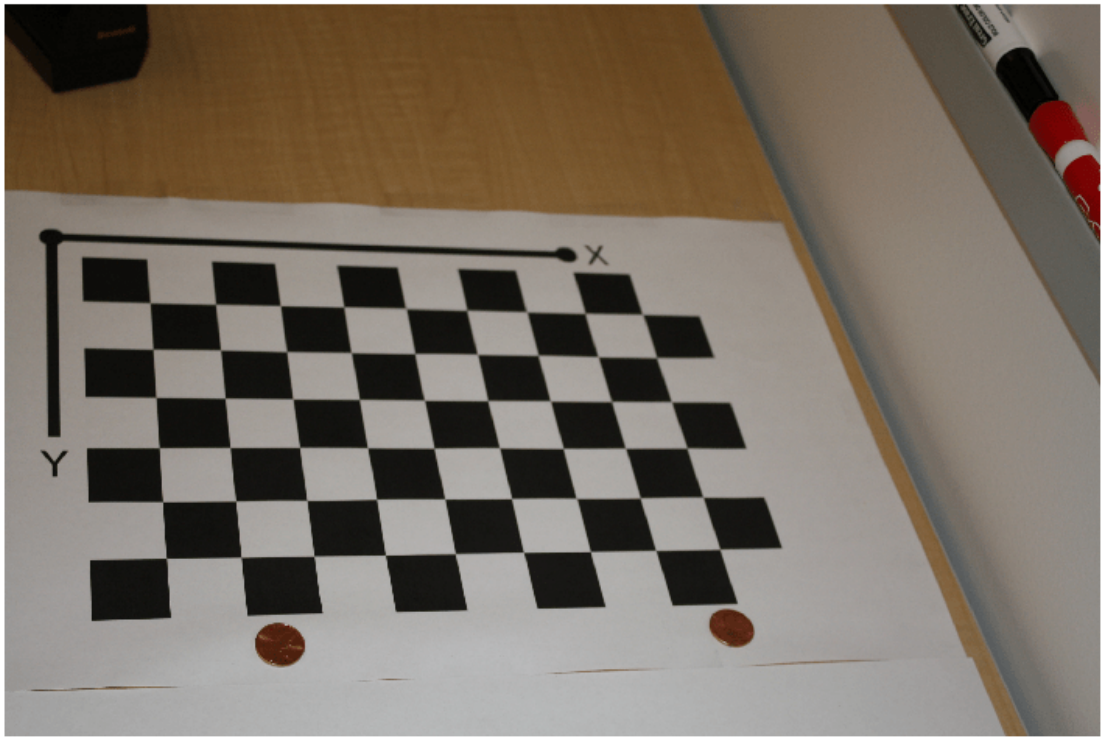
The pattern must be affixed to a flat surface, and it should be at approximately the same distance from the camera as the objects you want to measure. The size of a square must be measured in world units, for example millimeters, as precisely as possible. In this example we use 9 images of the pattern,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

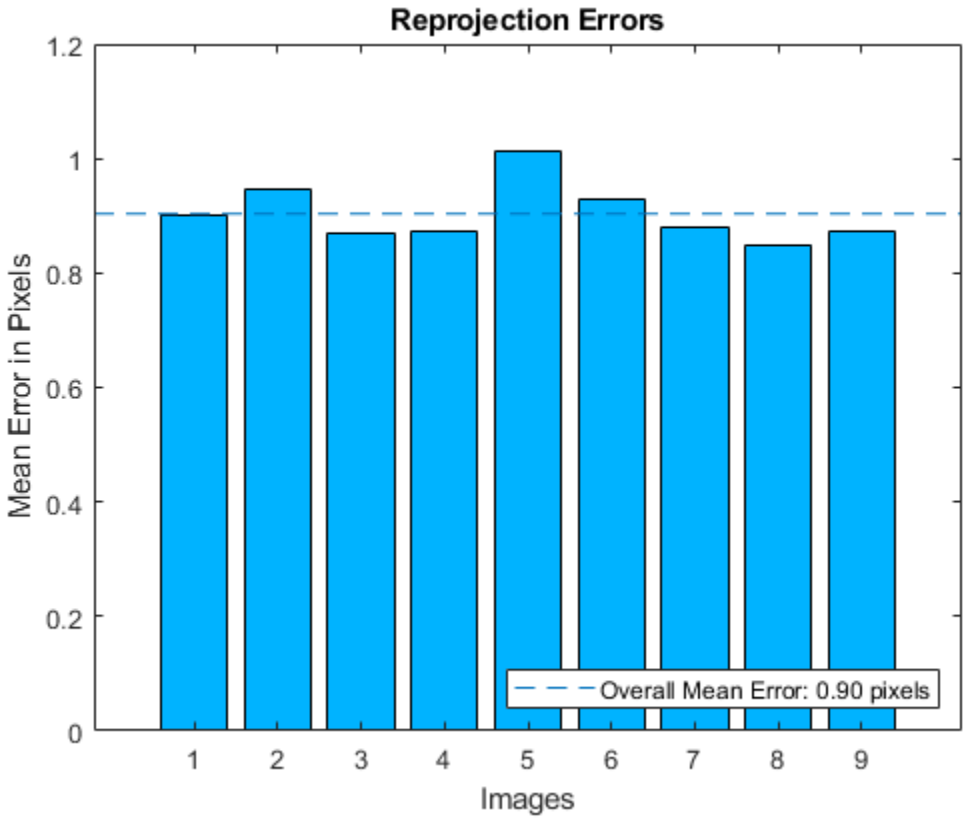
```
numImages = 9;
files = cell(1, numImages);
for i = 1:numImages
    files{i} = fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', ...
        'calibration', 'slr', sprintf('image%d.jpg', i));
end

% 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');
```



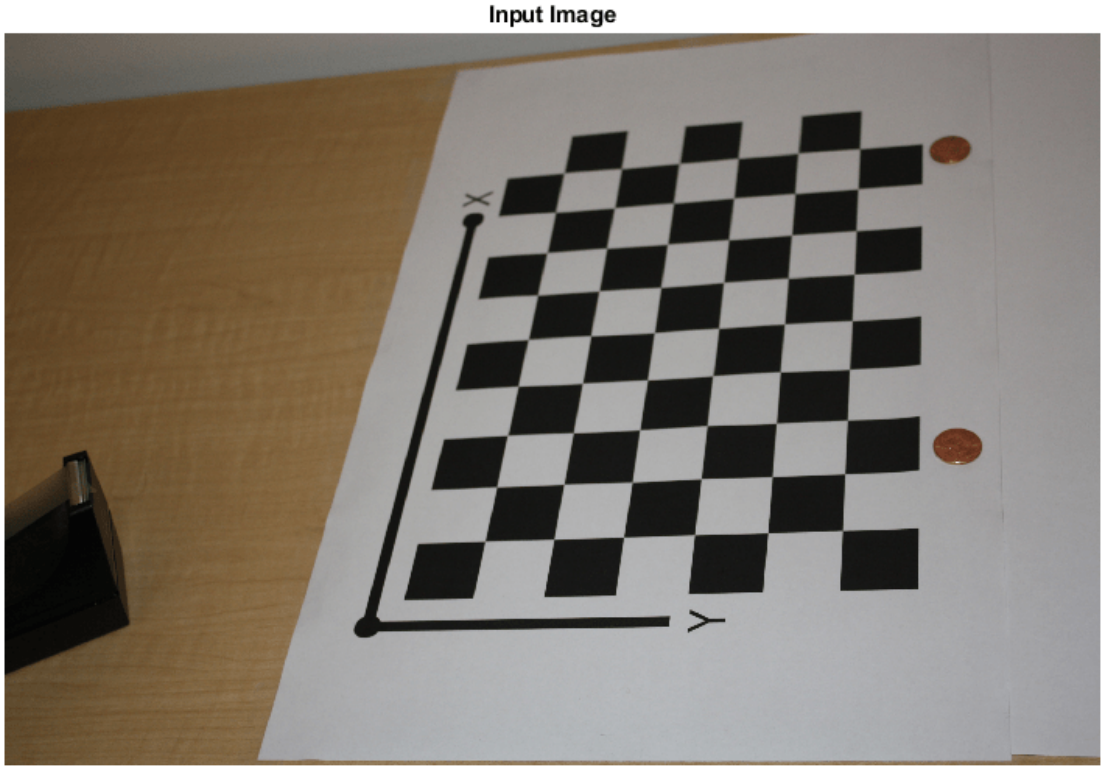
The bar graph indicates the accuracy of the calibration. Each bar shows the mean reprojection error for the corresponding calibration image. The reprojection errors are the distances between the corner points detected in the image, and the corresponding ideal world points projected into the image.

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.

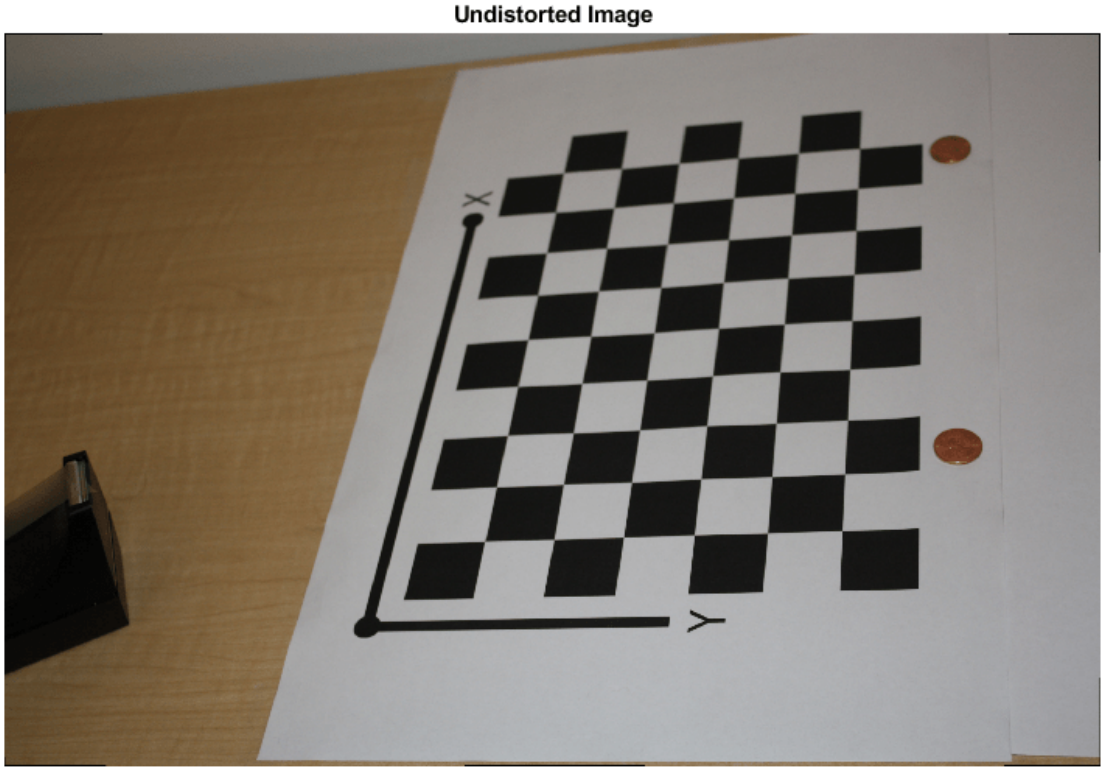
```
imOrig = imread(fullfile(matlabroot, 'toolbox', 'vision', 'visiondata', ...
        'calibration', 'slr', 'image9.jpg'));
figure; imshow(imOrig, 'InitialMagnification', magnification);
title('Input Image');
```



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 undistored. 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');
```

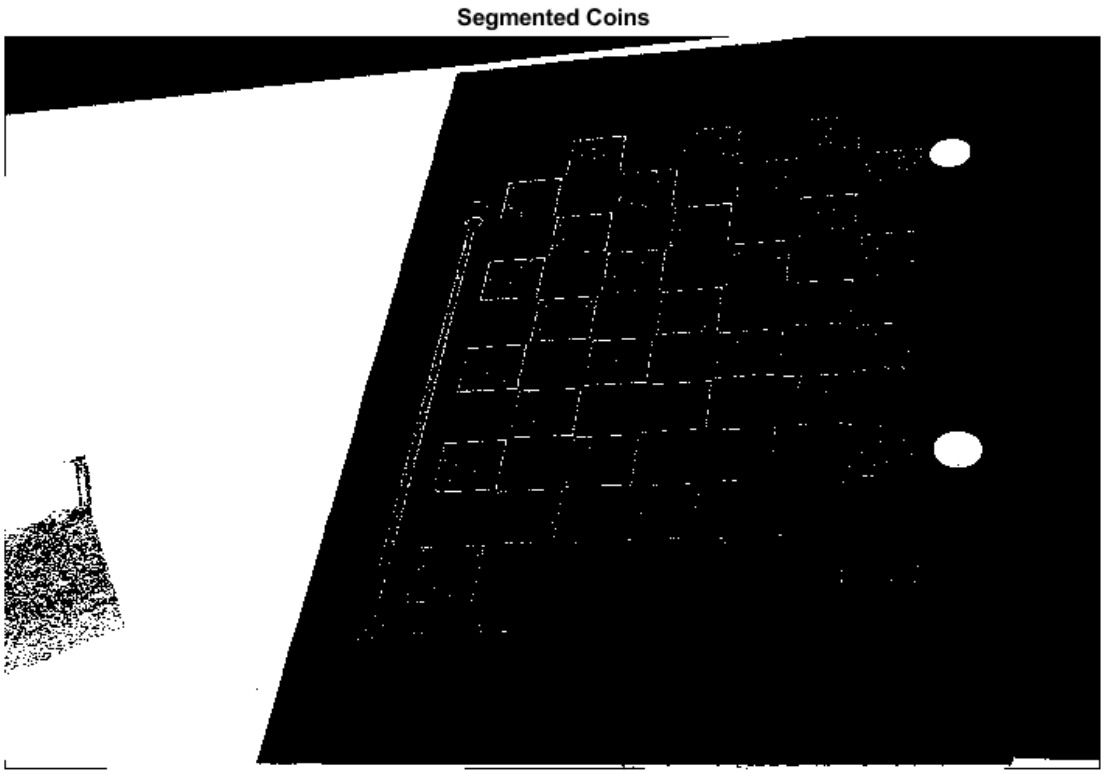


Note that this image exhibits very little lens distortion. The undistortion step is far more important if you use a wide-angle lens, or a low-end webcam.

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');
```

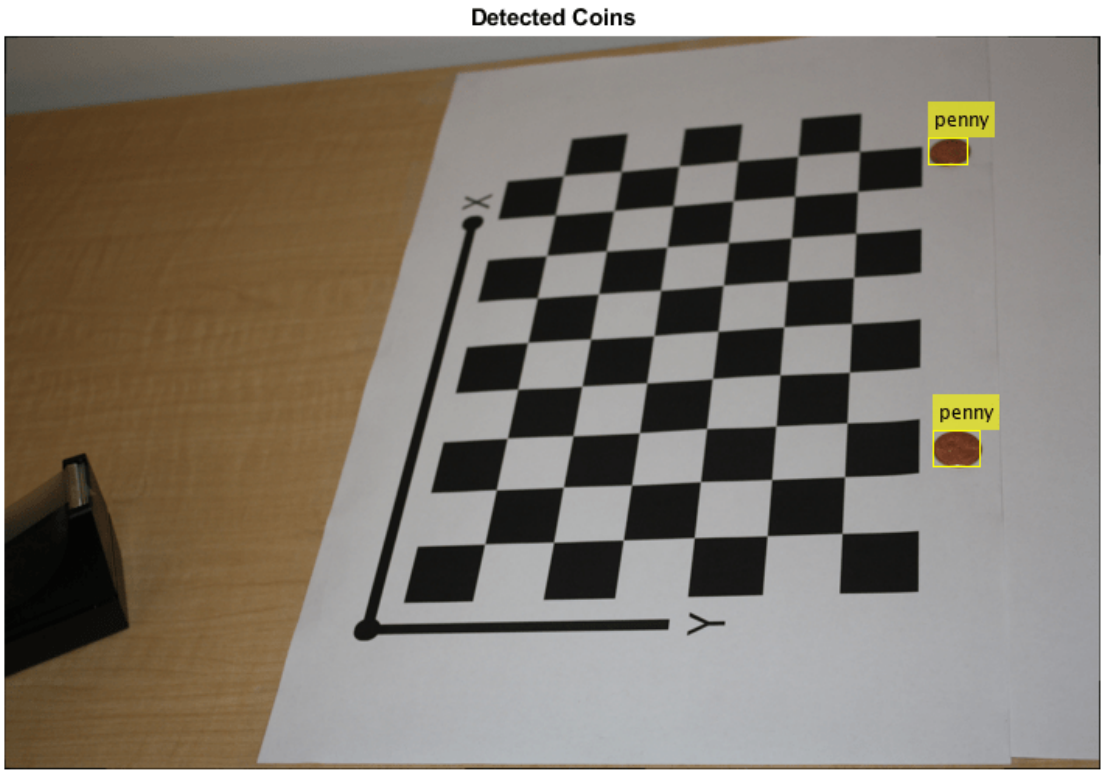


Detect Coins

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 Extrinsics

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.

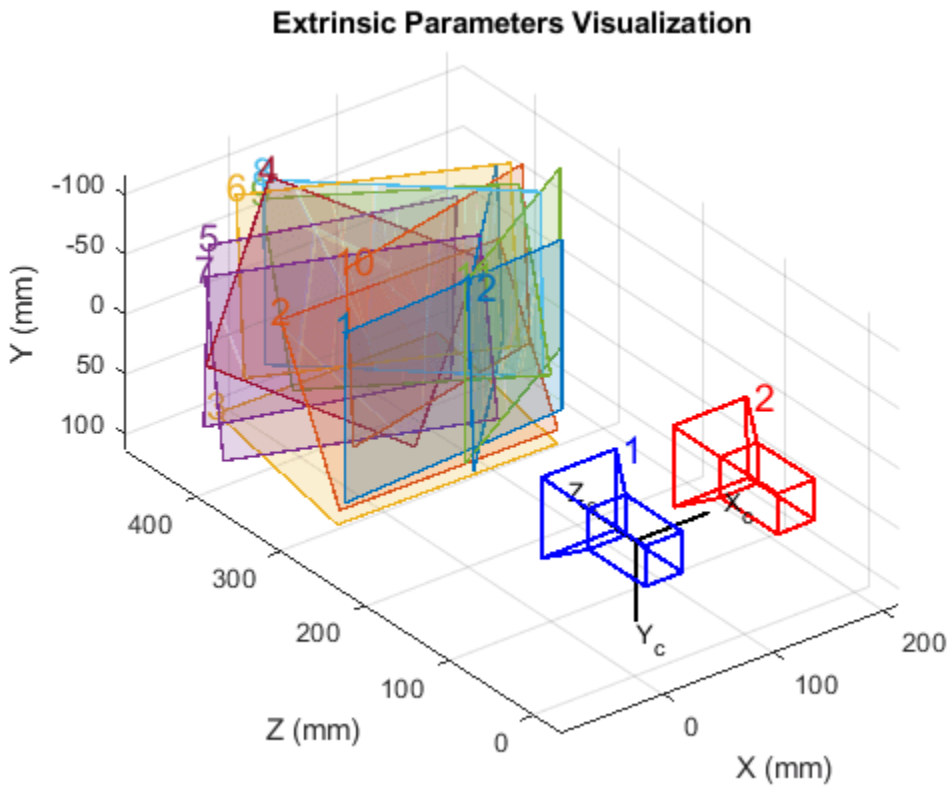
Depth Estimation From Stereo Video

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

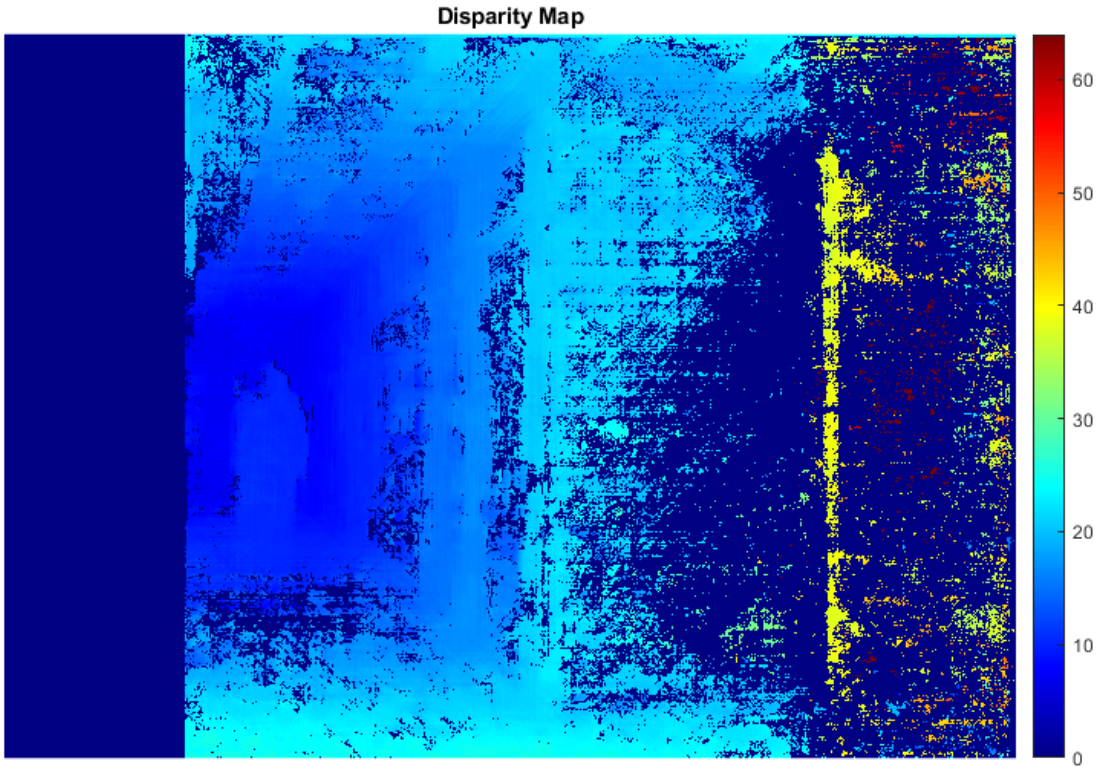
The frames from the left and the right cameras must be rectified in order to compute disparity and reconstruct the 3-D scene. Rectified images have horizontal epipolar lines, and are row-aligned. This simplifies the computation of disparity by reducing the search space for matching points to one dimension. 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, reprojectionMatrix] = ...
    rectifyStereoImages(frameLeft, frameRight, stereoParams);

figure;
imshow(stereoAnaglyph(frameLeftRect, frameRightRect));
title('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
```

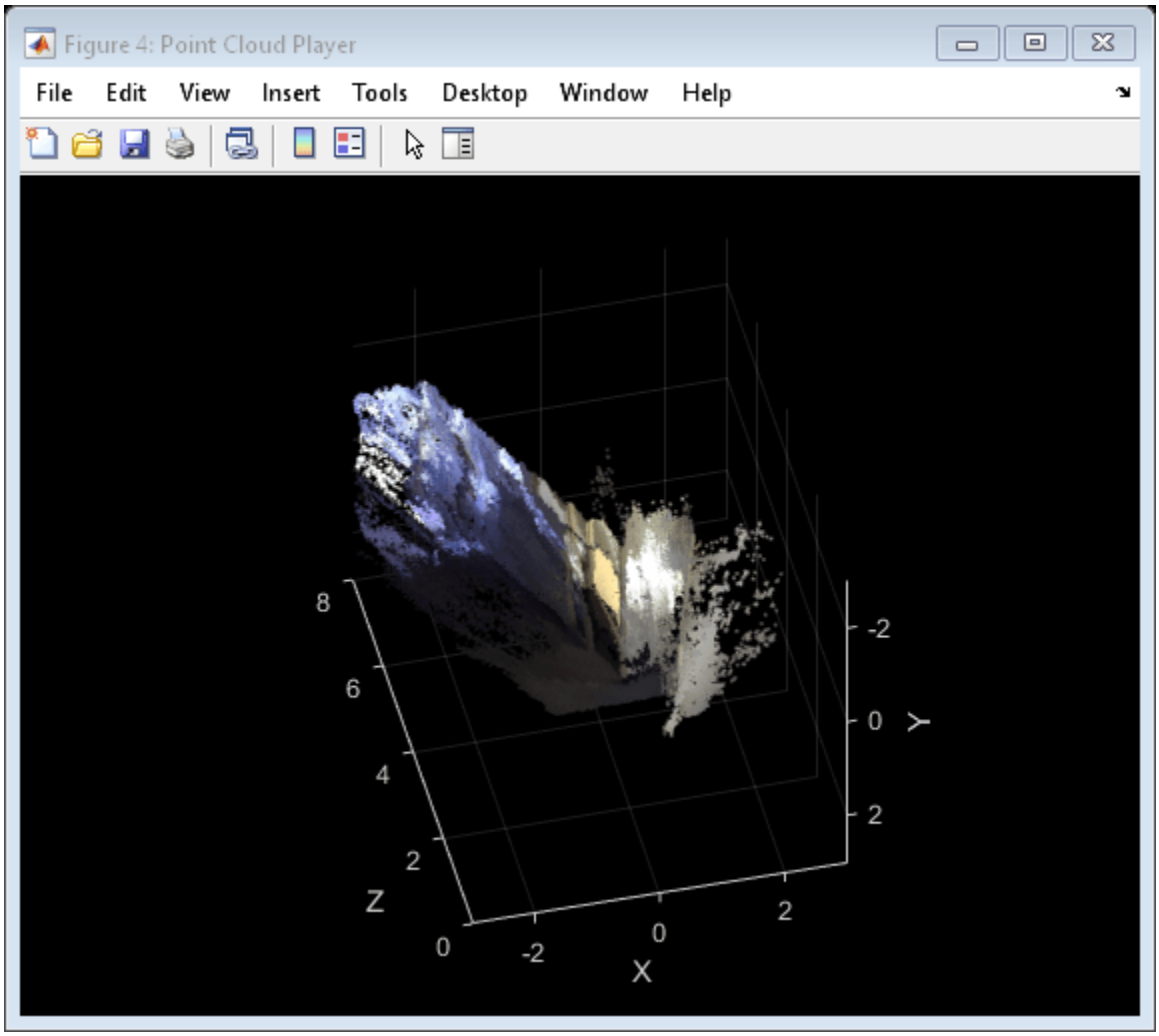


Reconstruct the 3-D Scene

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

```
points3D = reconstructScene(disparityMap, reprojectionMatrix);

% 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.

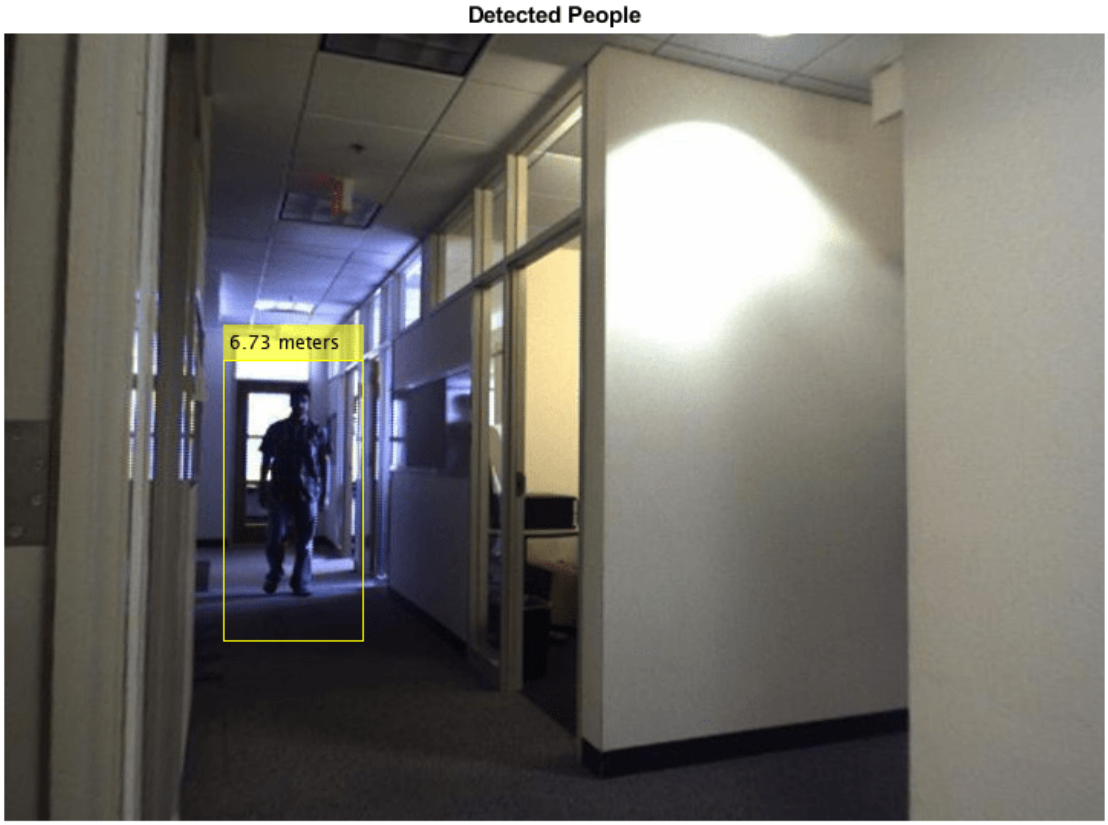
```
% Create the people detector object. Limit the minimum object size for
% speed.
peopleDetector = vision.PeopleDetector('MinSize', [166 83]);

% 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');
```

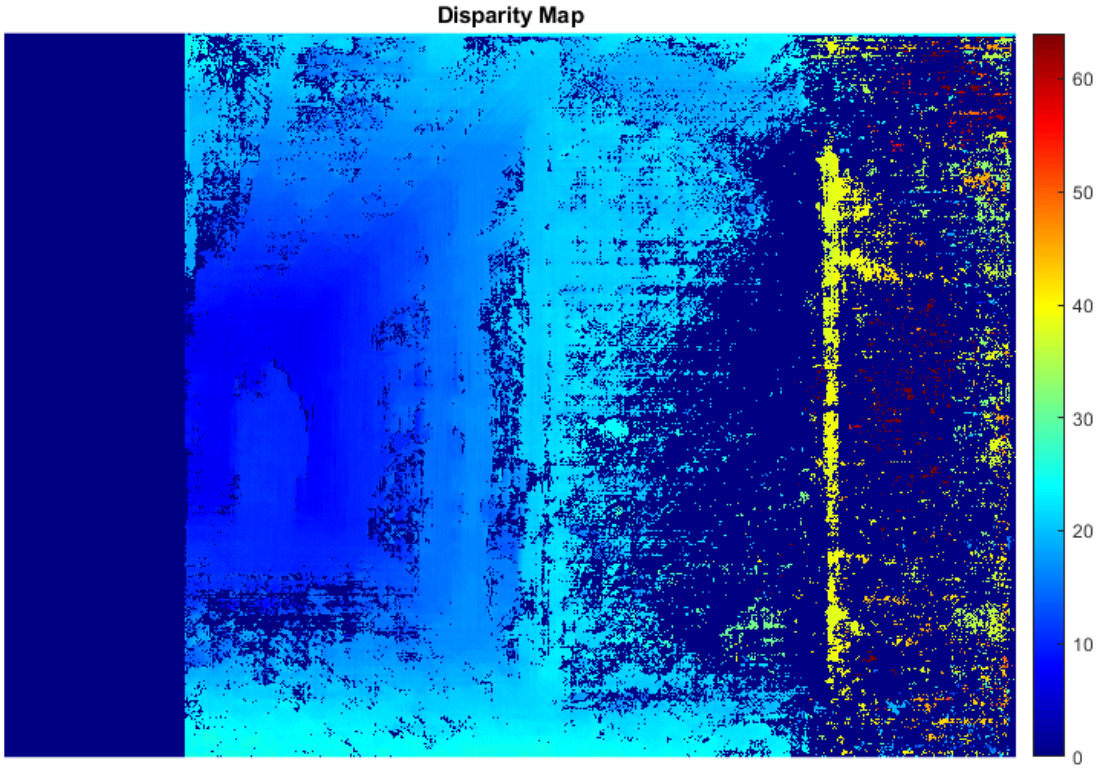


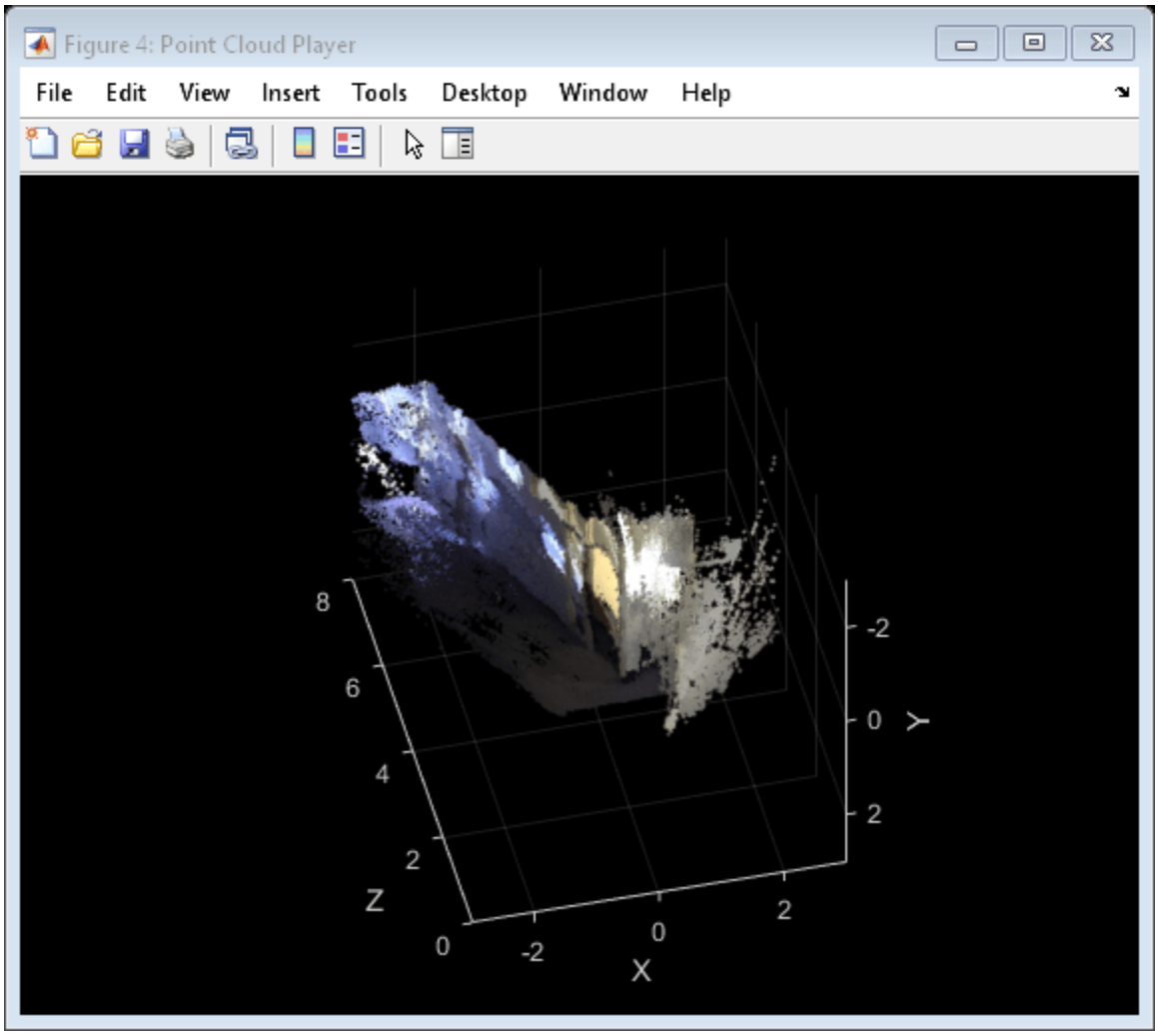
Process the Rest of the Video

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

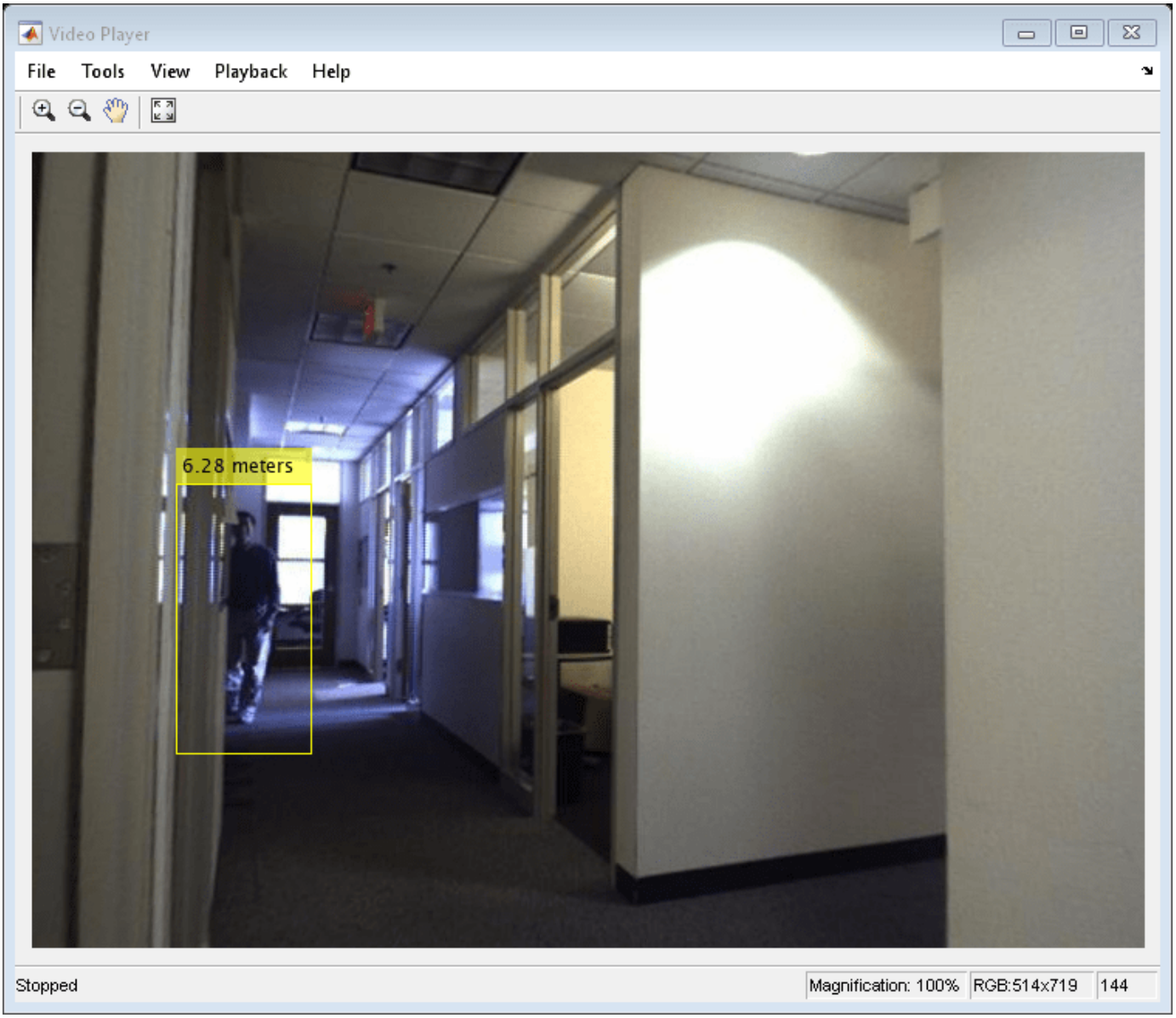
```
while hasFrame(readerLeft) && hasFrame(readerRight)
    % Read the frames.
    frameLeft = readFrame(readerLeft);
    frameRight = readFrame(readerRight);

    % 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, reprojectionMatrix);
    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
```


Disparity Map
60
50
40
30
20
10
0




```
% 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:

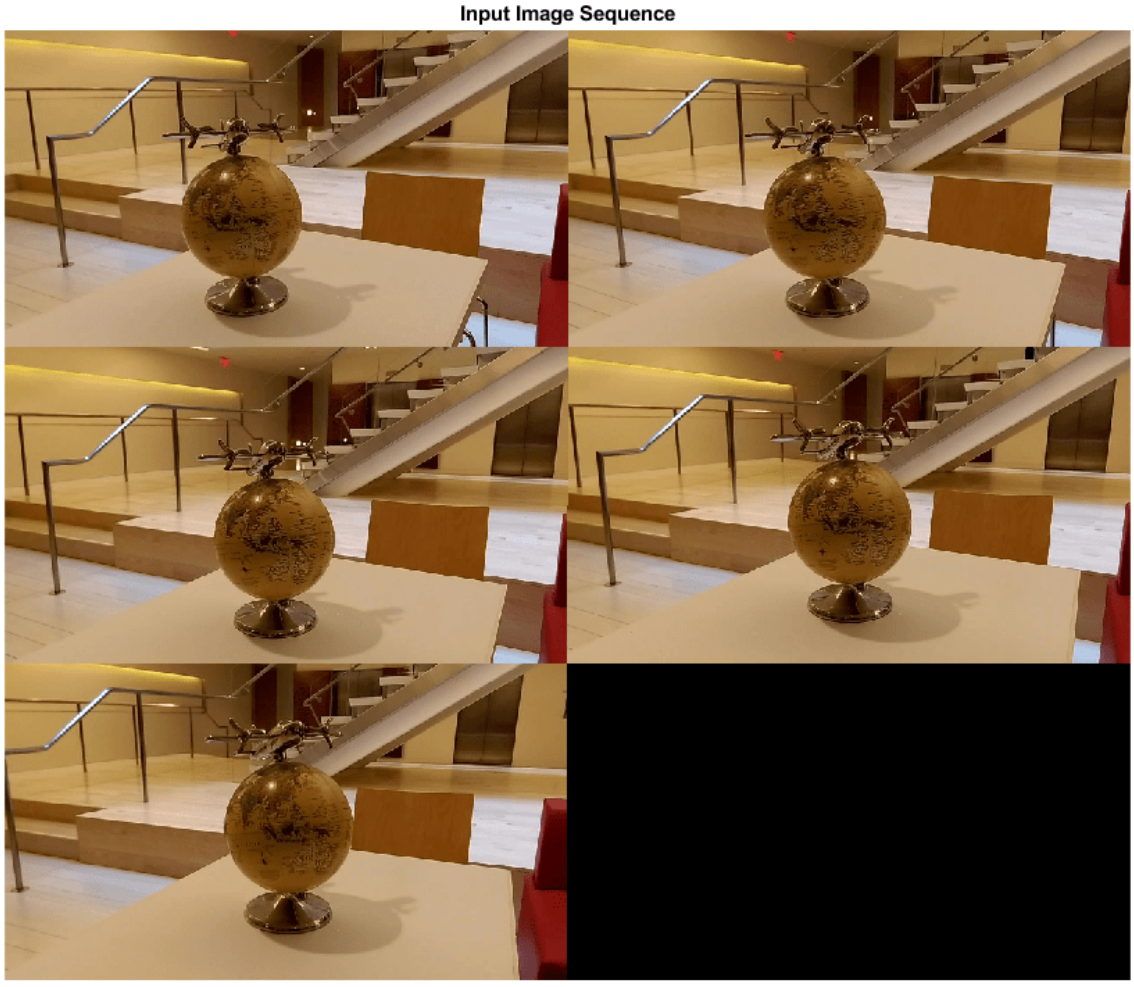
1. For each pair of consecutive images, find a set of point correspondences. This example detects the interest points using the `detectSURFFeatures` function, extracts the feature descriptors using the `extractFeatures` functions, and finds the matches using the `matchFeatures` function. Alternatively, 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} = im2gray(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.
```

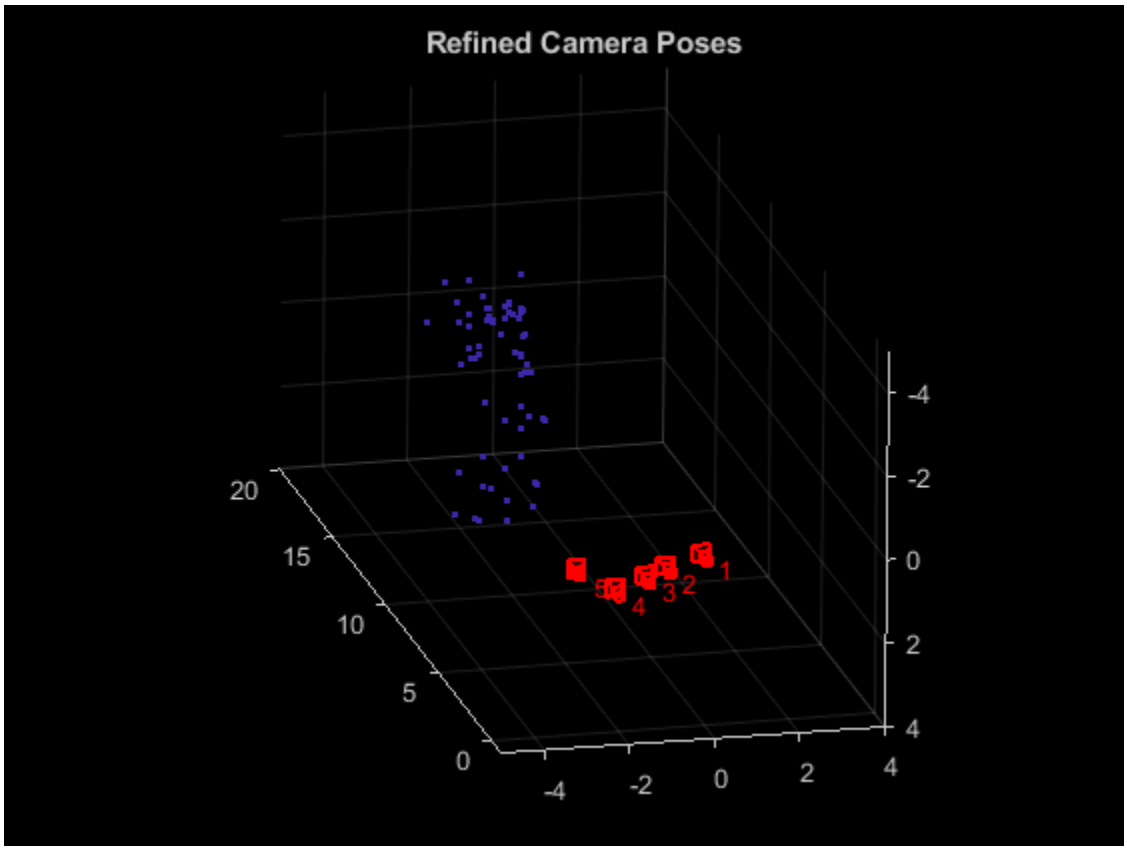
```
    currPoints   = detectSURFFeatures(I, 'NumOctaves', 8, 'ROI', roi);
    currFeatures = extractFeatures(I, currPoints, 'Upright', true);
    indexPairs   = matchFeatures(prevFeatures, currFeatures, ...
        'MaxRatio', .7, 'Unique',  true);

    % 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

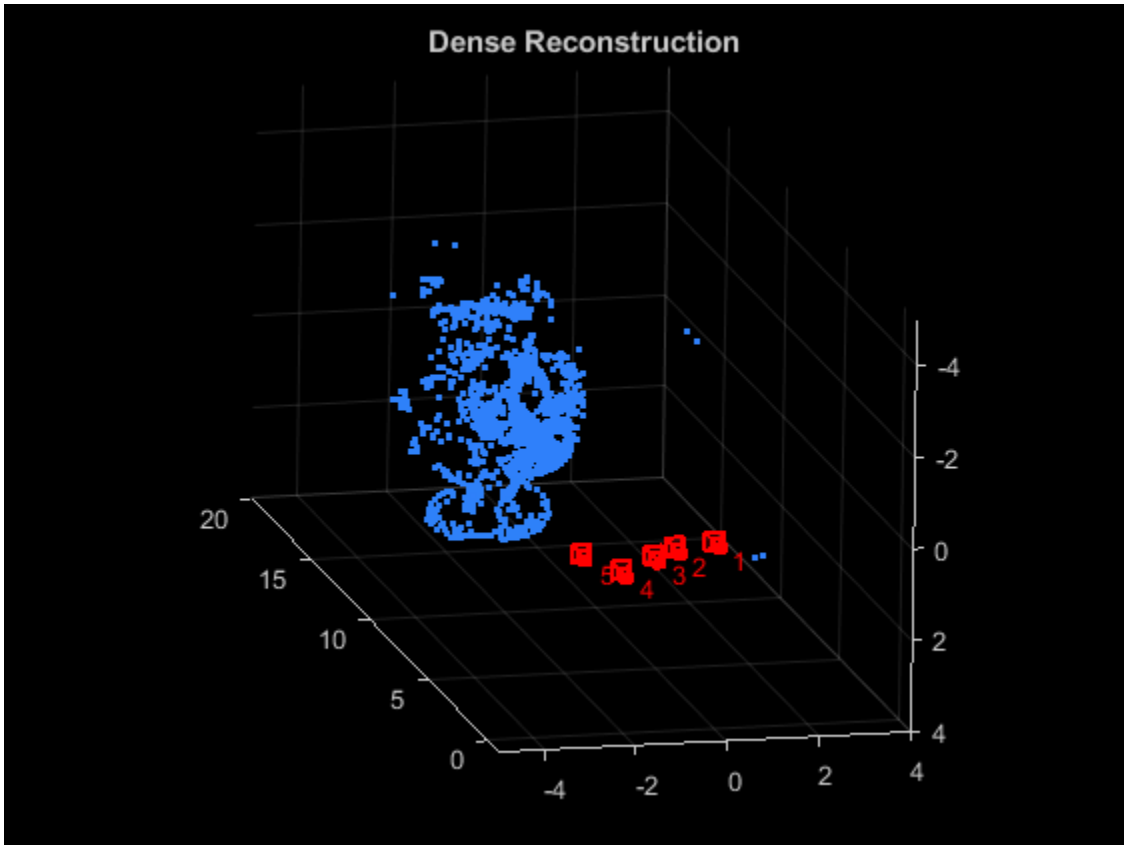
Go through the images again. This time detect a dense set of corners, and track them across all views using`vision.PointTracker`.

```
% 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.

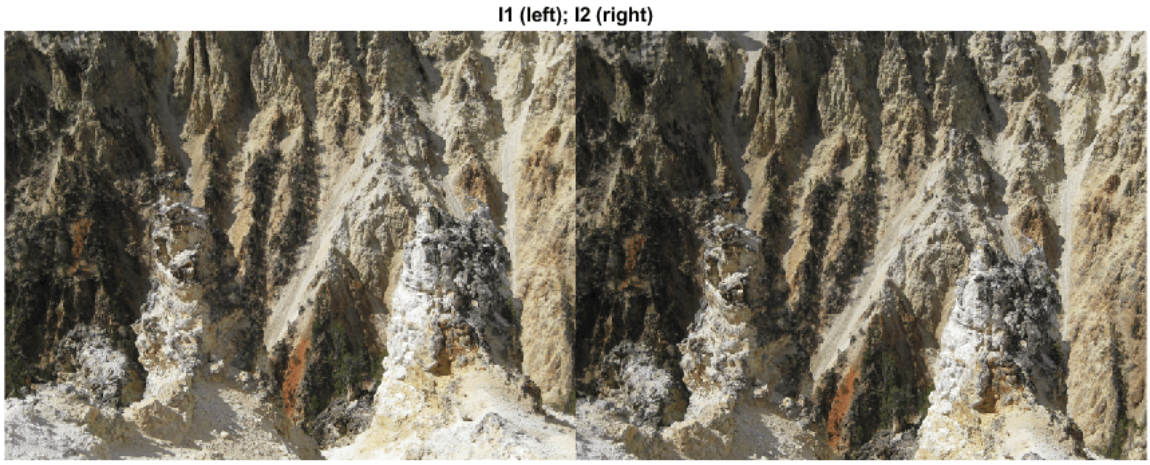
Step 1: Read Stereo Image Pair

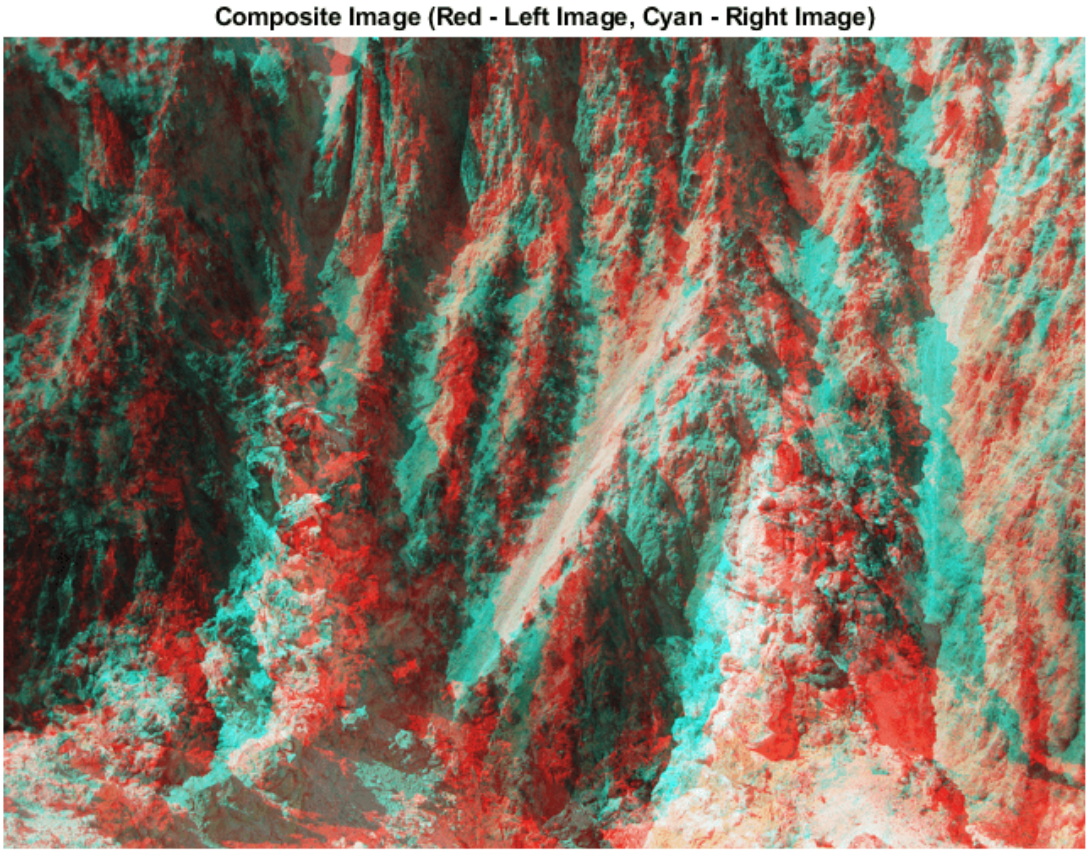
Read in two color images of the same scene, which were taken from different positions. Then, convert them to grayscale. Colors are not required for the matching process.

```
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)');
```





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

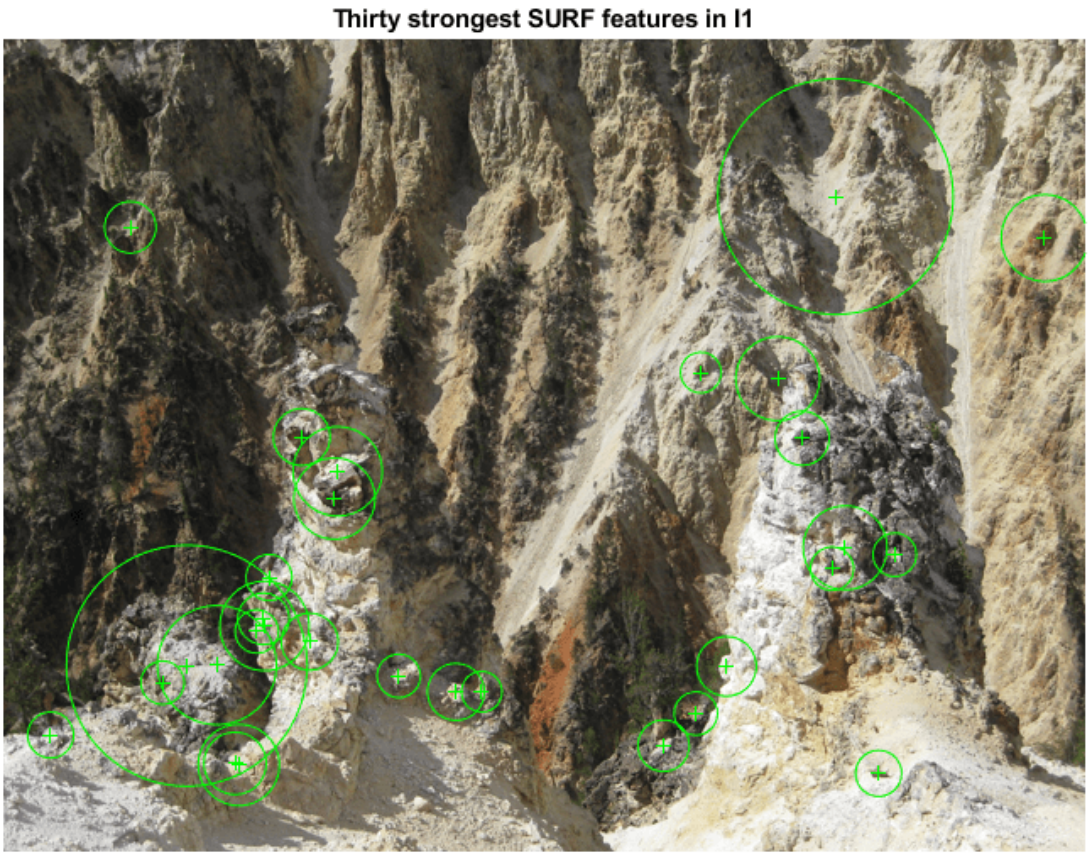
The rectification process requires a set of point correspondences between the two images. To generate these correspondences, you will collect points of interest from both images, and then choose potential matches between them. Use `detectSURFFeatures` to find blob-like features in both images.

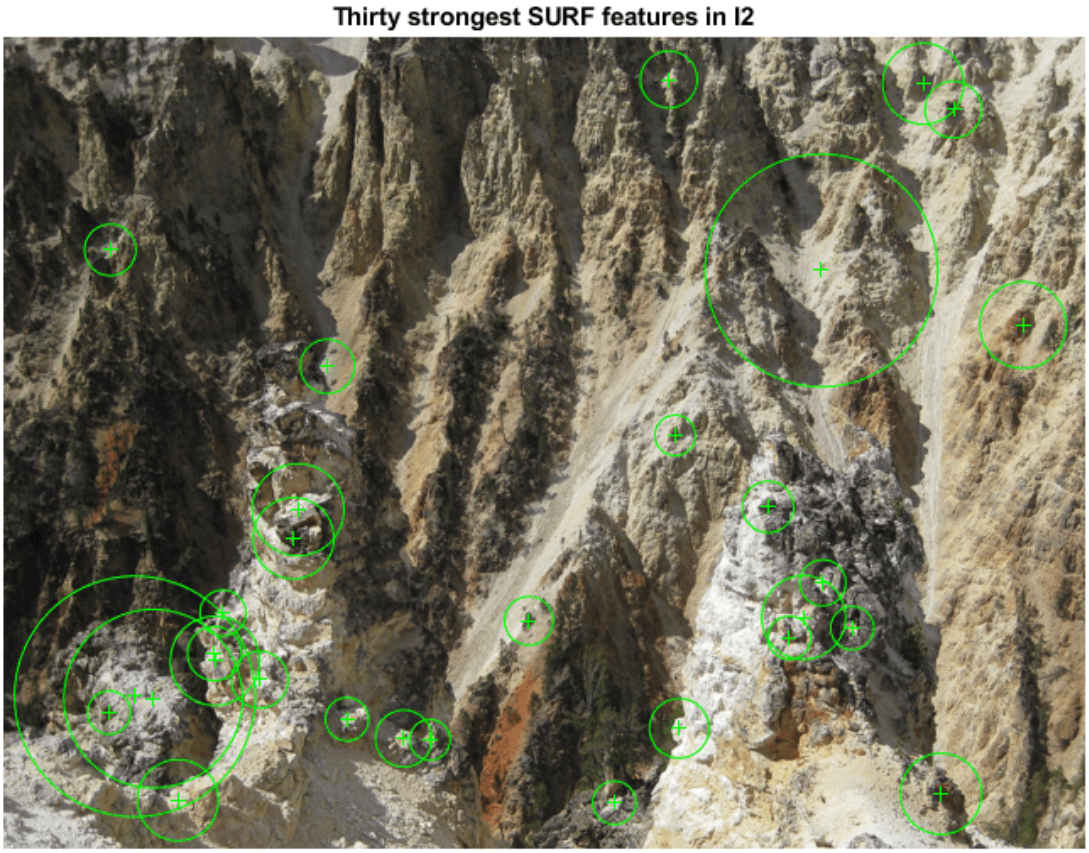
```
blobs1 = detectSURFFeatures(I1gray, 'MetricThreshold', 2000);
blobs2 = detectSURFFeatures(I2gray, 'MetricThreshold', 2000);
```

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');
```





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.

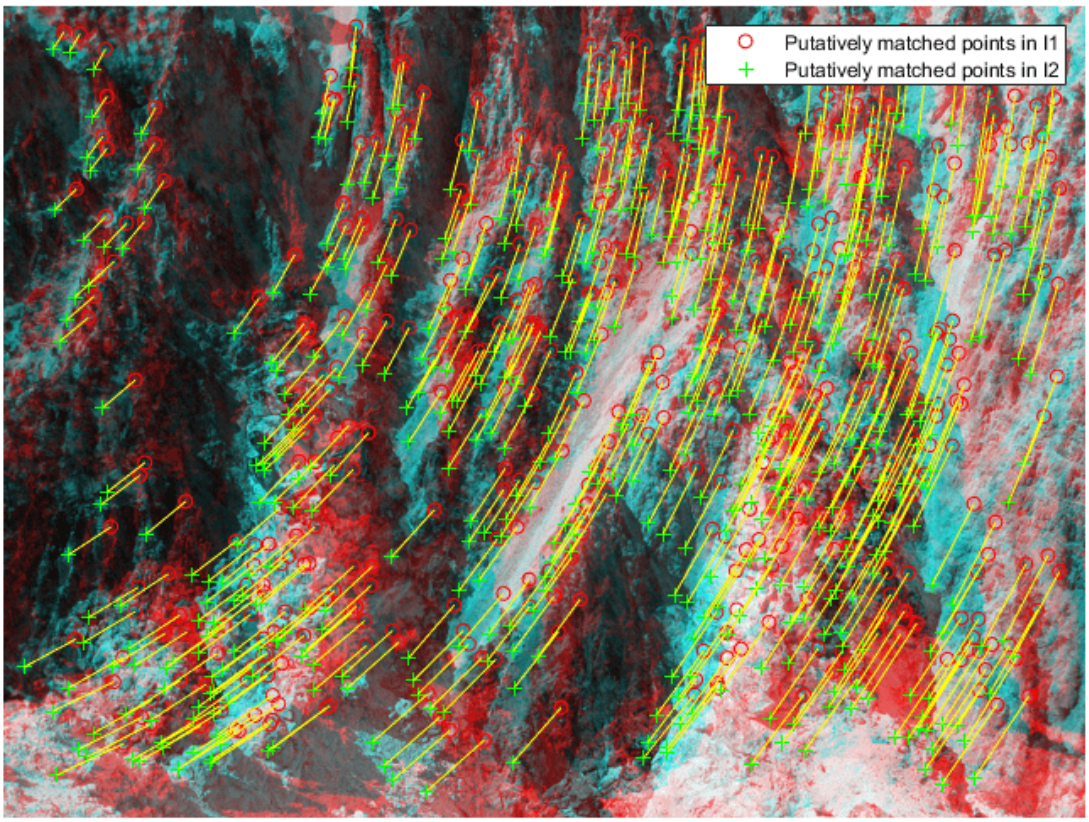
```
indexPairs = matchFeatures(features1, features2, 'Metric', 'SAD', ...
  'MatchThreshold', 5);
```

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');
```

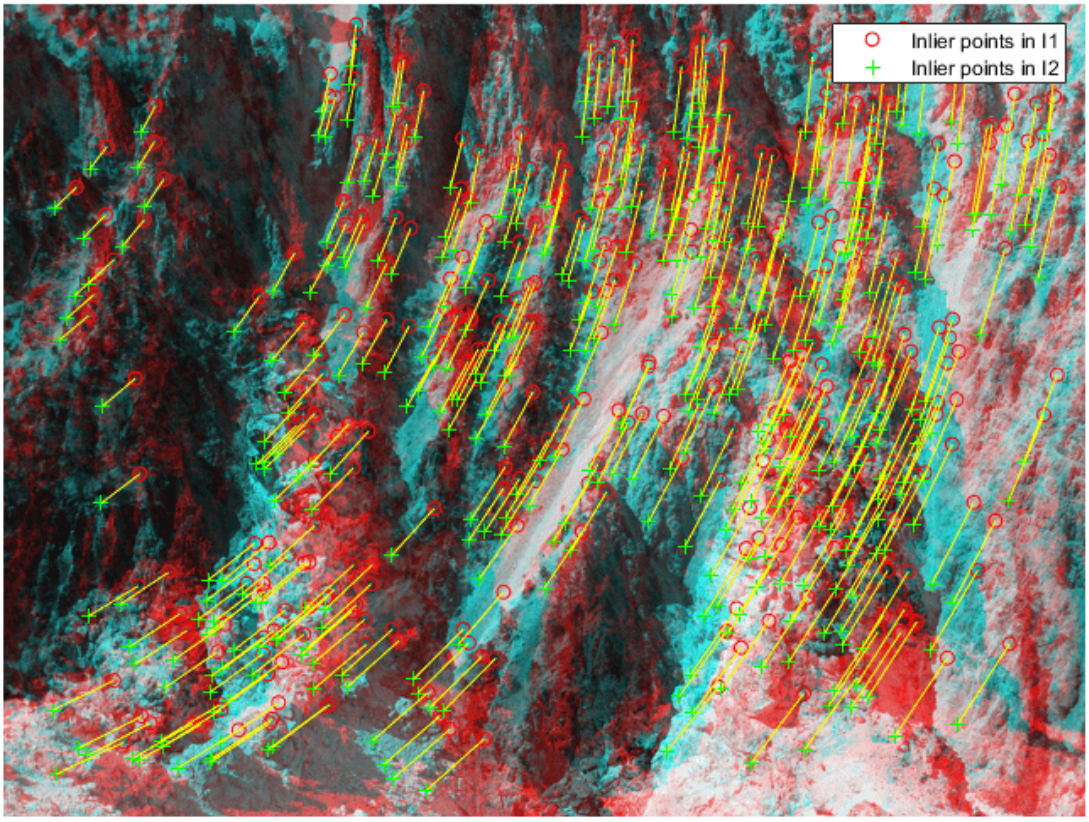


Step 4: Remove Outliers Using Epipolar Constraint

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');
```



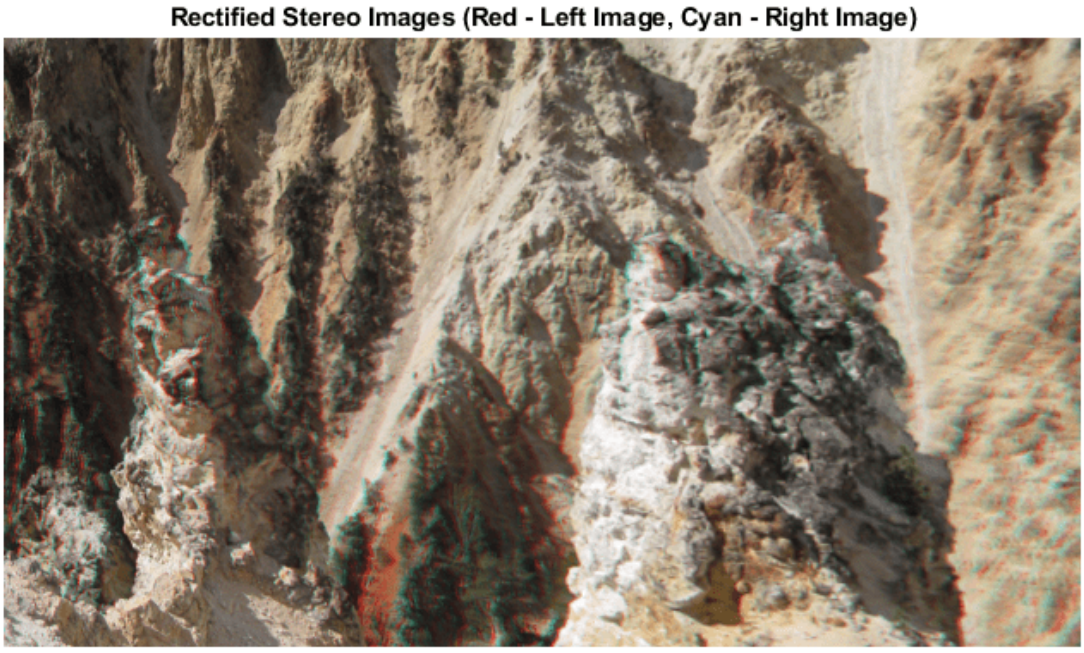
Step 5: Rectify Images

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)');
```



Step 6: Generalize The Rectification Process

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

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

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 2-9
- “Code Generation for Face Tracking with PackNGo” on page 2-16
- “Code Generation for Depth Estimation From Stereo Video” on page 2-24
- “Detect Face (Raspberry Pi2)” on page 2-29
- “Track Face (Raspberry Pi2)” on page 2-35
- “Video Display in a Custom User Interface” on page 2-41
- “Generate Code for Detecting Objects in Images by Using ACF Object Detector” on page 2-46

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 3-201. 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 DAG Network

This example uses the `ssdResNet50VehicleExample_20a` MAT-file containing the pretrained SSD network. This file is approximately 44 MB size. Download the file from the MathWorks website.

```
ssdNetFile = matlab.internal.examples.downloadSupportFile('vision/data','ssdResNet50VehicleExamp
```

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,matFile)

%   Copyright 2019-2022 The MathWorks, Inc.

persistent ssdObj;

if isempty(ssdObj)
    ssdObj = coder.loadDeepLearningNetwork(matFile);
end

% Pass in input
[bboxes,~,labels] = detect(ssdObj,in,'Threshold',0.5);

% Convert categorical labels to cell array of charactor 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-by-300-by-3. This value corresponds to the input layer size of SSD Network.

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

```
Code generation successful: View report
```

Run Generated MEX

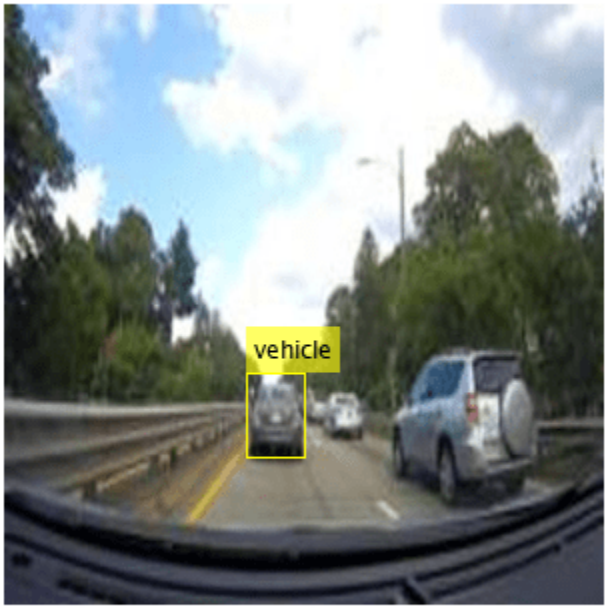
To test the generated MEX, the example uses a small vehicle data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission.

Load the vehicle data set and randomly select 10 images to test the generated code.

```
unzip vehicleDatasetImages.zip
imageNames = dir(fullfile(pwd,'vehicleImages','*.jpg'));
imageNames = {imageNames.name}';
rng(0);
imageIndices = randi(length(imageNames),1,10);
```

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

```
for idx = 1:10
    testImage = imread(fullfile(pwd,'vehicleImages',imageNames{imageIndices(idx)}));
    resizedImage = imresize(testImage,[300,300]);
    detectorOutput = ssdObj_detect_mex(resizedImage,ssdNetFile);
    imshow(detectorOutput);
    pause(0.5)
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 3-338. 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 3-306.

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

This example uses the `yolov2ResNet50VehicleExample` MAT-file containing the pretrained network. The file is approximately 98MB in size. Download the file from the MathWorks website.

```
matFile = matlab.internal.examples.downloadSupportFile('vision/data','yolov2ResNet50VehicleExamp
vehicleDetector = load(matFile);
net = vehicleDetector.detector.Network
```

```
net =
  DAGNetwork with properties:

         Layers: [150×1 nnet.cnn.layer.Layer]
    Connections: [162×2 table]
     InputNames: {'input_1'}
```

```
    OutputNames: {'yolov2OutputLayer'}
```

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,matFile)

%   Copyright 2018-2021 The MathWorks, Inc.

persistent yolov2Obj;

if isempty(yolov2Obj)
    yolov2Obj = coder.loadDeepLearningNetwork(matFile);
end

% Call to detect method
[bboxes,~,labels] = yolov2Obj.detect(in,'Threshold',0.5);

% Convert categorical labels to cell array of charactor vectors
labels = cellstr(labels);

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

Run MEX Code Generation

To generate CUDA code for the 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-by-224-by-3. This value corresponds to the input layer size of YOLOv2.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.GenerateReport = true;
inputArgs = {ones(224,224,3,'uint8'),coder.Constant(matFile)};

codegen -config cfg yolov2_detect -args inputArgs
```

```
Code generation successful: View report
```

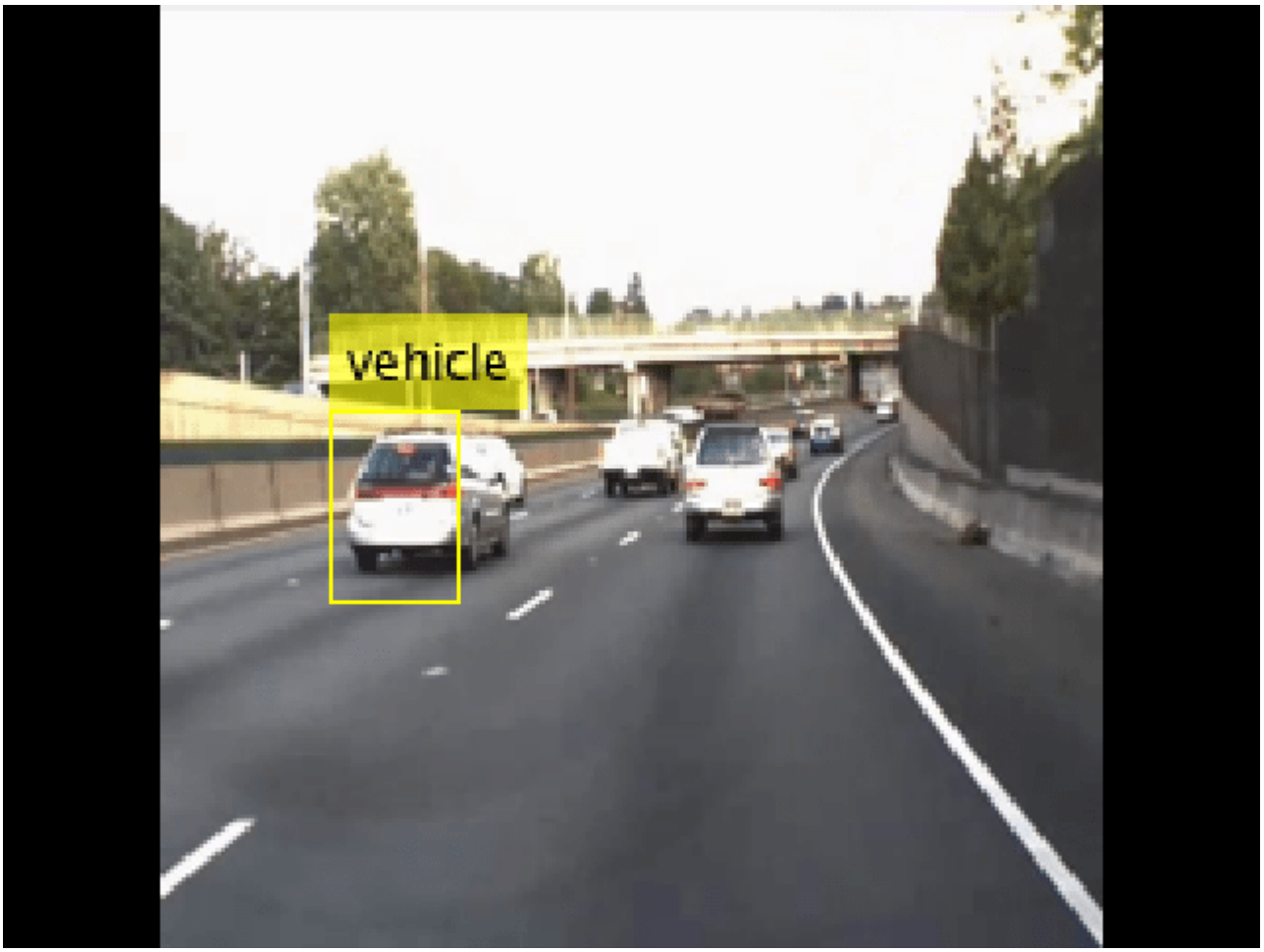
Run Generated MEX

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,matFile);
    step(depVideoPlayer, out);
    % Exit the loop if the video player figure window is closed
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer);
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 4-29 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 `visionRecovertformCodeGeneration_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.

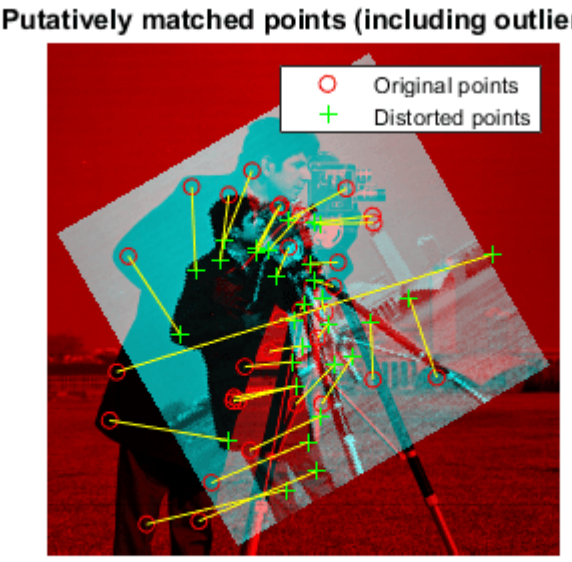
The original example uses showMatchedFeatures and imshowpair routines for visualization of the results. These routines are extracted to a new function `featureMatchingVisualization_extrinsic`. This function is declared extrinsic.

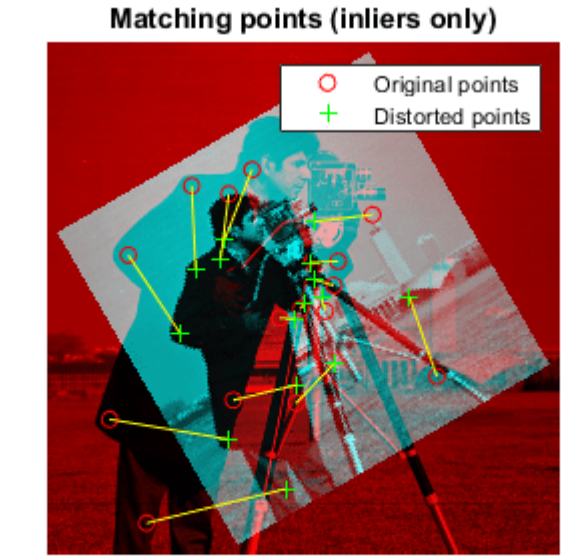
Run the Simulation

The kernel file `visionRecovertformCodeGeneration_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] = ...
    visionRecovertformCodeGeneration_kernel(original, distorted);

scaleRecovered = 0.701003
```

```
thetaRecovered = 30.235065
```





Compile the MATLAB Function Into a MEX File

Now use the `codegen` (MATLAB Coder) function to compile the visionRecovertformCodeGeneration_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 % `visionRecovertformCodeGeneration_kernel` are a pair of images, we define both the inputs with the following properties:

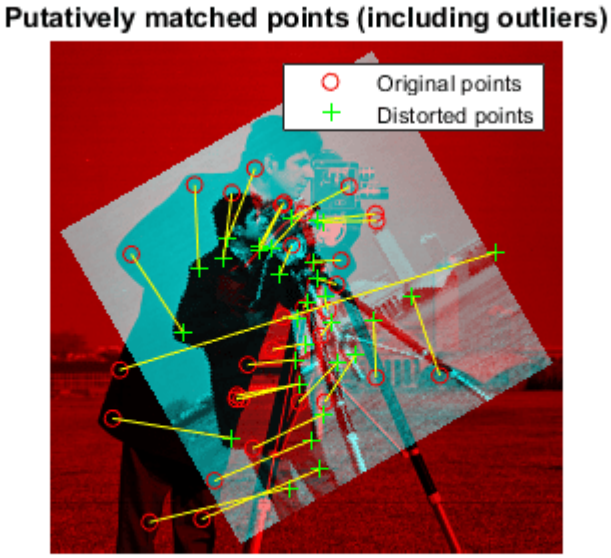
- variable-sized at run-time with upper-bound [1000 1000]
- data type uint8

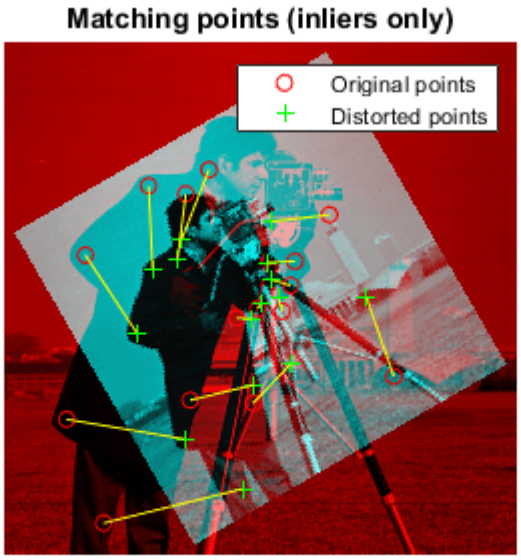
```
% Define the properties of input images
imageTypeAndSize = coder.typeof(uint8(0), [1000 1000],[true true]);
compileTimeInputs  = {imageTypeAndSize, imageTypeAndSize};

codegen visionRecovertformCodeGeneration_kernel.m -report -args compileTimeInputs;

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

Run the Generated Code

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

```
scaleRecovered = 0.701003

thetaRecovered = 30.235065
```





Clean Up

```
clear visionRecovertformCodeGeneration_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” on page 8-20 example with packNGo function. The `packNGo` (MATLAB Coder) 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 “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4.

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” on page 8-20. 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” on page 2-9.

```
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) ....
Warning: C Compiler produced warnings. See the build log for further details.

Code generation successful (with warnings): To view the report, open('codegen\exe\FaceTrackingKL
```

Note that, instead of using codegen command, you can open a dialog and launch a code generation project using `codegen` (MATLAB 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', unzipFolderL
  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(binfo.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

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
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',s

            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(regexp(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

    % 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
```

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 1-131 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 2-9 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.

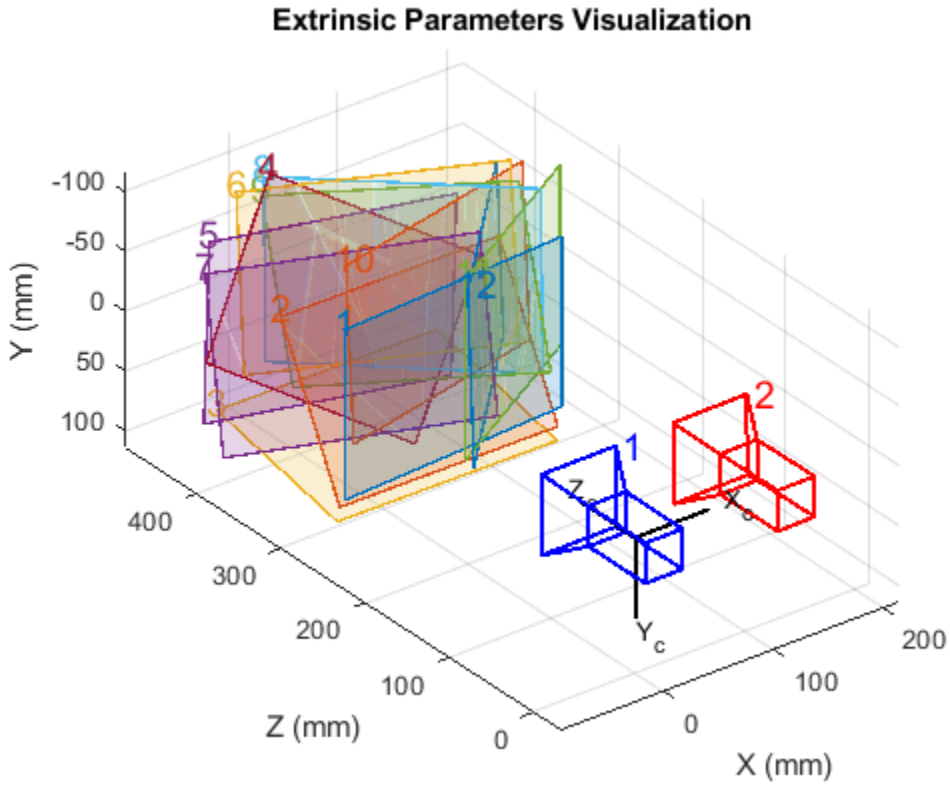
This presents a problem for generating code from MATLAB code, which uses `cameraParameters` or `stereoParameters` objects, which are typically created in advance during camera calibration. To solve this problem, use the `toStruct()` method to convert the `cameraParameters` or the `stereoParameters` object 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 1-131 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

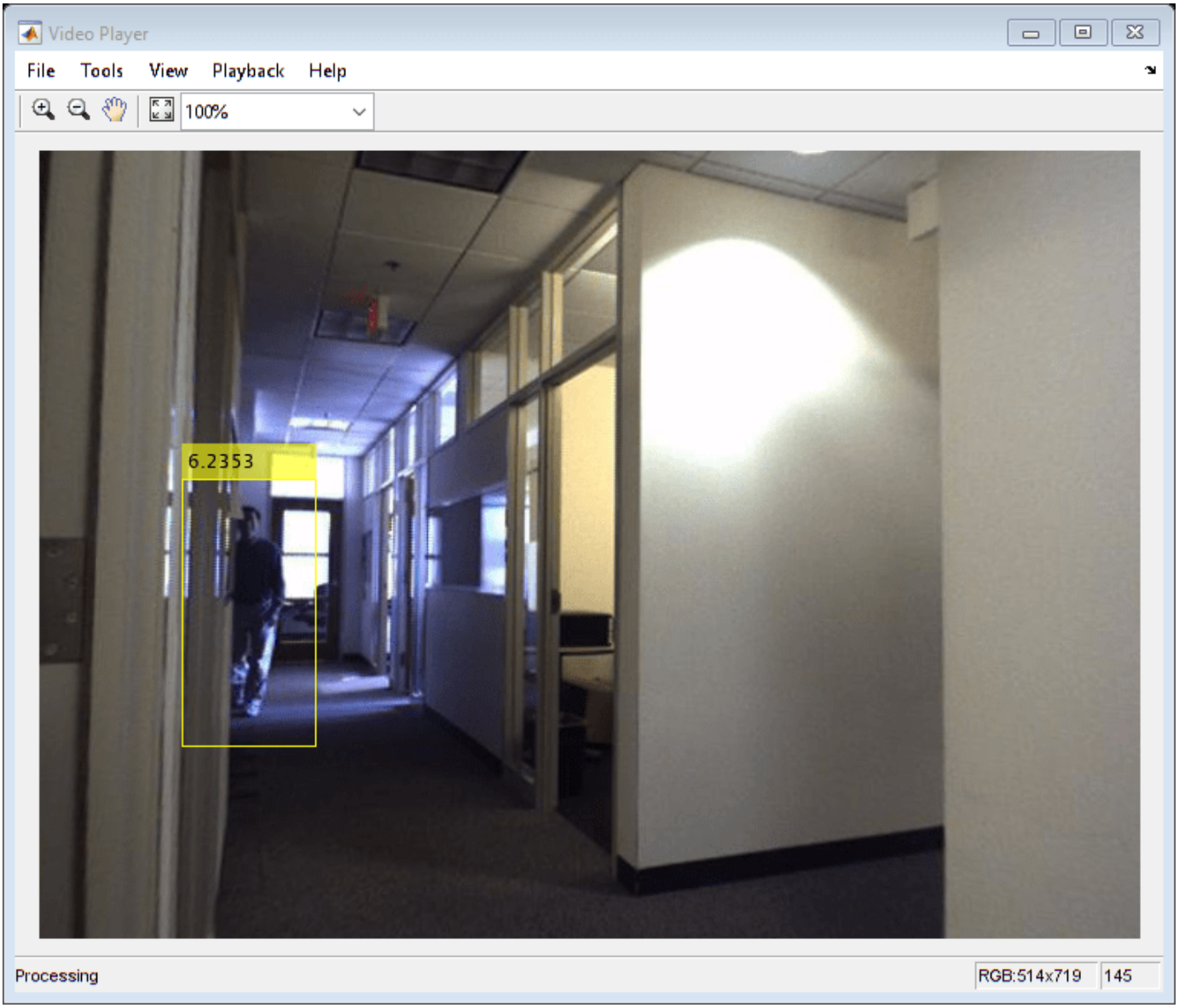
Use the codegen function to compile the `depthEstimationFromStereoVideo_kernel` function into a MEX-file. You can specify the '-report' option to generate a compilation report that shows the original MATLAB code and the associated files that were created during C code generation. You may want to create 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;
```

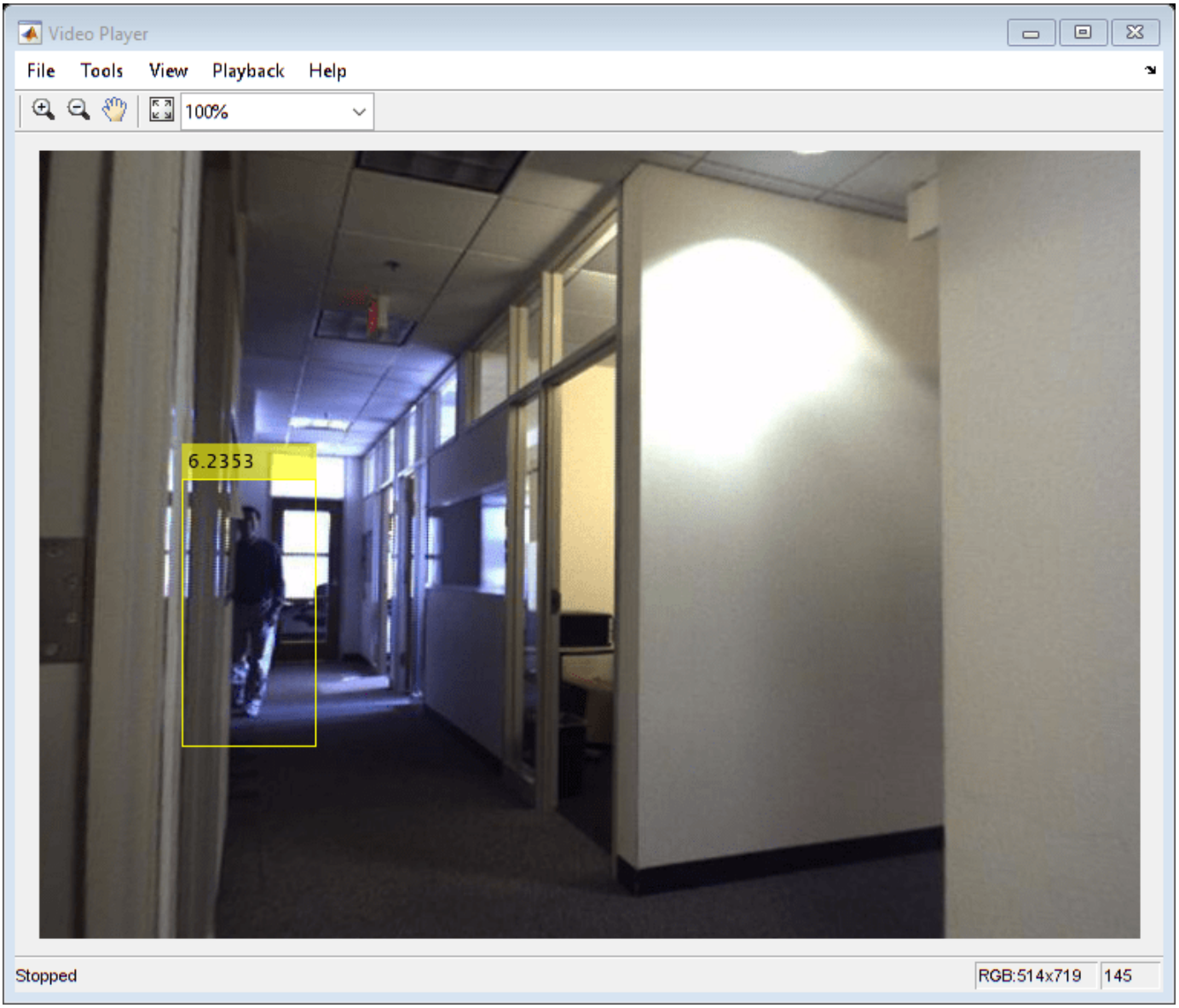
Run the Generated Code

```
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” on page 2-9.

```
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 highqui 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) ....
Code generation successful.
```

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();
```

```
Warning: Directory already exists.
```

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 faceDetectionARMMakefi
```

```
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
```

```
                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
```

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” on page 2-9.

```
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) ....
Code generation successful.
```

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();
```

```
Warning: Directory already exists.
```

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 faceTrackingARMMakefi
```

```
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
```

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.

This example is written as a function with the main body at the top. The example also uses nested functions and a separate helper function listed.

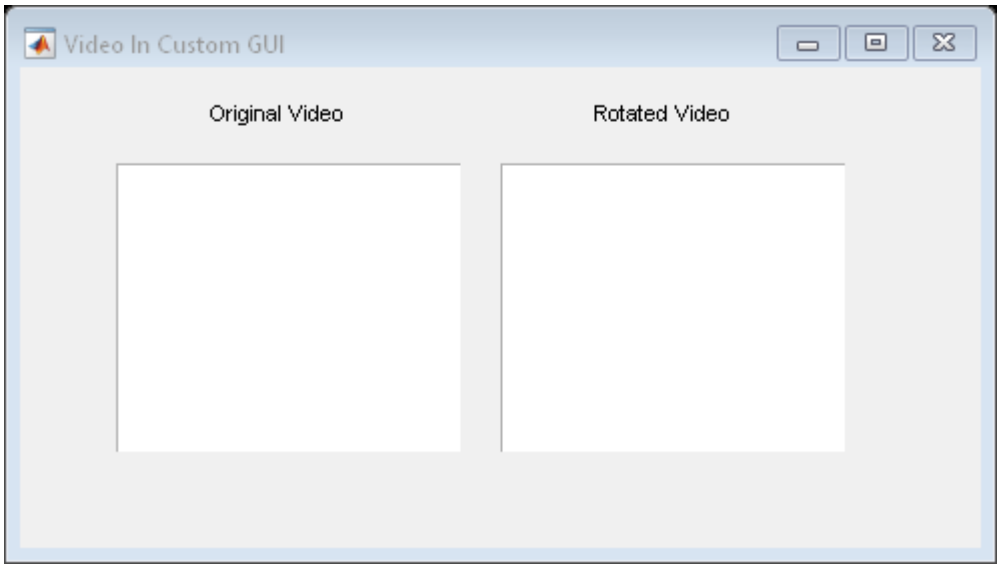
```
function VideoInCustomGUIExample()
```

Initialize the video reader.

```
videoSrc = vision.VideoFileReader('vipmen.avi', 'ImageColorSpace', 'Intensity');
```

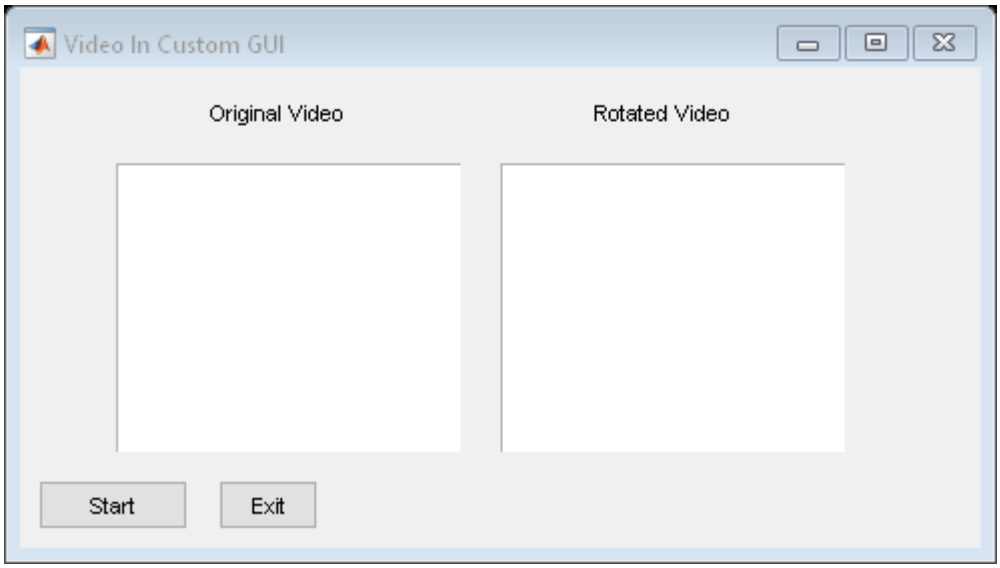
Create a figure window and two axes to display the input video and the processed video.

```
[hFig, hAxes] = createFigureAndAxes();
```



Add buttons to control video playback.

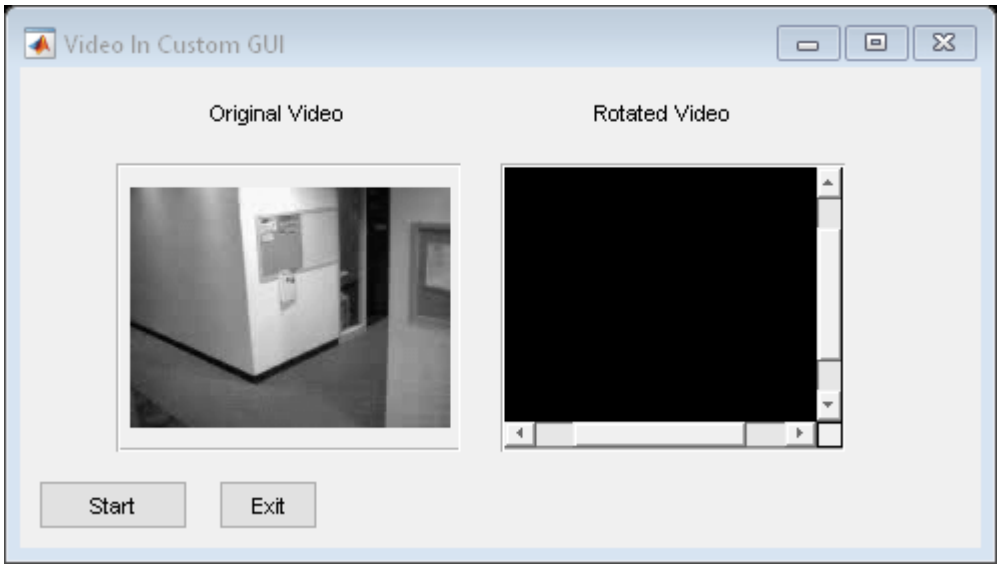
```
insertButtons(hFig, hAxes, videoSrc);
```



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','PBButton123','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 the input video frame and displays the original input and rotated frame on two different axes. The helper function `showFrameOnAxis`, is responsible for displaying a frame of the video on the user-defined axis. This helper 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
```

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
```

Generate Code for Detecting Objects in Images by Using ACF Object Detector

This example shows how to generate code from a MATLAB function that detects objects in images by using an `acfObjectDetector` object. When you intend to generate code from your MATLAB function that uses an `acfObjectDetector` object, you must create the object outside of the MATLAB function. The example explains how to modify the MATLAB code in Train a Stop Sign Detector Using an ACF Object Detector to support code generation.

Design the MATLAB Code File for Code Generation

To generate C Code, MATLAB Coder requires MATLAB code to be in the form of a function. The arguments of the function cannot be MATLAB objects. This requirement presents a problem for generating code from the MATLAB function that uses `acfObjectDetector` objects created outside of the MATLAB function. To solve this problem, use the `toStruct` function to convert the `acfObjectDetector` object into a structure and pass the structure to the MATLAB function.

To support code generation, this example restructures the code of an existing example (See Train a Stop Sign Detector Using an ACF Object Detector) in a function called `detectObjectsUsingACF`, which is present in the current working folder as a supporting file. The `detectObjectsUsingACF` function takes an image as an input and loads the pretrained ACF stop sign detector.

```
type('detectObjectsUsingACF.m')
```

```
function [bboxes,scores] = detectObjectsUsingACF(InputImage)
% Load a trained detector from a MAT file
S = coder.load('detectorStruct.mat');
% Define a persistent variable
persistent detector
if isempty(detector)
% Re-create the ACF Object Detector
detector = acfObjectDetector(S.detectorStruct.Classifier,S.detectorStruct.TrainingOptions);
end
% Use the detect function to detect objects in the input image
[bboxes,scores] = detect(detector,InputImage);
end
```

Create ACF Stop Sign Detector Outside of the MATLAB Function

Load the training data.

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

Select the ground truth for stop signs. The ground truth data is the set of known locations of stop signs in the images.

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

Add the full path to the image files.

```
stopSigns.imageFilename = fullfile(toolboxdir('vision'),...
    'visiondata',stopSigns.imageFilename);
```

Use the `trainACFObjectDetector` function to train the ACF detector. Turn off the training progress output by setting `'Verbose' to false`.

```
detector = trainACFObjectDetector(stopSigns,'NegativeSamplesFactor',2,'Verbose',false);
```

Generate C-MEX Function

Because you intend to generate code for the MATLAB function `detectObjectsUsingACF`, convert the created `detector` into a structure.

```
detectorStruct = toStruct(detector);
```

Save the trained object structure as a MAT file.

```
save('detectorStruct.mat','detectorStruct');
```

Generate C-MEX code that you can run in the MATLAB environment. Use the `codegen` (MATLAB Coder) command.

```
codegen detectObjectsUsingACF -report -args { coder.typeof(uint8(0), [inf inf 3])}
```

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

Detect Objects Using Generated C-MEX Function

To detect objects in an image, load a test image.

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

Call the generated C-MEX function by passing the loaded image `img` as an input.

```
[bboxes, scores] = detectObjectsUsingACF_mex(img);
```

Display the detection results and insert the bounding boxes for objects into the image.

```
img = insertObjectAnnotation(img,'rectangle',bboxes,scores);
figure
imshow(img)
```

Clean Up

Release the system memory used to store the generated C-MEX file.

```
clear ObjectDetectionFromImages_mex;
```

See Also

“Introduction to Code Generation with Feature Matching and Registration” on page 2-9 | “Generate Code to Detect Edges on Images” (MATLAB Coder)

Deep Learning, Semantic Segmentation, and Detection Examples

- “Multiclass Object Detection Using YOLO v2 Deep Learning” on page 3-3
- “Generate Adversarial Examples for Semantic Segmentation” on page 3-13
- “Classify Defects on Wafer Maps Using Deep Learning” on page 3-24
- “Detect Image Anomalies Using Explainable One-Class Classification Neural Network” on page 3-40
- “Detect Image Anomalies Using Pretrained ResNet-18 Feature Embeddings” on page 3-59
- “Train Object Detectors in Experiment Manager” on page 3-79
- “Activity Recognition Using R(2+1)D Video Classification” on page 3-86
- “Activity Recognition from Video and Optical Flow Data Using Deep Learning” on page 3-108
- “Evaluate a Video Classifier” on page 3-136
- “Extract Training Data for Video Classification” on page 3-140
- “Classify Streaming Webcam Video Using SlowFast Video Classifier” on page 3-144
- “Gesture Recognition using Videos and Deep Learning” on page 3-147
- “Explore Semantic Segmentation Network Using Grad-CAM” on page 3-168
- “Point Cloud Classification Using PointNet Deep Learning” on page 3-178
- “Object Detection Using SSD Deep Learning” on page 3-201
- “Object Detection in a Cluttered Scene Using Point Feature Matching” on page 3-211
- “Semantic Segmentation Using Deep Learning” on page 3-222
- “Calculate Segmentation Metrics in Block-Based Workflow” on page 3-239
- “Semantic Segmentation of Multispectral Images Using Deep Learning” on page 3-244
- “3-D Brain Tumor Segmentation Using Deep Learning” on page 3-262
- “Image Category Classification Using Bag of Features” on page 3-275
- “Image Category Classification Using Deep Learning” on page 3-282
- “Image Retrieval Using Customized Bag of Features” on page 3-291
- “Create SSD Object Detection Network” on page 3-298
- “Train YOLO v2 Network for Vehicle Detection” on page 3-301
- “Import Pretrained ONNX YOLO v2 Object Detector” on page 3-306
- “Export YOLO v2 Object Detector to ONNX” on page 3-313
- “Estimate Anchor Boxes From Training Data” on page 3-319
- “Object Detection Using YOLO v3 Deep Learning” on page 3-323
- “Object Detection Using YOLO v2 Deep Learning” on page 3-338
- “Create YOLO v2 Object Detection Network” on page 3-348
- “Train Object Detector Using R-CNN Deep Learning” on page 3-351

- “Object Detection Using Faster R-CNN Deep Learning” on page 3-364
- “Train Classification Network to Classify Object in 3-D Point Cloud” on page 3-374
- “Estimate Body Pose Using Deep Learning” on page 3-384
- “Generate Image from Segmentation Map Using Deep Learning” on page 3-392
- “Train Simple Semantic Segmentation Network in Deep Network Designer” on page 3-407
- “Train ACF-Based Stop Sign Detector” on page 3-412
- “Train Fast R-CNN Stop Sign Detector” on page 3-415
- “Perform Instance Segmentation Using Mask R-CNN” on page 3-418
- “Object Detection Using YOLO v4 Deep Learning” on page 3-423

Multiclass Object Detection Using YOLO v2 Deep Learning

This example shows how to train a multiclass object detector.

Overview

Deep learning is a powerful machine learning technique that you can use to train robust multiclass object detectors such as YOLO v2, YOLO v4, SSD, and Faster R-CNN. This example trains a YOLO v2 multiclass indoor object detector using the `trainYOLOv2ObjectDetector` function. The trained object detector is able to detect and identify multiple different indoor objects. For more information regarding training other multiclass object detectors such as YOLO v4, SSD, or Faster R-CNN, see, “Getting Started with Object Detection Using Deep Learning” on page 17-24.

Perform Object Detection using Pretrained Detector

Download the pretrained YOLO v2 object detetctor.

```
if ~exist('yolov2IndoorObjectDetector.mat','file')
    disp('Downloading pretrained detector (98 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/yolov2IndoorObjectDetecto
    websave('yolov2IndoorObjectDetector.zip',pretrainedURL);
    unzip('yolov2IndoorObjectDetector.zip')
end
```

```
Downloading pretrained detector (98 MB)...
```

Load the object detector.

```
pretrained = load('yolov2IndoorObjectDetector.mat');
detector = pretrained.detector;
```

Read a test image that contains objects of the target classes and resize it to the input size of the network.

```
inputSize = [450 450];
imTest = imread('indoorTest.jpg');
imTest = imresize(imTest,inputSize);
```

Detect objects and their labels in the image using the `detect` function.

```
[bbox, score, label]  = detect(detector,imTest);
```

Visualize the predictions by overlaying the detected bounding boxes on the image using the `insertObjectAnnotation` function.

```
I = insertObjectAnnotation(imTest,'rectangle',bbox,label);
imshow(I)
```

Load Dataset

This example uses the Indoor Object Detection dataset created by Bishwo Adhikari [1]. The dataset consists of 2213 labeled images collected from indoor scenes containing 7 classes - fireextinguisher, chair, clock, trashbin, screen, and printer. Each image contains one or more labeled instances of the categories mentioned.

Download the dataset.

```
dsURL = 'https://zenodo.org/record/2654485/files/Indoor%20Object%20Detection%20Dataset.zip?downlo

outputFolder = fullfile(tempdir,'indoorObjectDetection');
imagesZip = fullfile(outputFolder,'indoor.zip');

if ~exist(imagesZip,'file')
    mkdir(outputFolder)
    disp('Downloading 401 MB Indoor Objects dataset images...');
    websave(imagesZip, dsURL);
    unzip(imagesZip, fullfile(outputFolder));
```

```
end

datapath = fullfile(outputFolder,'Indoor Object Detection Dataset');
```

The images are organized into 6 folders of different sequences. Create an `imageDatastore` by specifying the different folder paths.

```
numSequences = 6;
imds = imageDatastore(datapath,'IncludeSubfolders',true,'FileExtensions','.jpg');
```

Annotations and dataset split have been provided in the file `annotationsIndoor.mat`. Load the annotations and the indices corresponding to the training, validation, and test splits. Note that the split contains 2207 images in total instead of 2213 images as 6 images have no labels associated with them. Store the indices of images containing labels in `cleanIdx`.

```
data = load('annotationsIndoor.mat');
bbStore = data.BBstore;
trainingIdx = data.trainingIdx;
validationIdx = data.validationIdx;
testIdx = data.testIdx;
cleanIdx = data.idxs;
```

Finally, combine the `imageDatastore` and the `boxLabelDatastore`. Split the combined datastore into train, validation and test datastores by using the `subset` command and specifying the preloaded indices.

```
ds = combine(imds,bbStore);
% Remove the 6 images with no labels.
ds = subset(ds,cleanIdx);

% Set random seed.
rng(0);

% Shuffle the dataset before the split to ensure good class distibution.
ds = shuffle(ds);
dsTrain = subset(ds,trainingIdx);
dsVal = subset(ds,validationIdx);
dsTest = subset(ds,testIdx);
```

Analyze the Data

First, visualize a sample image from the dataset with the dataset.

```
data = read(dsTrain);
I = data{1,1};
box = data{1,2};
label = data{1,3};
I2 = insertObjectAnnotation(I,'rectangle',box,label);
imshow(I2)
```

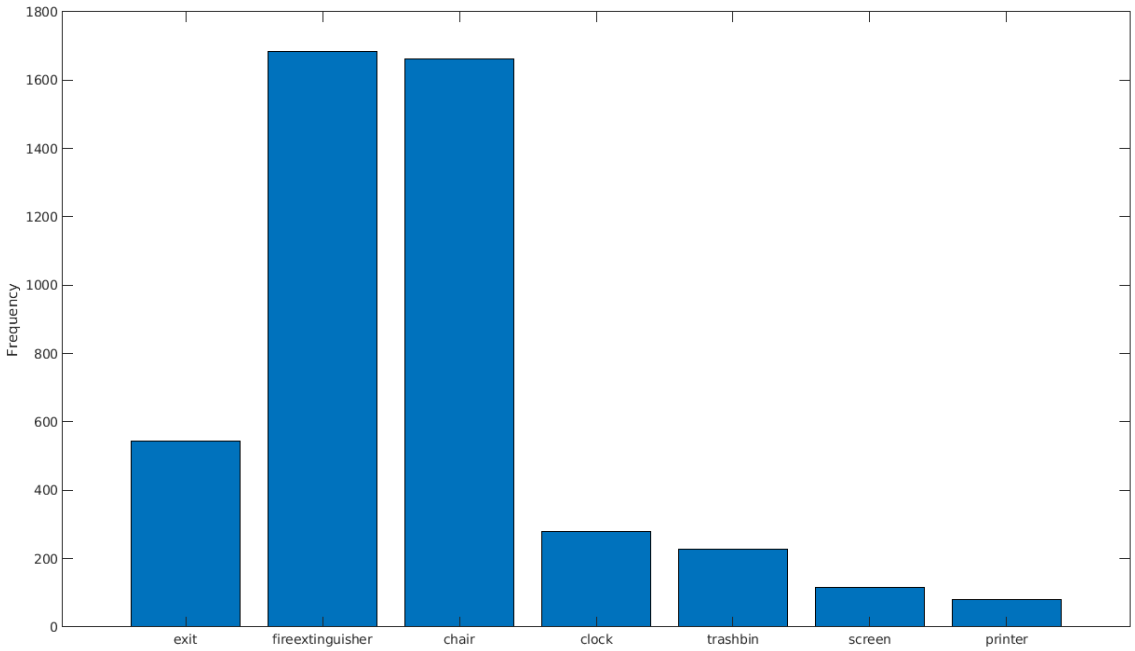
To measure distribution of class labels in the dataset, use `countEachLabel` to counts the number of objects by the class label.

```
bbStore = ds.UnderlyingDatastores{2};
tbl = countEachLabel(bbStore)
```

```
tbl=7×3 table
         Label          Count    ImageCount
    ________________    _____    __________

    exit                 545        504
    fireextinguisher    1684        818
    chair               1662        850
    clock                280        277
    trashbin             228        170
    screen               115         94
    printer               81         81
```

Visualize the counts by class.

```
bar(tbl.Label,tbl.Count)
ylabel('Frequency')
```



The classes in this dataset are unbalanced. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes. There are multiple techniques used to deal will this issue - oversampling the underrepresented classes, modifying loss function, and data augmentation. You will apply data augmentation to your training data in a later section.

Create a Yolov2 Object Detection Network

For this example, you will 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. This example uses ResNet-50 for feature extraction.

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. However, reducing image resolution can make it harder for the object detector to detect smaller objects. To maintain a balance between accuracy and computational cost of running the example, specify a network input size of [450 450 3].

```
inputSize = [450 450 3];
```

Define number of object classes to detect.

```
numClasses = 7;
```

Select the base network and the feature extraction layer. 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.

```
network = resnet50();
featureLayer = 'activation_40_relu';
```

Preprocess the training data to prepare data for training. The preprocessing function will resize images and the bounding boxes. In addition, it also sanitizes the bounding boxes to convert them to a valid shape.

```
preprocessedTrainingData = transform(dsTrain,@(data)resizeImageAndLabel(data,inputSize));
```

Next, use estimateAnchorBoxes to estimate two anchor boxes based on the size of objects in the training data. Choosing the optimal number of anchor boxes requires empirical analysis.

```
numAnchors = 2;
aboxes = estimateAnchorBoxes(preprocessedTrainingData,numAnchors);
```

Use the `yolov2Layers` function to create a YOLO v2 object detection network.

```
lgraph = yolov2Layers(inputSize,numClasses,aboxes,network,featureLayer);
```

You can visualize the network using `analyzeNetwork` or `DeepNetworkDesigner` from Deep Learning Toolbox.

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.

```
augmentedTrainingData = transform(preprocessedTrainingData,@augmentData);
```

Display one of the training images and box labels.

```
data = read(augmentedTrainingData);
I = data{1};
bbox = data{2};
label = data{3};
annotatedImage = insertObjectAnnotation(I,'rectangle',bbox,label);
figure
imshow(annotatedImage)
```

Train YOLOv2 Object Detector

Use `trainingOptions` to specify network training options.

```
opts = trainingOptions('rmsprop',...
        'InitialLearnRate',0.001,...
        'MiniBatchSize',4,...
        'MaxEpochs',10,...
        'LearnRateSchedule','piecewise',...
        'LearnRateDropPeriod',3,...
        'VerboseFrequency',30, ...
        'L2Regularization',0.001,...
        'ValidationData',dsVal,...
        'ValidationFrequency',50);
```

Use `trainYOLOv2ObjectDetector` function to train YOLO v2 object detector if `doTraining` is true.

```
doTraining = false;
if doTraining
```

```
    % Train the YOLO v2 detector.
    [detector, info] = trainYOLOv2ObjectDetector(augmentedTrainingData,lgraph,opts);
else
    % Load pretrained detector for the example.
    pretrained = load('yolov2IndoorObjectDetector.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 2 hours using this setup. Training time varies depending on the hardware you use.

Evaluate Detector Using Test Set

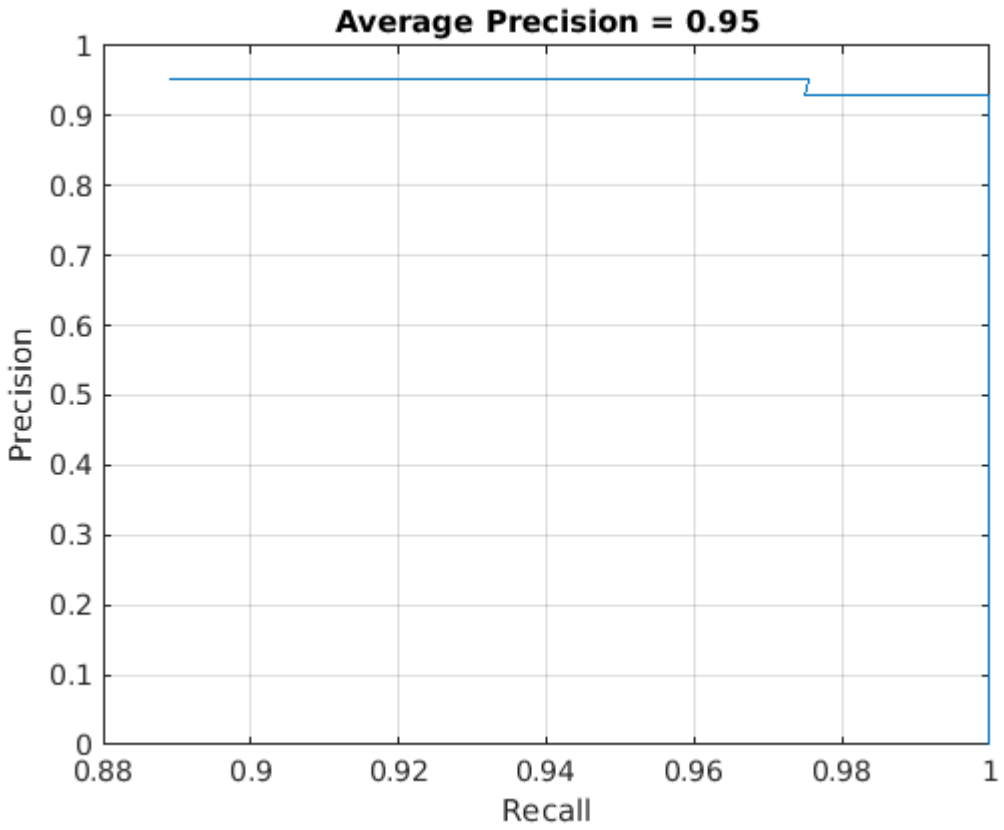
Evaluate the trained object detector on test images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this 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(dsTest,@(data)resizeImageAndLabel(data,inputSize));
results = detect(detector,preprocessedTestData,'MiniBatchSize',4,'Threshold',0.5);
[ap,precision,recall] = evaluateDetectionPrecision(results,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 for a selected class.

```
classID = 1;
figure
plot(recall{classID},precision{classID})
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap(classID)))
```



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);

% Sanitize boxes, if needed.
A{2} = helperSanitizeBoxes(A{2}, sz);

% 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
% helperSanitizeBoxes Sanitize box data.
% If none of the boxes are valid, this function passes the data through to
% enable downstream processing to issue proper errors.
function boxes = helperSanitizeBoxes(boxes, ~)
persistent hasInvalidBoxes
valid = all(boxes > 0, 2);
if any(valid)
    if ~all(valid) && isempty(hasInvalidBoxes)
        % Issue one-time warning about removing invalid boxes.
        hasInvalidBoxes = true;
        warning('Removing ground truth bouding box data with values <= 0.')
    end
    boxes = boxes(valid,:);
end
end

function data = resizeImageAndLabel(data,targetSize)
% Resize the images and scale the corresponding bounding boxes.

    scale = (targetSize(1:2))./size(data{1},[1 2]);
    data{1} = imresize(data{1},targetSize(1:2));
    data{2} = bboxresize(data{2},scale);

    data{2} = floor(data{2});
    imageSize = targetSize(1:2);
    boxes = data{2};
    % Set boxes with negative values to have value 1.
    boxes(boxes<=0) = 1;

    % Validate if bounding box in within image boundary.
    boxes(:,3) = min(boxes(:,3),imageSize(2) - boxes(:,1)-1);
    boxes(:,4) = min(boxes(:,4),imageSize(1) - boxes(:,2)-1);

    data{2} = boxes;

end
```

References

[1] Adhikari, Bishwo; Peltomaki, Jukka; Huttunen, Heikki. (2019). Indoor Object Detection Dataset [Data set]. 7th European Workshop on Visual Information Processing 2018 (EUVIP), Tampere, Finland.

Generate Adversarial Examples for Semantic Segmentation

This example shows how to generate adversarial examples for a semantic segmentation network using the basic iterative method (BIM).

Semantic segmentation is the process of assigning each pixel in an image a class label, for example, car, bike, person, or sky. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis.

Neural networks can be susceptible to a phenomenon known as adversarial examples [1], where very small changes to an input can cause it to be misclassified. These changes are often imperceptible to humans. This example shows how to generate an adversarial example for a semantic segmentation network.

This example generates adversarial examples using the CamVid [2] data set from the University of Cambridge. The CamVid data set is a collection of images containing street-level views obtained while driving. The data set provides pixel-level labels for 32 semantic classes including car, pedestrian, and road.

Load Network

Load a pretrained semantic segmentation network. This example loads a Deeplab v3+ network trained on the CamVid data set with weights initialized from a pretrained ResNet-18 network. For more information on building and training a Deeplab v3+ semantic segmentation network, see "Semantic Segmentation Using Deep Learning" (Deep Learning Toolbox).

```
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
```

Load the network and convert it to a `dlnetwork`.

```
data = load(pretrainedNetwork);
net = data.net;

lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,lgraph.Layers(end).Name);
net = dlnetwork(lgraph);
```

Load Image

Load an image and its corresponding label image. The image is a street-level view obtained from a car being driven. The label image contains the ground truth pixel labels. In this example, you create an adversarial example that causes the semantic segmentation network to misclassify the pixels in the `Bicyclist` class.

```
img = imread("0016E5_08145.png");
```

Use the supporting function `convertCamVidLabelImage`, defined at the end of this example, to convert the label image to a categorical array.

```
T = convertCamVidLabelImage(imread("0016E5_08145_L.png"));
```

The data set contains 32 classes. Use the supporting function `camVidClassNames11`, defined at the end of this example, to reduce the number of classes to 11 by grouping multiple classes from the original data set together.

```
classNames = camVidClassNames11;
```

Use the supporting function `camVidColorMap11` to create a colormap for the 11 classes.

```
cmap = camVidColorMap11;
```

Display the image with an overlay showing the pixels with the ground truth label `Bicyclist`.

```
classOfInterest = "Bicyclist";
notTheClassOfInterest = T ~= classOfInterest;

TClassOfInterest = T;
TClassOfInterest(notTheClassOfInterest) = "";

overlayImage = labeloverlay(img,TClassOfInterest,ColorMap=cmap);
imshow(overlayImage)
```

Create Adversarial Target Labels

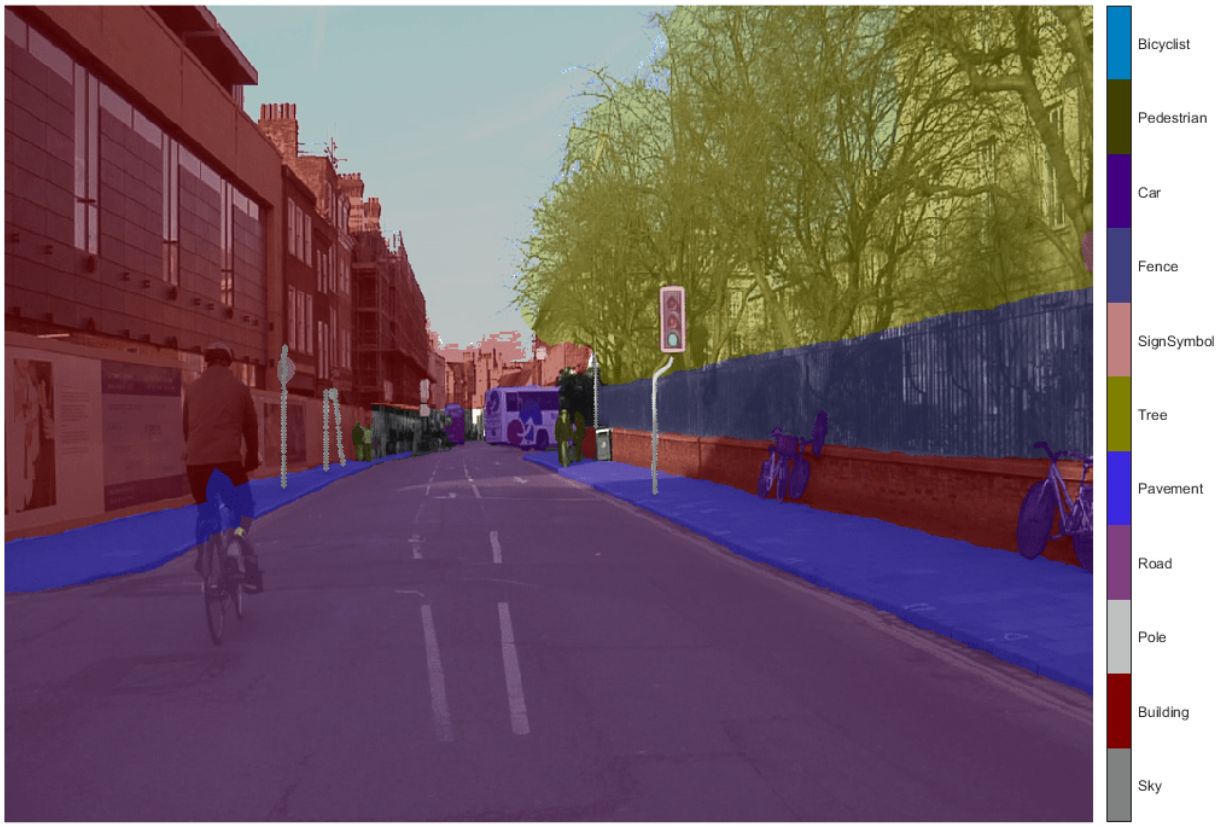
To create an adversarial example, you must specify the adversarial target label for each pixel you want the network to misclassify. In this example, the aim is to get the network to misclassify the `Bicyclist` pixels as another class. Therefore, you need to specify target classes for each of the `Bicyclist` pixels.

Using the supporting function `eraseClass`, defined at the end of this example, create adversarial target labels by replacing all `Bicyclist` pixel labels with the label of the nearest pixel that is not in the `Bicyclist` class [3].

```
TDesired = eraseClass(T,classOfInterest);
```

Display the adversarial target labels.

```
overlayImage = labeloverlay(img,TDesired,ColorMap=cmap);
figure
imshow(overlayImage)
pixelLabelColorbar(cmap,classNames);
```



The labels of the `Bicyclist` pixels are now `Road`, `Building`, or `Pavement`.

Prepare Data

To create the adversarial example using the image and the adversarial target labels, you must first prepare the image and the labels.

Prepare the image by converting it to a `dlarray`.

```
X = dlarray(single(img), "SSCB");
```

Prepare the label by one-hot encoding it. Because some of the pixels have undefined labels, replace `NaN` values with `0`.

```
TDesired = onehotencode(TDesired,3,"single",ClassNames=classNames);
TDesired(isnan(TDesired)) = 0;
TDesired = dlarray(TDesired,"SSCB");
```

Create Adversarial Example

Use the adversarial target labels to create an adversarial example using the basic iterative method (BIM) [4]. The BIM iteratively calculates the gradient $\nabla_X L(X, T)$ of the loss function L with respect to the image X you want to find an adversarial example for and the adversarial target labels T. The negative of this gradient describes the direction to "push" the image in to make the output closer to the desired class labels.

The adversarial example image is calculated iteratively as follows:

$X_{\text{adv}} = X - \alpha \cdot \text{sign}(\nabla_X L(X, T))$.

Parameter α controls the size of the push for a single iteration. After each iteration, clip the perturbation to ensure the magnitude does not exceed ϵ. Parameter ϵ defines a ceiling on how large the total change can be over all the iterations. A larger ϵ value increases the chance of generating a misclassified image, but makes the change in the image more visible.

Set the `epsilon` value to `5`, set the step size `alpha` to `1`, and perform `10` iterations.

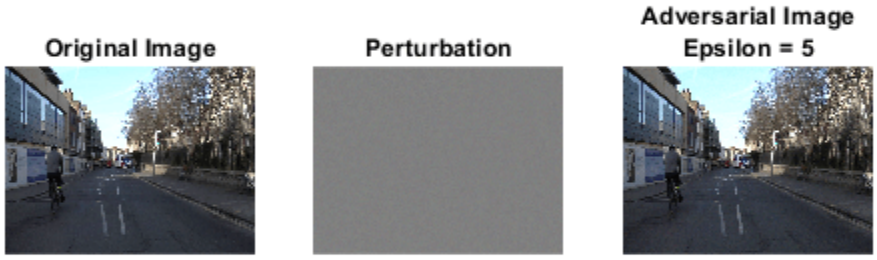
```
epsilon = 5;
alpha = 1;
numIterations = 10;
```

Keep track of the perturbation and clip any values that exceed `epsilon`.

```
delta = zeros(size(X),like=X);
for i = 1:numIterations
    gradient = dlfeval(@targetedGradients,net,X+delta,TDesired);

    delta = delta - alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdvTarget = X + delta;
```

Display the original image, the perturbation added to the image, and the adversarial image.

```
showAdversarialImage(X,XAdvTarget,epsilon)
```



The added perturbation is imperceptible, demonstrating how adversarial examples can exploit robustness issues within a network.

Predict Pixel Labels

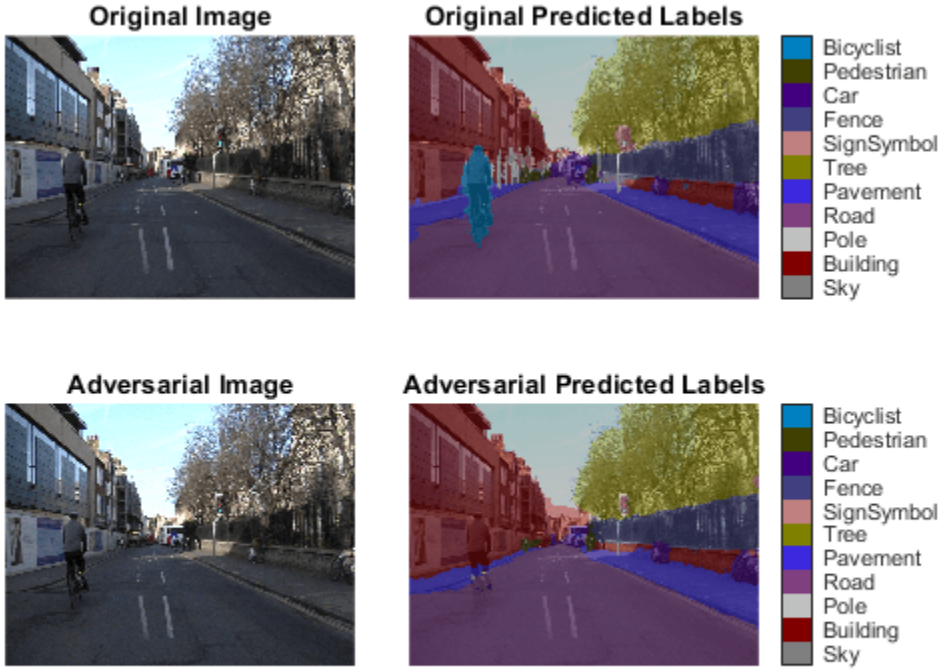
Predict the class labels of the original image and the adversarial image using the semantic segmentation network.

```
Y = semanticseg(extractdata(X),net);
YAdv = semanticseg(extractdata(XAdvTarget),net);
```

Display an overlay of the predictions for both images.

```
overlayImage = labeloverlay(uint8(extractdata(X)),Y,ColorMap=cmap);
overlayAdvImage = labeloverlay(uint8(extractdata(XAdvTarget)),YAdv,ColorMap=cmap);

figure
tiledlayout("flow",TileSpacing="tight")
nexttile
imshow(uint8(extractdata(X)))
title("Original Image")
nexttile
imshow(overlayImage)
pixelLabelColorbar(cmap,classNames);
title("Original Predicted Labels")
nexttile
imshow(uint8(extractdata(XAdvTarget)))
```

```
title("Adversarial Image")
nexttile
imshow(overlayAdvImage)
pixelLabelColorbar(cmap,classNames);
title("Adversarial Predicted Labels")
```



The network correctly identifies the bicyclist in the original image. However, because of imperceptible perturbation, the network mislabels the bicyclist in the adversarial image.

Supporting Functions

Convert CamVid Label Image to a Categorical

The supporting function `convertCamVidLabelImage` takes as input a label image from the CamVid data set and converts it to a categorical array.

```
function labelImage = convertCamVidLabelImage(image)

colorMap32 = camVidColorMap32;
map32To11 = cellfun(@(x,y)repmat(x,size(y,1),1), ...
    num2cell((1:numel(colorMap32))'), ...
    colorMap32, ...
    UniformOutput=false);

colorMap32 = cat(1,colorMap32{:});
map32To11 = cat(1,map32To11{:});

labelImage = rgb2ind(double(image)./255,colorMap32);
```

```
labelImage = map32To11(labelImage+1);
labelImage = categorical(labelImage,1:11,camVidClassNames11);

end
```

CamVid Color Map (32 classes)

The supporting function `camVidColorMap32` returns the color map for the 32 original classes in the CamVid data set.

```
function cmap = camVidColorMap32

cmap = {

    % Sky
    [
    128 128 128
    ]

    % Building
    [
      0 128  64     % Bridge
    128   0   0     % Building
     64 192   0     % Wall
     64   0  64     % Tunnel
    192   0 128     % Archway
    ]

    % Pole
    [
    192 192 128     % Column_Pole
      0   0  64     % TrafficCone
    ]

    % Road
    [
    128  64 128     % Road
    128   0 192     % LaneMkgsDriv
    192   0  64     % LaneMkgsNonDriv
    ]

    % Pavement
    [
      0   0 192     % Sidewalk
     64 192 128     % ParkingBlock
    128 128 192     % RoadShoulder
    ]

    % Tree
    [
    128 128   0     % Tree
    192 192   0     % VegetationMisc
    ]

    % SignSymbol
    [
    192 128 128     % SignSymbol
    128 128  64     % Misc_Text
```

```
      0  64  64     % TrafficLight
    ]

    % Fence
    [
     64  64 128     % Fence
    ]

    % Car
    [
     64   0 128     % Car
     64 128 192     % SUVPickupTruck
    192 128 192     % Truck_Bus
    192  64 128     % Train
    128  64  64     % OtherMoving
    ]

    % Pedestrian
    [
     64  64   0     % Pedestrian
    192 128  64     % Child
     64   0 192     % CartLuggagePram
     64 128  64     % Animal
    ]

    % Bicyclist
    [
      0 128 192     % Bicyclist
    192   0 192     % MotorcycleScooter
    ]

    % Void
    [
      0   0   0     % Void
    ]

    };

% Normalize between [0 1].
cmap = cellfun(@(x)x./255,cmap,UniformOutput=false);

end
```

CamVid Color Map (11 classes)

The supporting function `camVidColorMap11` returns the color map for the 11 umbrella classes in the CamVid data set.

```
function cmap = camVidColorMap11

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
```

CamVid Labels (11 classes)

The supporting function `classNames` returns the 11 umbrella classes of the CamVid data set.

```
function classNames = camVidClassNames11
classNames = [
    "Sky"
    "Building"
    "Pole"
    "Road"
    "Pavement"
    "Tree"
    "SignSymbol"
    "Fence"
    "Car"
    "Pedestrian"
    "Bicyclist"
    ];
end
```

Pixel Label Colorbar Function

The supporting function `pixelLabelColorbar` adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function pixelLabelColorbar(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
```

Erase Class Function

The supporting function `eraseClass` removes class `classToErase` from the label image `T` by relabeling the pixels in class `classToErase`. For each pixel in class `classToErase`, the `eraseClass` function sets the pixel label to the class of the nearest pixel not in class `classToErase`.

```
function TDesired = eraseClass(T,classToErase)
classToEraseMask = T == classToErase;
[~,idx] = bwdist(~(classToEraseMask | isundefined(T)));
TDesired = T;
TDesired(classToEraseMask) = T(idx(classToEraseMask));
end
```

Targeted Input Gradient Function

Calculate the gradient used to create a targeted adversarial example. The gradient is the gradient of the mean squared error.

```
function gradient = targetedGradients(net,X,target)
Y = predict(net,X);
loss = mse(Y,target);
gradient = dlgradient(loss,X);
end
```

Show Adversarial Image

Show an image, the corresponding adversarial image, and the difference between the two (perturbation).

```
function showAdversarialImage(image,imageAdv,epsilon)

figure
tiledlayout(1,3,TileSpacing="compact")
nexttile
imgTrue = uint8(extractdata(image));
imshow(imgTrue)
title("Original Image")

nexttile
perturbation = uint8(extractdata(imageAdv-image+127.5));
imshow(perturbation)
title("Perturbation")

nexttile
advImg = uint8(extractdata(imageAdv));
imshow(advImg)
title("Adversarial Image" + newline + "Epsilon = " + string(epsilon))
end
```

References

[1] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. “Explaining and Harnessing Adversarial Examples.” Preprint, submitted March 20, 2015. https://arxiv.org/abs/1412.6572.

[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.

[3] Fischer, Volker, Mummadi Chaithanya Kumar, Jan Hendrik Metzen, and Thomas Brox. “Adversarial Examples for Semantic Image Segmentation.” Preprint, submitted March 3, 2017. http://arxiv.org/abs/1703.01101.

[4] Kurakin, Alexey, Ian Goodfellow, and Samy Bengio. “Adversarial Examples in the Physical World.” Preprint, submitted February 10, 2017. https://arxiv.org/abs/1607.02533.

Classify Defects on Wafer Maps Using Deep Learning

This example shows how to classify eight types of manufacturing defects on wafer maps using a simple convolutional neural network (CNN).

Wafers are thin disks of semiconducting material, typically silicon, that serve as the foundation for integrated circuits. Each wafer yields several individual circuits (ICs), separated into dies. Automated inspection machines test the performance of ICs on the wafer. The machines produce images, called wafer maps, that indicate which dies perform correctly (pass) and which dies do not meet performance standards (fail).

The spatial pattern of the passing and failing dies on a wafer map can indicate specific issues in the manufacturing process. Deep learning approaches can efficiently classify the defect pattern on a large number of wafers. Therefore, by using deep learning, you can quickly identify manufacturing issues, enabling prompt repair of the manufacturing process and reducing waste.

This example shows how to train a classification network that detects and classifies eight types of manufacturing defect patterns. The example also shows how to evaluate the performance of the network.

Download WM-811K Wafer Defect Map Data

This example uses the WM-811K Wafer Defect Map data set [1 on page 3-0] [2 on page 3-0]. The data set consists of 811,457 wafer maps images, including 172,950 labeled images. Each image has only three pixel values. The value `0` indicates the background, the value `1` represents correctly behaving dies, and the value `2` represents defective dies. The labeled images have one of nine labels based on the spatial pattern of the defective dies. The size of the data set is 3.5 GB.

Set `dataDir` as the desired location of the data set. Download the data set using the `downloadWaferMapData` helper function. This function is attached to the example as a supporting file.

```
dataDir = fullfile(tempdir,"WaferDefects");
downloadWaferMapData(dataDir)
```

Preprocess and Augment Data

The data is stored in a MAT file as an array of structures. Load the data set into the workspace.

```
dataMatFile = fullfile(dataDir,"MIR-WM811K","WM811K.mat");
waferData = load(dataMatFile);
waferData = waferData.data;
```

Explore the data by displaying the first element of the structure. The `waferMap` field contains the image data. The `failureType` field contains the label of the defect.

```
disp(waferData(1))
```

```
          waferMap: [45×48 uint8]
           dieSize: 1683
           lotName: 'lot1'
         waferIndex: 1
    trainTestLabel: 'Training'
       failureType: 'none'
```

Reformat Data

This example uses only labeled images. Remove the unlabeled images from the structure.

```
unlabeledImages = zeros(size(waferData),"logical");
for idx = 1:size(unlabeledImages,1)
    unlabeledImages(idx) = isempty(waferData(idx).trainTestLabel);
end
waferData(unlabeledImages) = [];
```

The `dieSize`, `lotName`, and `waferIndex` fields are not relevant to the classification of the images. The example partitions data into training, validation, and test sets using a different convention than specified by `trainTestLabel` field. Remove these fields from the structure using the `rmfield` function.

```
fieldsToRemove = ["dieSize","lotName","waferIndex","trainTestLabel"];
waferData = rmfield(waferData,fieldsToRemove);
```

Specify the image classes.

```
defectClasses = ["Center","Donut","Edge-Loc","Edge-Ring","Loc","Near-full","Random","Scratch","no
numClasses = numel(defectClasses);
```

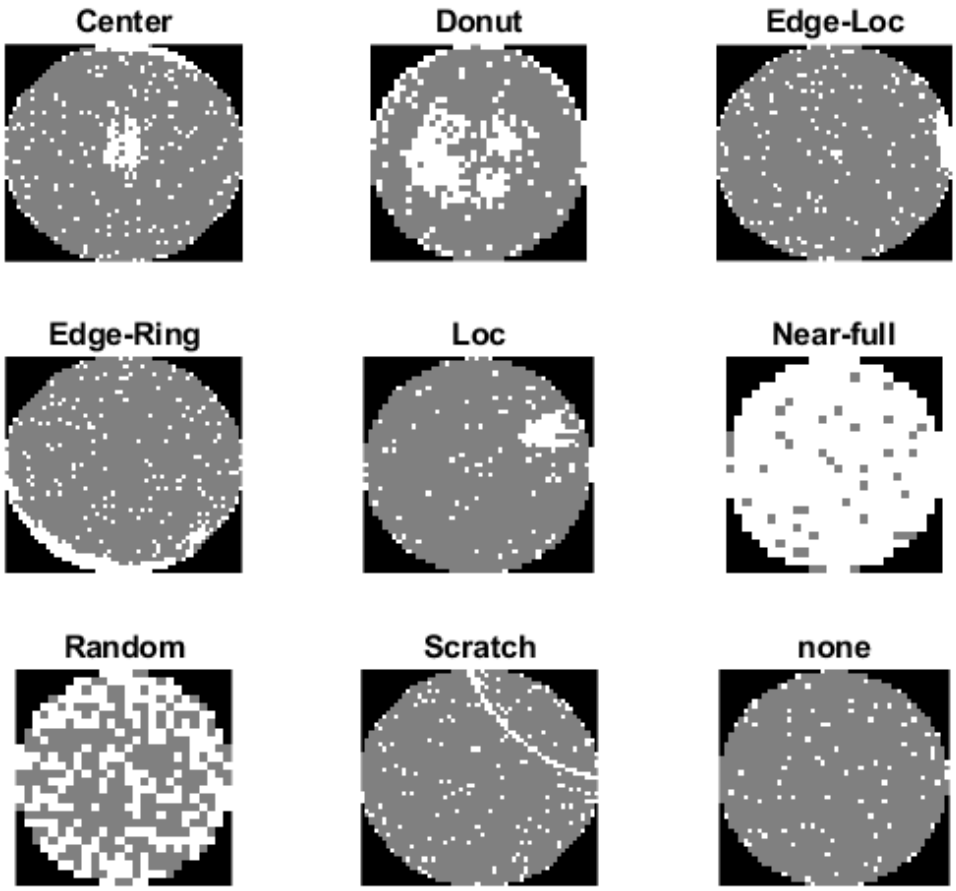
To apply additional preprocessing operations on the data, such as resizing the image to match the network input size or applying random train the network for classification, you can use an augmented image datastore. You cannot create an augmented image datastore from data in a structure, but you can create the datastore from data in a table. Convert the data into a table with two variables:

- `WaferImage` - Wafer defect map images
- `FailureType` - Categorical label for each image

```
waferData = struct2table(waferData);
waferData.Properties.VariableNames = ["WaferImage","FailureType"];
waferData.FailureType = categorical(waferData.FailureType,defectClasses);
```

Display a sample image from each input image class using the `displaySampleWaferMaps` helper function. This function is attached to the example as a supporting file.

```
displaySampleWaferMaps(waferData)
```



Balance Data By Oversampling

Display the number of images of each class. The data set is heavily unbalanced, with significantly fewer images of each defect class than the number of images without defects.

```
summary(waferData.FailureType)
```

```
    Center          4294
    Donut            555
    Edge-Loc        5189
    Edge-Ring       9680
    Loc             3593
    Near-full        149
    Random           866
    Scratch         1193
    none          147431
```

To improve the class balancing, oversample the defect classes using the `oversampleWaferDefectClasses` helper function. This function is attached to the example as a supporting file. The helper function appends the data set with five modified copies of each defect

image. Each copy has one of these modifications: horizontal reflection, vertical reflection, or rotation by a multiple of 90 degrees.

```
waferData = oversampleWaferDefectClasses(waferData);
```

Display the number of images of each class after class balancing.

```
summary(waferData.FailureType)
```

```
     Center          25764
     Donut            3330
     Edge-Loc        31134
     Edge-Ring       58080
     Loc             21558
     Near-full         894
     Random           5196
     Scratch          7158
     none           147431
```

Partition Data into Training, Validation, and Test Sets

Split the oversampled data set into training, validation, and test sets using the `splitlabels` function. Approximately 90% of the data is used for training, 5% is used for validation, and 5% is used for testing.

```
labelIdx = splitlabels(waferData,[0.9 0.05 0.05],"randomized",TableVariable="FailureType");
trainingData = waferData(labelIdx{1},:);
validationData = waferData(labelIdx{2},:);
testingData = waferData(labelIdx{3},:);
```

Augment Training Data

Specify a set of random augmentations to apply to the training data using an `imageDataAugmenter` (Deep Learning Toolbox) object. Adding random augmentations to the training images can avoid the network from overfitting to the training data.

```
aug = imageDataAugmenter(FillValue=0,RandXReflection=true,RandYReflection=true,RandRotation=[0 3
```

Specify the input size for the network. Create an `augmentedImageDatastore` (Deep Learning Toolbox) that reads the training data, resizes the data to the network input size, and applies random augmentations.

```
inputSize = [48 48];
dsTrain = augmentedImageDatastore(inputSize,trainingData,"FailureType",DataAugmentation=aug);
```

Create datastores that read validation and test data and resize the data to the network input size. You do not need to apply random augmentations to validation or test data.

```
dsVal = augmentedImageDatastore(inputSize,validationData,"FailureType");
dsVal.MiniBatchSize = 64;
dsTest = augmentedImageDatastore(inputSize,testingData,"FailureType");
```

Create Network

Define the convolutional neural network architecture. The range of the image input layer reflects the fact that the wafer maps have only three levels.

```
layers = [
    imageInputLayer([inputSize 1], ...
```

```
        Normalization="rescale-zero-one",Min=0,Max=2);

    convolution2dLayer(3,8,Padding="same")
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,Stride=2)

    convolution2dLayer(3,16,Padding="same")
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,Stride=2)

    convolution2dLayer(3,32,Padding="same")
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,Stride=2)

    convolution2dLayer(3,64,Padding="same")
    batchNormalizationLayer
    reluLayer

    dropoutLayer

    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify Training Options

Specify the training options for Adam optimization. Train the network for 30 epochs.

```
options = trainingOptions("adam", ...
    ResetInputNormalization=true, ...
    MaxEpochs=30, ...
    InitialLearnRate=0.001, ...
    L2Regularization=0.001, ...
    MiniBatchSize=64, ...
    Shuffle="every-epoch", ...
    Verbose=false, ...
    Plots="training-progress", ...
    ValidationData=dsVal, ...
    ValidationFrequency=20);
```

Train Network or Download Pretrained Network

By default, the example loads a pretrained wafer defect classification network. 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 using the `trainNetwork` (Deep Learning Toolbox) function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
doTraining = false;
if doTraining
    trainedNet = trainNetwork(dsTrain,layers,options);
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save(fullfile(dataDir,"trained-WM811K-"+modelDateTime+".mat"),"trainedNet");

else
    downloadTrainedWaferNet(dataDir);
    trainedNet = load(fullfile(dataDir,"CNN-WM811K.mat"));
    trainedNet = trainedNet.preTrainedNetwork;
end
```

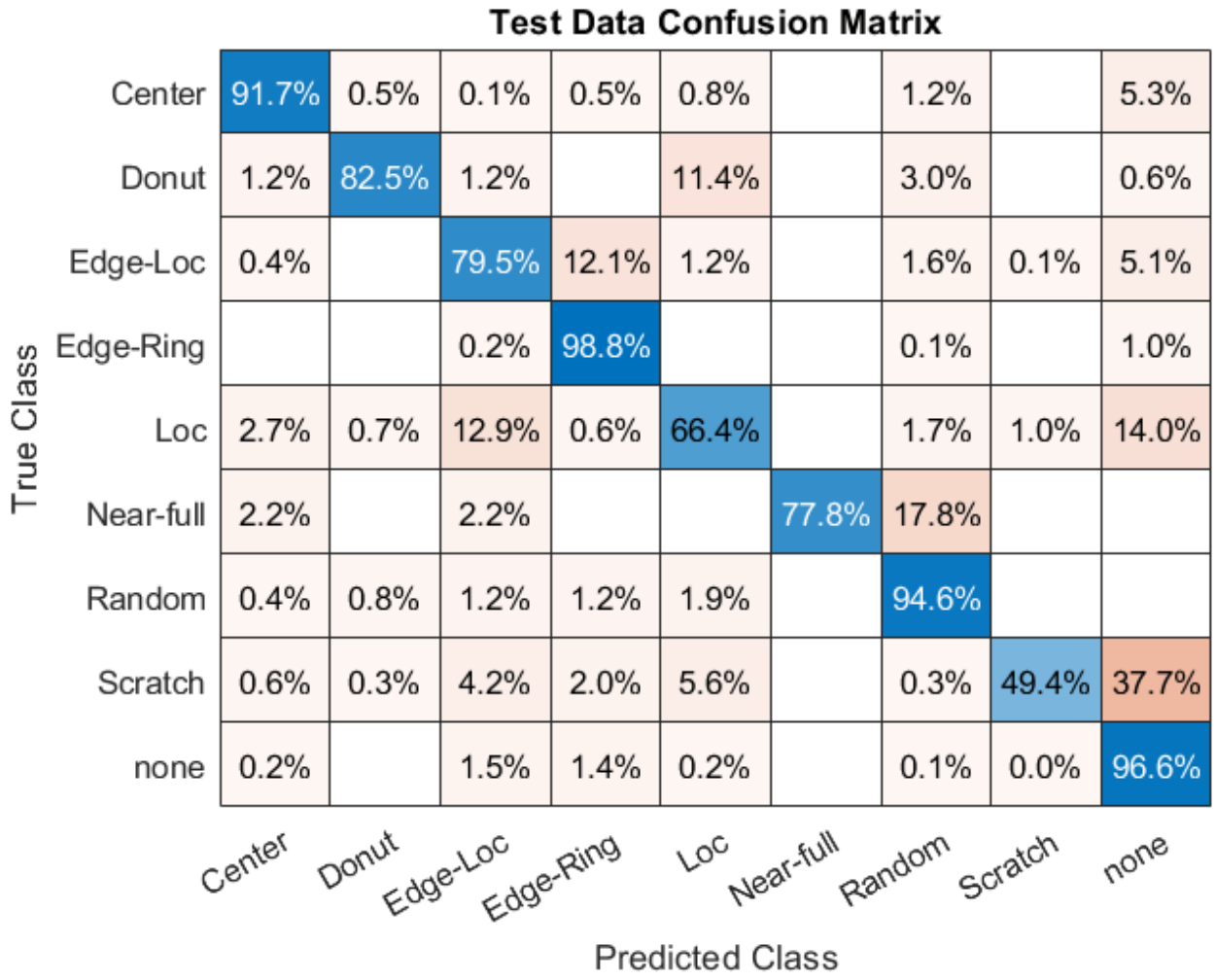
Quantify Network Performance on Test Data

Classify each of test image using the `classify` (Deep Learning Toolbox) function.

```
defectPredicted = classify(trainedNet,dsTest);
```

Calculate the performance of the network compared to the ground truth classifications as a confusion matrix using the `confusionmat` (Deep Learning Toolbox) function. Visualize the confusion matrix using the `confusionchart` (Deep Learning Toolbox) function. The values across the diagonal of this matrix indicate correct classifications. The confusion matrix for a perfect classifier has values only on the diagonal.

```
defectTruth = testingData.FailureType;

cmTest = confusionmat(defectTruth,defectPredicted);
figure
confusionchart(cmTest,categories(defectTruth),Normalization="row-normalized", ...
    Title="Test Data Confusion Matrix");
```



Precision, Recall, and F1 Scores

This example evaluates the network performance using several metrics: precision, recall, and F1 scores. These metrics are defined for a binary classification. To overcome the limitation for this multiclass problem, you can consider the prediction as a set of binary classifications, one for each class.

Precision is the proportion of images that are correctly predicted to belong to a class. Given the count of true positive (TP) and false positive (FP) classifications, you can calculate precision as:

$$\text{precision} = \frac{\text{TP}}{(\text{TP} + \text{FP})}$$

Recall is the proportion of images belonging to a specific class that were predicted to belong the class. Given the count of TP and false negative (FN) classifications, you can calculate recall as:

$$\text{recall} = \frac{\text{TP}}{(\text{TP} + \text{FN})}$$

F1 scores are the harmonic mean of the precision and recall values:

$$F1 = \frac{2 * \text{precision} * \text{recall}}{(\text{precision} + \text{recall})}$$

For each class, calculate the precision, recall, and F1 score using the counts of TP, FP, and FN results available in the confusion matrix.

```
prTable = table(Size=[numClasses 3],VariableTypes=["cell","cell","double"], ...
    VariableNames=["Recall","Precision","F1"],RowNames=defectClasses);

for idx = 1:numClasses
    numTP = cmTest(idx,idx);
    numFP = sum(cmTest(:,idx)) - numTP;
    numFN = sum(cmTest(idx,:),2) - numTP;

    precision = numTP / (numTP + numFP);
    recall = numTP / (numTP + numFN);

    defectClass = defectClasses(idx);
    prTable.Recall{defectClass} = recall;
    prTable.Precision{defectClass} = precision;
    prTable.F1(defectClass) = 2*precision*recall/(precision + recall);
end
```

Display the metrics for each class. Scores closer to 1 indicate better network performance.

```
prTable
```

```
prTable=9×3 table
                Recall      Precision       F1
              __________    __________    _______

    Center    {[0.9169]}    {[0.9547]}    0.93545
    Donut     {[0.8253]}    {[0.8839]}    0.85358
    Edge-Loc  {[0.7951]}    {[0.8188]}    0.80678
    Edge-Ring {[0.9876]}    {[0.9005]}    0.94203
    Loc       {[0.6642]}    {[0.8939]}    0.76211
    Near-full {[0.7778]}    {[     1]}      0.875
    Random    {[0.9462]}    {[0.7523]}    0.83816
    Scratch   {[0.4944]}    {[0.9219]}    0.64364
    none      {[0.9660]}    {[0.9390]}    0.95226
```

Precision-Recall Curves and Area-Under-Curve (AUC)

In addition to returning a classification of each test image, the network can also predict the probability that a test image is each of the defect classes. In this case, precision-recall curves provide an alternative way to evaluate the network performance.

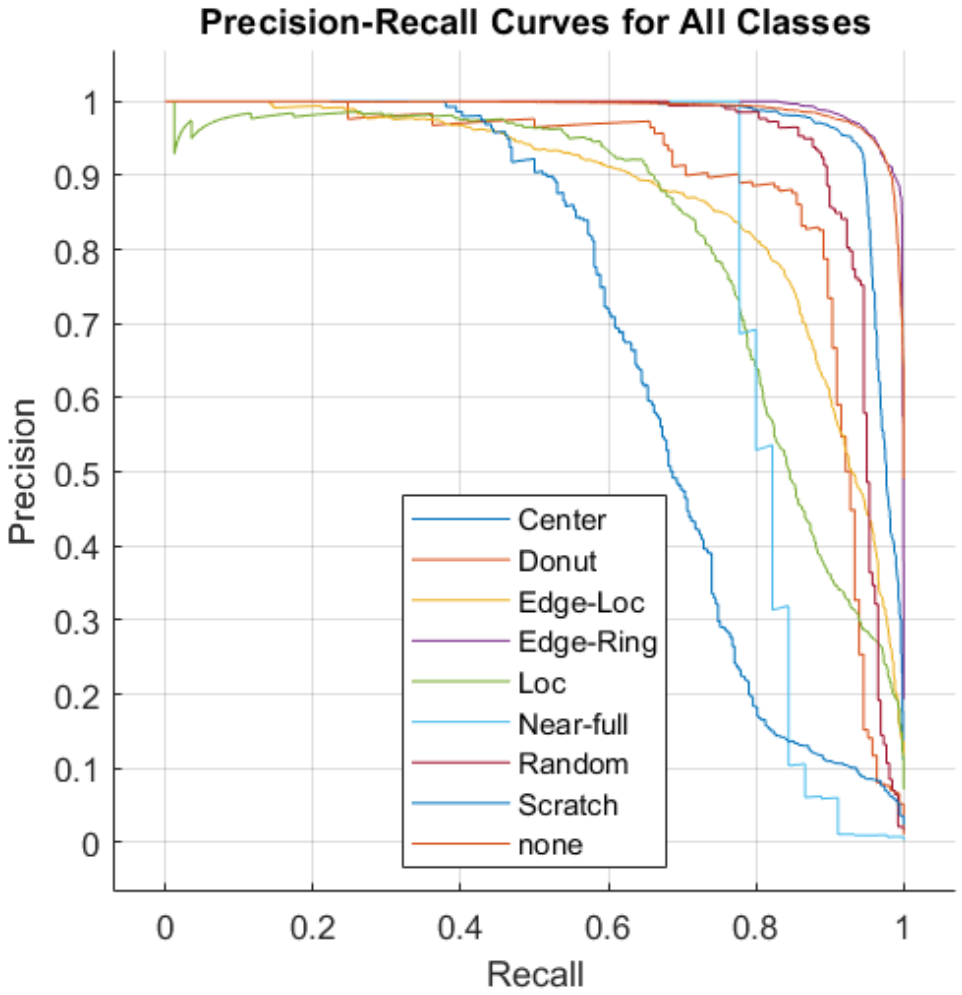
To calculate precision-recall curves, start by performing a binary classification for each defect class by comparing the probability against an arbitrary threshold. When the probability exceeds the threshold, you can assign the image to the target class. The choice of threshold impacts the number of TP, FP, and FN results and the precision and recall scores. To evaluate the network performance, you must consider the performance at a range of thresholds. Precision-recall curves plot the tradeoff between precision and recall values as you adjust the threshold for the binary classification. The AUC metric summarizes the precision-recall curve for a class as a single number in the range [0, 1], where 1 indicates a perfect classification regardless of threshold.

Calculate the probability that each test image belongs to each of the defect classes using the `predict` (Deep Learning Toolbox) function.

```
defectProbabilities = predict(trainedNet,dsTest);
```

Use the `rocmetrics` function to calculate the precision, recall, and AUC for each class over a range of thresholds. Plot the precision-recall curves.

```
roc = rocmetrics(defectTruth,defectProbabilities,defectClasses,AdditionalMetrics="prec");
figure
plot(roc,XAxisMetric="reca",YAxisMetric="prec");
xlabel("Recall")
ylabel("Precision")
grid on
title("Precision-Recall Curves for All Classes")
```



The precision-recall curve for an ideal classifier passes through the point (1, 1). The classes that have precision-recall curves that tend towards (1, 1), such as `Edge-Ring` and `Center`, are the classes for which the network has the best performance. The network has the worst performance for the `Scratch` class.

Compute and display the AUC values of the precision/recall curves for each class.

```
prAUC = zeros(numClasses, 1);
for idx = 1:numClasses
    defectClass = defectClasses(idx);
    currClassIdx = strcmpi(roc.Metrics.ClassName, defectClass);
    reca = roc.Metrics.TruePositiveRate(currClassIdx);
    prec = roc.Metrics.PositivePredictiveValue(currClassIdx);
    prAUC(idx) = trapz(reca(2:end),prec(2:end)); % prec(1) is always NaN
end
prTable.AUC = prAUC;
prTable
```

```
prTable=9×4 table
                Recall      Precision       F1         AUC
              __________    __________    _______    _______

    Center    {[0.9169]}    {[0.9547]}    0.93545    0.97074
    Donut     {[0.8253]}    {[0.8839]}    0.85358    0.88382
    Edge-Loc  {[0.7951]}    {[0.8188]}    0.80678    0.87182
    Edge-Ring {[0.9876]}    {[0.9005]}    0.94203    0.74137
    Loc       {[0.6642]}    {[0.8939]}    0.76211    0.82748
    Near-full {[0.7778]}    {[     1]}      0.875     0.7956
    Random    {[0.9462]}    {[0.7523]}    0.83816    0.93951
    Scratch   {[0.4944]}    {[0.9219]}    0.64364    0.69262
    none      {[0.9660]}    {[0.9390]}    0.95226    0.99076
```

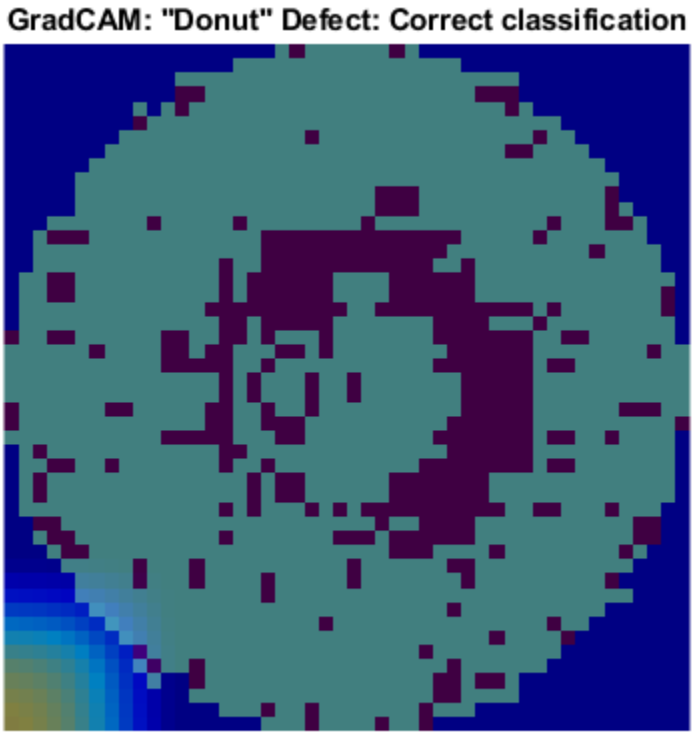
Visualize Network Decisions Using GradCAM

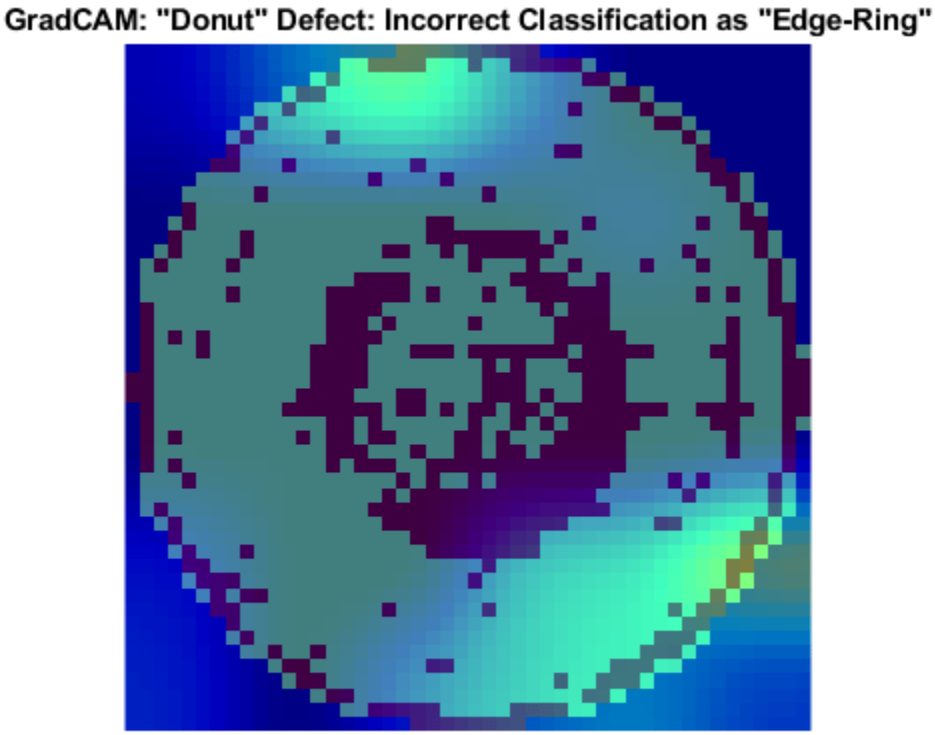
Gradient-weighted class activation mapping (Grad-CAM) produces a visual explanation of decisions made by the network. You can use the `gradCAM` (Deep Learning Toolbox) function to identify parts of the image that most influenced the network prediction.

`Donut` Defect Class

The `Donut` defect is characterized by an image having defective pixels clustered in a concentric circle around the center of the die. Most images of the `Donut` defect class do not have defective pixels around the edge of the die.

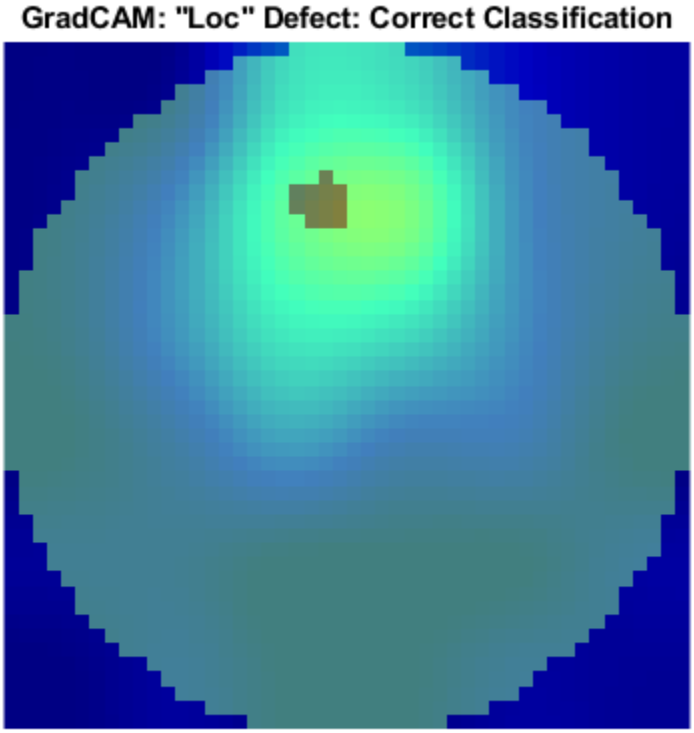
These two images both show data with the `Donut` defect. The network correctly classified the image on the left as a `Donut` defect. The network misclassified the image on the right as an `Edge-Ring` defect. The images have a color overlay that corresponds to the output of the `gradCAM` function. The regions of the image that most influenced the network classification appear with bright colors on the overlay. For the image classified as an `Edge-Ring` defect, the defects at the boundary at the die were treated as important. A possible reason for this could be there are far more `Edge-Ring` images in the training set as compared to `Donut` images.

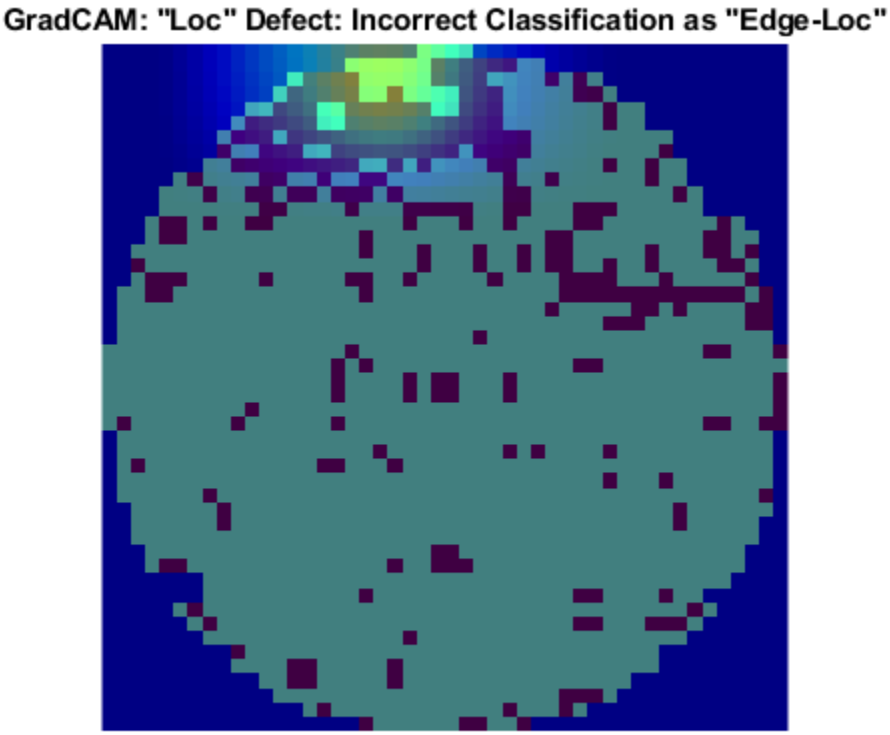

GradCAM: "Donut" Defect: Correct classification


GradCAM: "Donut" Defect: Incorrect Classification as "Edge-Ring"

`Loc` Defect Class

The `Loc` defect is characterized by an image having defective pixels clustered in a blob away from the edges of the die. These two images both show data with the Loc defect. The network correctly classified the image on the left as a `Loc` defect. The network misclassified the image on the right and classified the defect as an `Edge-Loc` defect. For the image classified as an `Edge-Loc` defect, the defects at the boundary at the die are most influential in the network prediction. The `Edge-Loc` defect differs from the `Loc` defect primarily in the location of the cluster of defects.


GradCAM: "Loc" Defect: Correct Classification


GradCAM: "Loc" Defect: Incorrect Classification as "Edge-Loc"

Compare Correct Classifications and Misclassifications

You can explore other instances of correctly classified and misclassified images. Specify a class to evaluate.

```
defectClass = Donut;
```

Find the index of all images with the specified defect type as the ground truth or predicted label.

```
idxTrue = find(testingData.FailureType == defectClass);
idxPred = find(defectPredicted == defectClass);
```

Find the indices of correctly classified images. Then, select one of the images to evaluate. By default, this example evaluates the first correctly classified image.

```
idxCorrect = intersect(idxTrue,idxPred);

idxToEvaluateCorrect = 1;
imCorrect = testingData.WaferImage{idxCorrect(idxToEvaluateCorrect)};
```

Find the indices of misclassified images. Then, select one of the images to evaluate and get the predicted class of that image. By default, this example evaluates the first misclassified image.

```
idxIncorrect = setdiff(idxTrue,idxPred);

idxToEvaluateIncorrect = 1;
imIncorrect = testingData.WaferImage{idxIncorrect(idxToEvaluateIncorrect)};
labelIncorrect = defectPredicted(idxIncorrect(idxToEvaluateIncorrect));
```

Resize the test images to match the input size of the network.

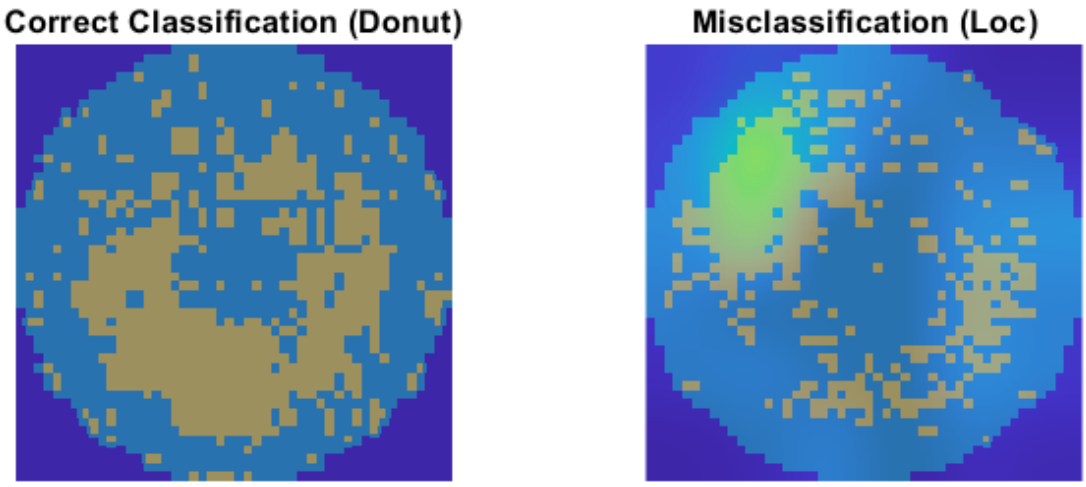
```
imCorrect = imresize(imCorrect,inputSize);
imIncorrect = imresize(imIncorrect,inputSize);
```

Generate the score maps using the `gradCAM` (Deep Learning Toolbox) function.

```
scoreCorrect = gradCAM(trainedNet,imCorrect,defectClass);
scoreIncorrect = gradCAM(trainedNet,imIncorrect,labelIncorrect);
```

Display the score maps over the original wafer maps using the `displayWaferScoreMap` helper function. This function is attached to the example as a supporting file.

```
figure
tiledlayout(1,2)
t = nexttile;
displayWaferScoreMap(imCorrect,scoreCorrect,t)
title("Correct Classification ("+defectClass+")")
t = nexttile;
displayWaferScoreMap(imIncorrect,scoreIncorrect,t)
title("Misclassification ("+string(labelIncorrect)+")")
```



References

[1] Wu, Ming-Ju, Jyh-Shing R. Jang, and Jui-Long Chen. “Wafer Map Failure Pattern Recognition and Similarity Ranking for Large-Scale Data Sets.” *IEEE Transactions on Semiconductor Manufacturing* 28, no. 1 (February 2015): 1–12. https://doi.org/10.1109/TSM.2014.2364237.

[2] Jang, Roger. "MIR Corpora." http://mirlab.org/dataset/public/.

[3] Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization.” In *2017 IEEE International Conference on Computer Vision (ICCV)*, 618–26. Venice: IEEE, 2017. https://doi.org/10.1109/ICCV.2017.74.

[4] T., Bex. “Comprehensive Guide on Multiclass Classification Metrics.” October 14, 2021. https://towardsdatascience.com/comprehensive-guide-on-multiclass-classification-metrics-af94cfb83fbd.

See Also

`trainingOptions` | `trainNetwork` | `augmentedImageDatastore` | `imageDataAugmenter` | `imageDatastore` | `classify` | `predict` | `confusionmat` | `confusionchart`

Related Examples

- “Detect Image Anomalies Using Explainable One-Class Classification Neural Network” on page 3-40
- “Detect Image Anomalies Using Pretrained ResNet-18 Feature Embeddings” on page 3-59

More About

- “Datastores for Deep Learning” (Deep Learning Toolbox)
- “Preprocess Images for Deep Learning” (Deep Learning Toolbox)
- “List of Deep Learning Layers” (Deep Learning Toolbox)

Detect Image Anomalies Using Explainable One-Class Classification Neural Network

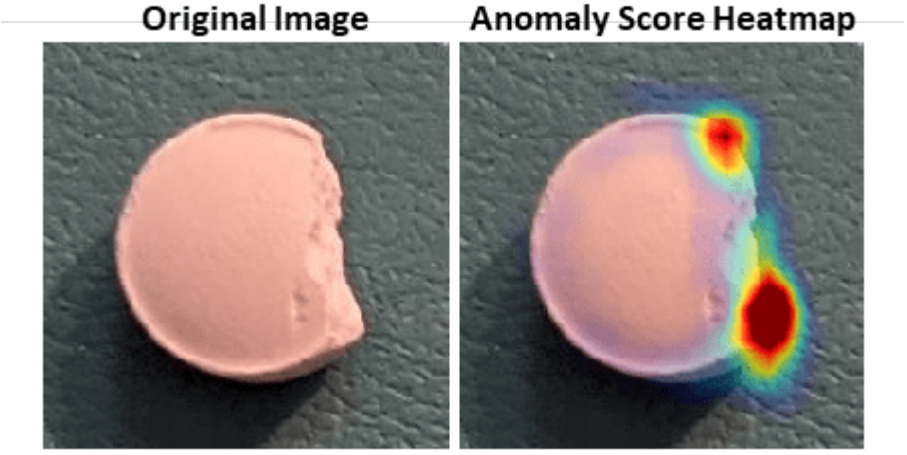
This example shows how to train an anomaly detector for visual inspection of pill images.

In one-class anomaly detection approaches, training is semisupervised, meaning that the network trains on data consisting only of normal images without anomalies [1 on page 3-0]. Despite training on samples only of normal scenes, the model learns how to distinguish between normal and anomalous scenes. One-class learning offers many advantages for anomaly detection problems:

- Representations of anomalies can be scarce.
- Anomalies can represent expensive or catastrophic outcomes.
- There can be many kinds of anomalies and the kinds of anomalies can change over the lifetime of the model. Describing what "good" looks like is often more feasible than providing data that represents all possible anomalies in real world settings.

A crucial goal of anomaly detection is for a human observer to be able to understand why a trained network classifies images as anomalies. *Explainable classification* supplements the class prediction with information that justifies how the neural network reached its classification decision.

This example explores how one-class deep learning can be used to create effective anomaly detectors. The example also implements explainable classification using a network that returns a heatmap with the probability that each pixel is anomalous. The classifier labels images as normal or anomalous based on the mean value of the anomaly score heatmap.



Download Pill Images for Classification Data Set

This example uses the pillQC data set. The data set contains images from three classes: `normal` images without defects, `chip` images with chip defects in the pills, and `dirt` images with dirt contamination. The data set provides 149 `normal` images, 43 `chip` images, and 138 `dirt` images. The size of the data set is 3.57 MB.

Set `dataDir` as the desired location of the data set. Download the data set using the `downloadPillQCData` helper function. This function is attached to the example as a supporting file. The function downloads a ZIP file and extracts the data into the subdirectories `chip`, `dirt`, and `normal`.

```
dataDir = fullfile(tempdir,"PillDefects");
downloadPillQCData(dataDir)
```

This image shows an example image from each class. A normal pill with no defects is on the left, a pill contaminated with dirt is in the middle, and a pill with a chip defect is on the right. While the images in this data set contain instances of shadows, focus blurring, and background color variation, the approach used in this example is robust to these image acquisition artifacts.

Load and Preprocess Data

Create an `imageDatastore` that reads and manages the image data. Label each image as `chip`, `dirt`, or `normal` according to the name of its directory.

```
imageDir = fullfile(dataDir,"pillQC-main","images");
imds = imageDatastore(imageDir,IncludeSubfolders=true,LabelSource="foldernames");
```

Partition Data into Training, Calibration, and Test Sets

To simulate a more typical semisupervised workflow, create a training set of 70 images from the `normal` class. Include two anomalous training images from each of the `chip` and `dirt` classes to get better classification results. Allocate 30 normal images and 15 images from each anomalous class to a calibration set. This example uses a calibration set to pick a threshold for the classifier. The classifier labels images with anomaly scores above the threshold as anomalous. Using separate calibration and test sets avoids information leaking from the test set into the design of the classifier. Assign the remaining images to the test set.

```
numTrainNormal = 70;
numTrainAnomaly = 2;
numCalNormal = 30;
numCalAnomaly = 15;

[imdsNormalTrain,imdsNormalCal,imdsNormalTest] = splitEachLabel(imds,numTrainNormal,numCalNormal
[imdsAnomalyTrain,imdsAnomalyCal,imdsAnomalyTest] = splitEachLabel(imds,numTrainAnomaly,numCalAno

imdsTrain = imageDatastore(vertcat(imdsNormalTrain.Files,imdsAnomalyTrain.Files),LabelSource="fo
imdsCal = imageDatastore(vertcat(imdsNormalCal.Files,imdsAnomalyCal.Files),LabelSource="foldernam
imdsTest = imageDatastore(vertcat(imdsNormalTest.Files,imdsAnomalyTest.Files),LabelSource="folde
trainLabels = countlabels(imdsTrain.Labels)
```

```
trainLabels=3×3 table
    Label     Count    Percent
```

```
_______    _____    _______

chip         2      2.7027
dirt         2      2.7027
normal      70      94.595
```

Augment Training Data

Augment the training data by using the `transform` function with custom preprocessing operations specified by the helper functions `augmentDataForPillAnomalyDetector` and `addConfettiNoiseForPillAnomalyDetector`. The helper functions are attached to the example as supporting files.

The `augmentDataForPillAnomalyDetector` function randomly applies 90 degree rotation and horizontal and vertical reflection to each input image. The `addConfettiNoiseForPillAnomalyDetector` function adds confetti noise to simulate local anomalies in normal images. Each normal image has a 50% likelihood of having confetti noise added to it. This augmentation step balances the training data between normal and anomalous, which helps to stabilize the loss function during training. Balancing the training data using simulated anomaly images is useful in applications where anomaly images are scarce.

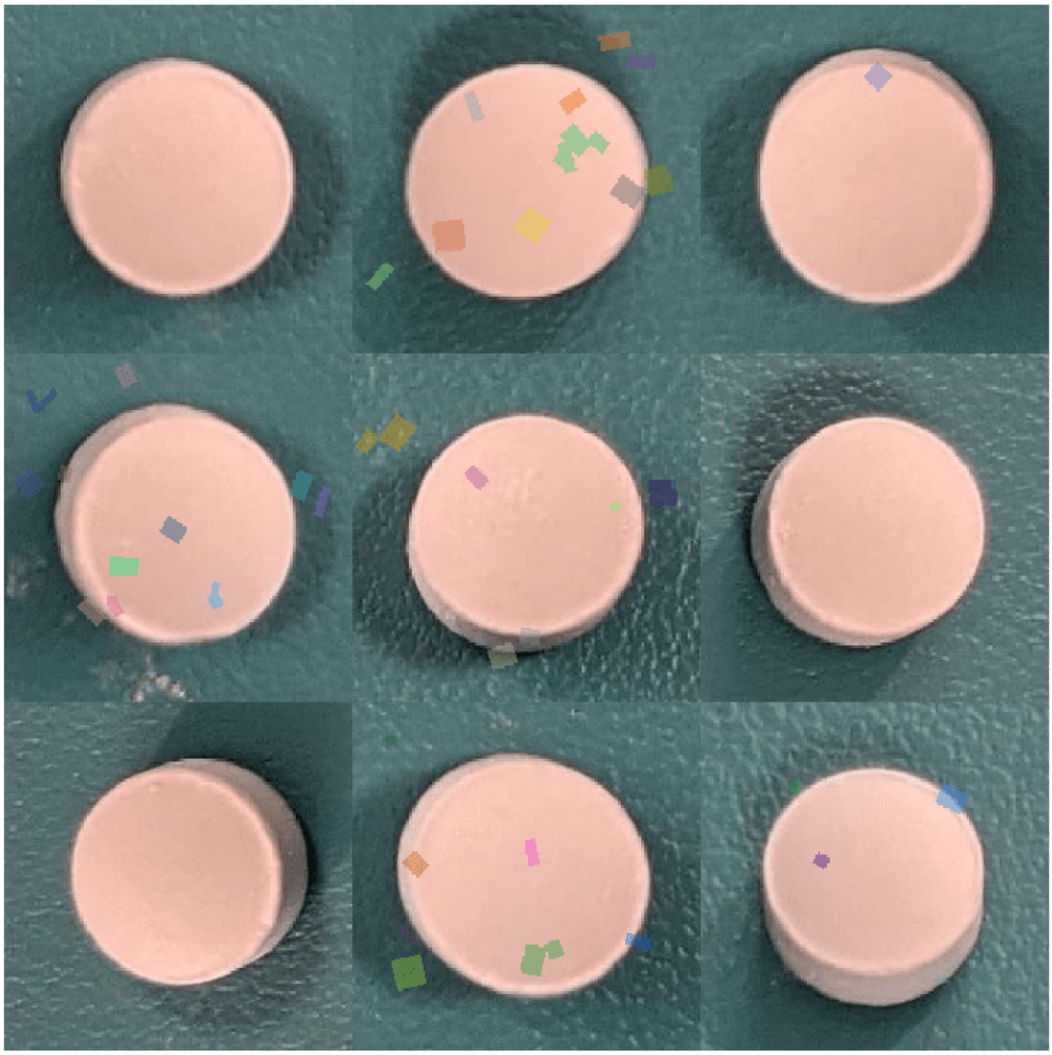
```
dsTrain = transform(imdsTrain,@augmentDataForPillAnomalyDetector);
dsTrain = transform(dsTrain,@addConfettiNoiseForPillAnomalyDetector,IncludeInfo=true);
```

Add binary labels to the calibration and test data sets by using the `transform` function with the operations specified by the `addLabelData` helper function. The helper function is defined at the end of this example, and assigns images in the `normal` class a binary label `0` and images in the `chip` or `dirt` classes a binary label `1`.

```
dsCal = transform(imdsCal,@addLabelData,IncludeInfo=true);
dsTest = transform(imdsTest,@addLabelData,IncludeInfo=true);
```

Visualize a sample of nine augmented training images. Approximately half of the total training images have confetti noise anomalies.

```
exampleData = readall(subset(dsTrain,1:9));
montage(exampleData(:,1));
```



Create FCDD Model

This example uses a fully convolutional data description (FCDD) model [1 on page 3-0]. The basic idea of FCDD is to train a network to produce an anomaly score map that describes the probability that each region in the input image contains anomaly content.

This example uses a VGG-16 network [3 on page 3-0] trained on ImageNet [4 on page 3-0] as the base fully convolutional network architecture. The example freezes the majority of the model and randomly initializes and trains the final convolutional stages. This approach enables quick training with small amounts of input training data.

The `vgg16` (Deep Learning Toolbox) function returns a pretrained VGG-16 network. This function requires the Deep Learning Toolbox™ Model for VGG-16 Network support package. If this support package is not installed, then the function provides a download link.

```
net = vgg16;
```

Replace the image input layer in the encoder with a new input layer that performs zero-center normalization using the computed mean. Set the input size of the network equal to the size of the images in the data set. Freeze the first 24 layers of the network by using the `freezeLayers` helper function. The helper function is defined at the end of this example.

```
inputSize = [225 225 3];
pretrainedVGG = [imageInputLayer(inputSize,Name="input",Normalization="zerocenter")
    net.Layers(2:24)];

pretrainedVGG = freezeLayers(pretrainedVGG);
```

Add a final convolutional stage. This stage is similar to the next convolutional stage of VGG-16 but has randomly initialized and trainable convolutional layers and with batch normalization. A 1-by-1 convolution compresses the network output into a one-channel anomaly score heatmap. The next layer is a pseudo-Huber loss function used to stabilize training with the FCDD loss and limit the output heatmap to the range [0, `inf`] [1 on page 3-0] [2 on page 3-0]. A resize layer is used to resize the output heatmap to the same size as the input image. A global averaging pooling layer computes the scalar anomaly score as the mean of the output heatmap returned by the network. A final custom loss layer `fcddLossLayerForPillAnomalyDetector` is used to implement the loss function [1 on page 3-0].

```
additionalFCLayers = [
    convolution2dLayer(3,512,Padding="same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,512,Padding="same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(1,1)
    functionLayer(@(x) sqrt(x.^2+1)-1)
    resize2dLayer(EnableReferenceInput=true,Method="bilinear",Name="upsampleHeatmap")
    globalAveragePooling2dLayer
    fcddLossLayer];
```

Assemble the complete network.

```
lgraph = layerGraph([pretrainedVGG;additionalFCLayers]);
lgraph = connectLayers(lgraph,"input","upsampleHeatmap/ref");
```

Train Network or Download Pretrained Network

By default, this example downloads a pretrained version of the VGG-16 network using the helper function `downloadTrainedNetwork`. The helper function is attached to this example as a supporting file. You can use the pretrained network 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`. Specify the number of epochs to use for training `numEpochs` by entering a value in the field. Train the model by using the `trainNetwork` (Deep Learning Toolbox) function.

Train on one or more GPUs, if available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 6 minutes on an NVIDIA Titan RTX™.

```
doTraining = false
numEpochs=100;
if doTraining
    options = trainingOptions("adam", ...
        Shuffle="every-epoch",...
        MaxEpochs=numEpochs,InitialLearnRate=1e-4, ...
        MiniBatchSize=numpartitions(dsTrain));
    net = trainNetwork(dsTrain,lgraph,options);
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save(fullfile(dataDir,"trainedPillAnomalyDetector-"+modelDateTime+".mat"),"net");
else
    trainedPillAnomalyDetectorNet_url = "https://ssd.mathworks.com/supportfiles/vision/data/trai
    downloadTrainedNetwork(trainedPillAnomalyDetectorNet_url,dataDir);
    net = load(fullfile(dataDir,"trainedAnomalyDetector","trainedPillFCDDNet.mat"));
    net = net.net;
end
```

Create Classification Model

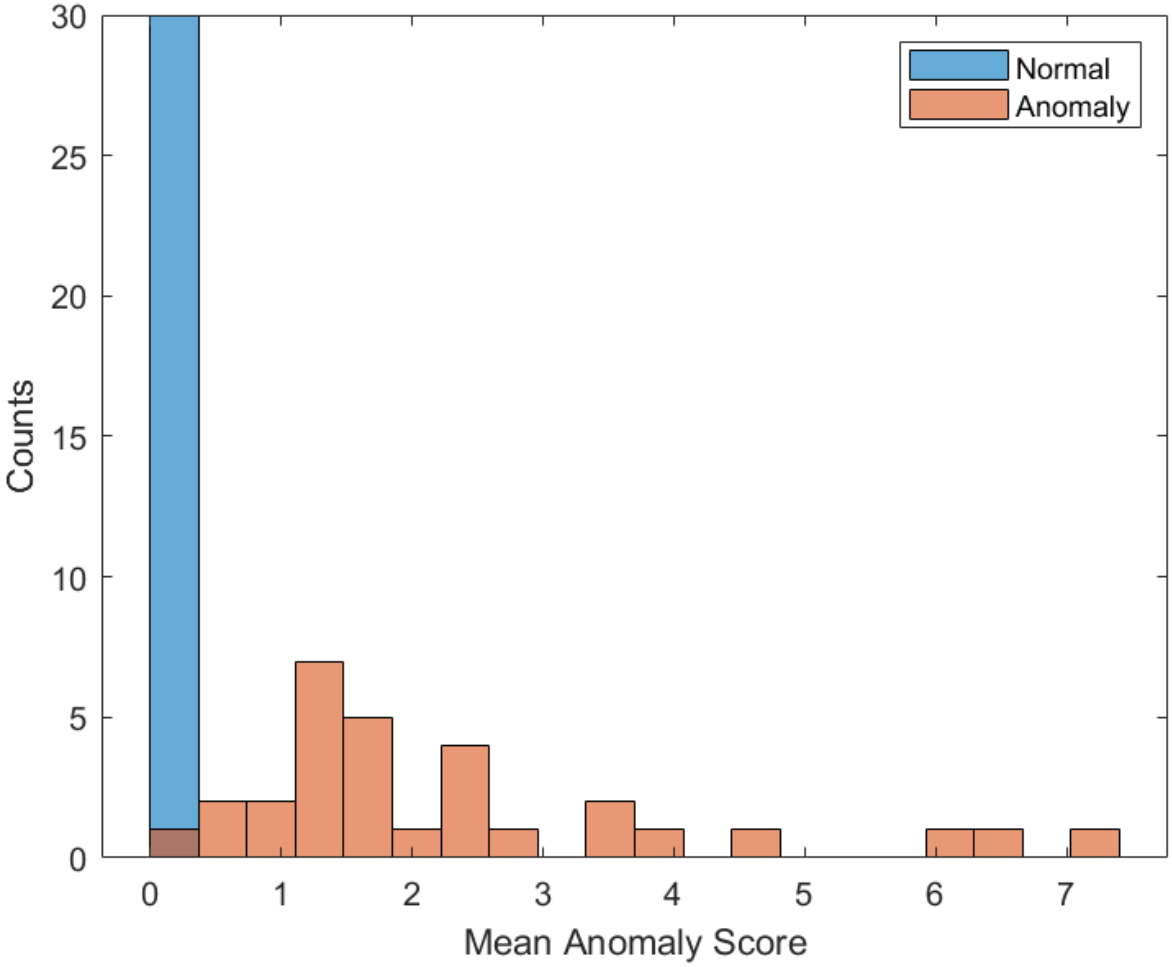
Classify an image as normal or anomalous based on whether the mean anomaly score of the image is greater than or less than a threshold value. The mean anomaly score is the average of the anomaly score heatmap. This example calculates the threshold that most accurately classifies the calibration image set.

Calculate the mean anomaly scores and known ground truth label (normal or anomaly) for each image in the calibration set.

```
scores = predict(net,dsCal);
labels = imdsCal.Labels ~= "normal";
```

Plot a histogram of the mean anomaly scores for the normal and anomaly classes. The distributions are well separated by the model-predicted anomaly score.

```
numBins = 20;
[~,edges] = histcounts(scores,numBins);
figure
hNormal = histogram(scores(labels==0),edges);
hold on
hAnomaly = histogram(scores(labels==1),edges);
hold off
legend([hNormal,hAnomaly],"Normal","Anomaly")
xlabel("Mean Anomaly Score");
ylabel("Counts");
```

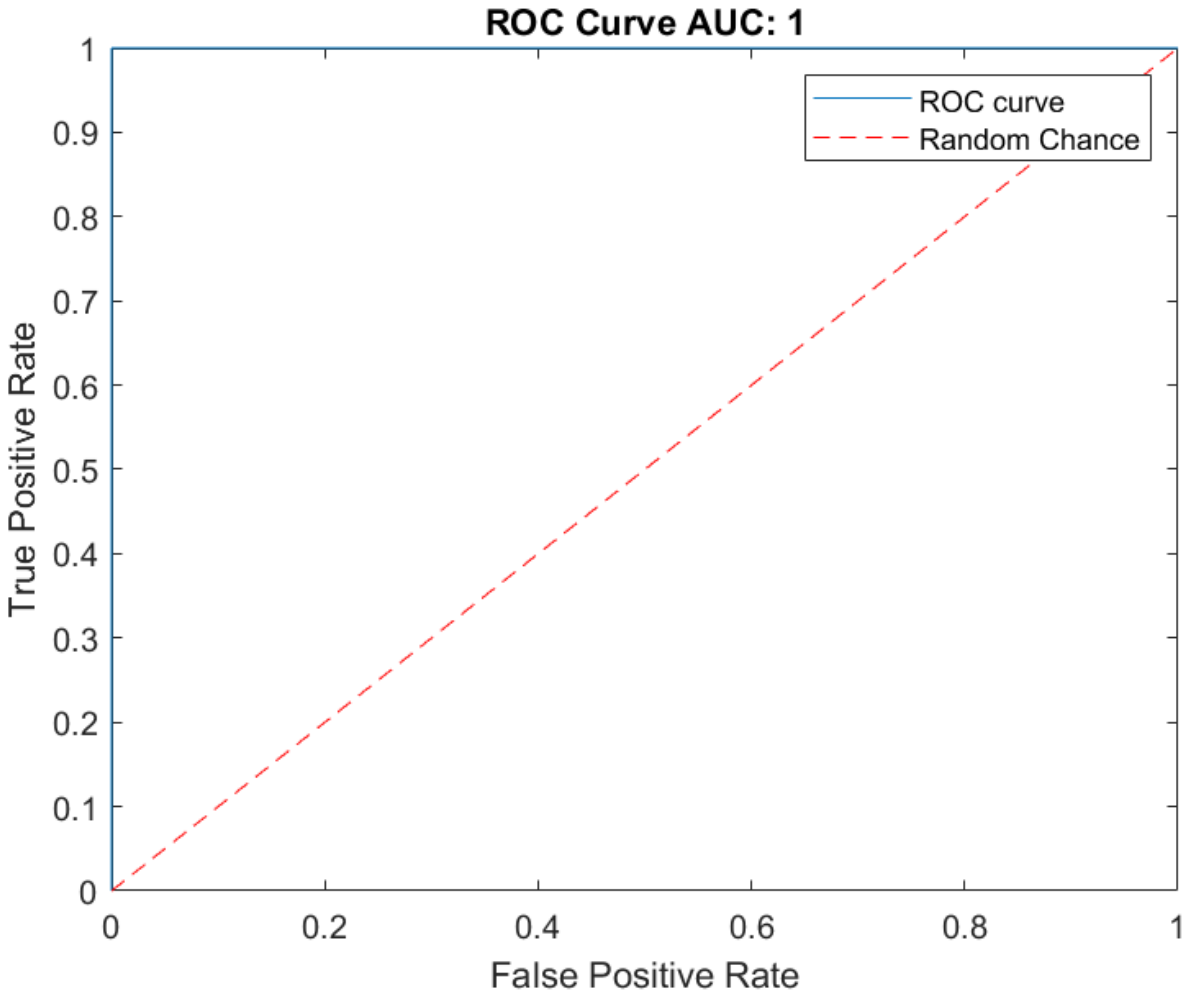


Create a receiver operating characteristic (ROC) curve to calculate the anomaly threshold. Each point on the ROC curve represents the false positive rate (*x*-coordinate) and true positive rate (*y*-coordinate) when the calibration set images are classified using a different threshold value. An optimal threshold maximizes the true positive rate and minimizes the false positive rate. Using ROC curves and related metrics allows you to select a threshold based on the tradeoff between false positives and false negatives. These tradeoffs depend on the application-specific implications of misclassifying images as false positives versus false negatives.

Create the ROC curve by using the `perfcurve` (Statistics and Machine Learning Toolbox) function. The solid blue line represents the ROC curve. The red dashed line represents a random classifier corresponding to a 50% success rate. Display the area under the curve (AUC) metric for the calibration set in the title of the figure. A perfect classifier has an ROC curve with a maximum AUC of 1.

```
[xroc,yroc,troc,auc] = perfcurve(labels,scores,true);
figure
lroc = plot(xroc,yroc);
hold on
lchance = plot([0 1],[0 1],"r--");
hold off
```

```
xlabel("False Positive Rate")
ylabel("True Positive Rate")
title("ROC Curve AUC: "+auc);
legend([lroc,lchance],"ROC curve","Random Chance")
```



This example uses the maximum Youden index metric to select the anomaly score threshold from the ROC curve. This value corresponds to the threshold that maximizes the distance between the blue model ROC curve and the red random chance ROC curve.

```
[~,ind] = max(yroc-xroc);
anomalyThreshold = troc(ind)
```

```
anomalyThreshold = single
    0.3696
```

Evaluate Classification Model

Predict the mean anomaly score for each image in the test set. Also get the ground truth labels of each test image.

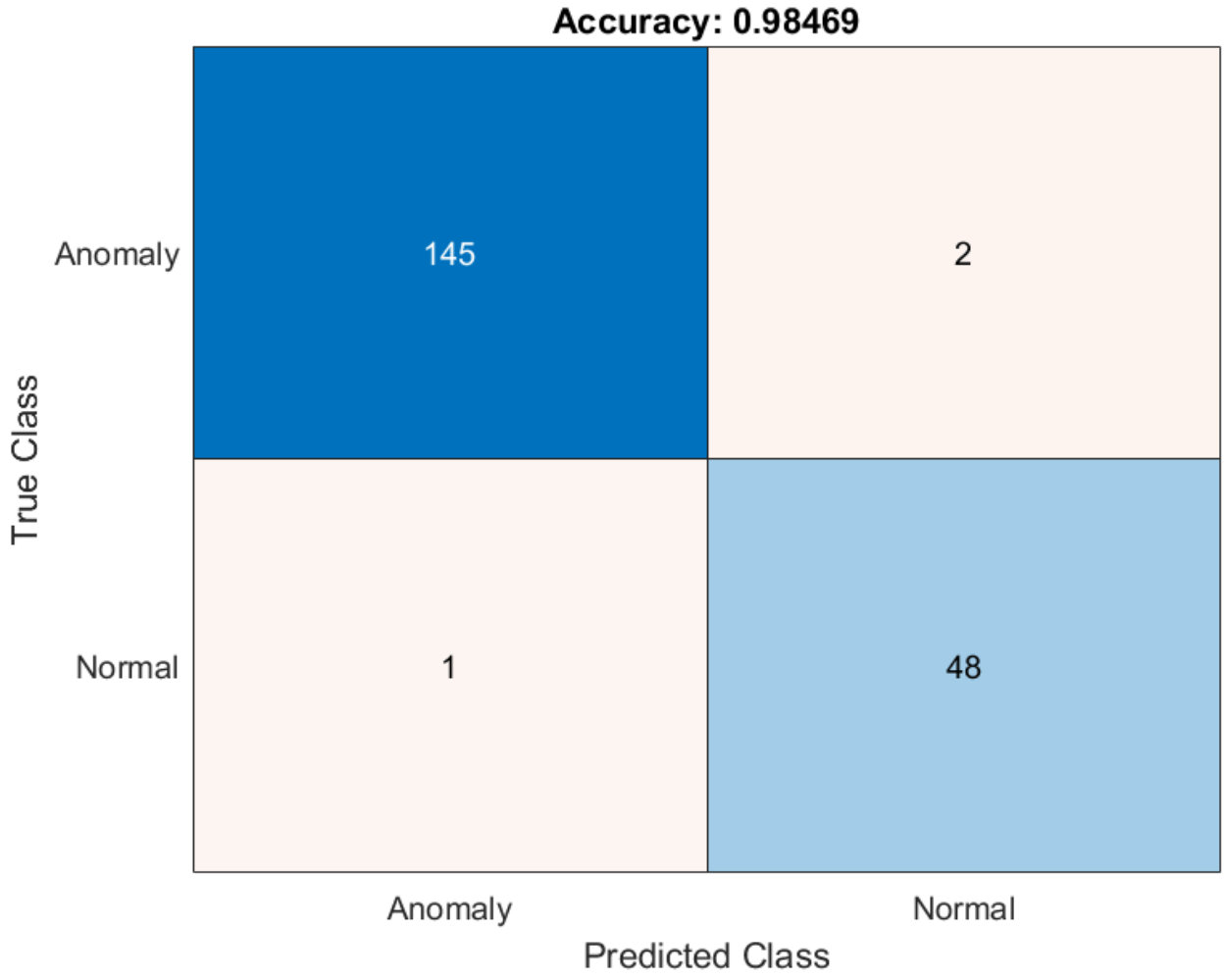
```
scores = predict(net,dsTest);
labels = imdsTest.Labels ~= "normal";
```

Assign a class label to test images by comparing the mean anomaly scores to the threshold value.

```
testSetOutputLabels = scores > anomalyThreshold;
```

Calculate the confusion matrix and classification accuracy for the test set. The classification model in this example is very accurate and predicts a small percentage of false positives and false negatives.

```
testSetTargetLabels = logical(labels);
M = confusionmat(testSetTargetLabels,testSetOutputLabels);
confusionchart(M,["Normal","Anomaly"])
acc = sum(diag(M)) / sum(M,"all");
title("Accuracy: "+acc);
```



Explain Classification Decisions

You can use the anomaly heatmap predicted by the network to help explain why an image is classified as normal or anomalous. This approach is useful for identifying patterns in false negatives and false positives. You can use these patterns to identify strategies for increasing class balancing of the training data or improving the network performance.

View Heatmap of Anomaly

Select an image of a correctly classified anomaly. This result is a true positive classification. Display the image.

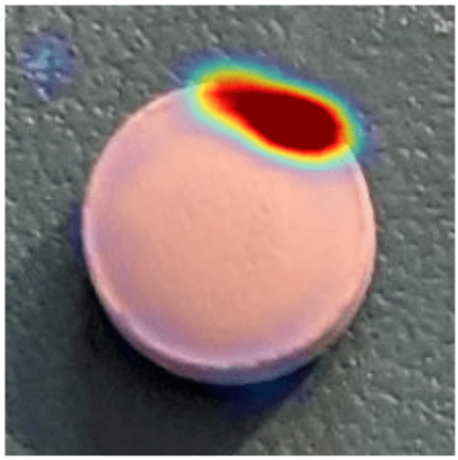
```
idxTruePositive = find(testSetTargetLabels & testSetOutputLabels);
dsExample = subset(dsTest,idxTruePositive);
data = read(dsExample);
img = data{1};
figure
imshow(img)
```

Obtain a heatmap of the anomaly image by extracting the activations of the `resize2dLayer` layer `upsampleHeatmap` of the network. The resize layer returns an anomaly score heatmap the same size as the input image.

```
map = activations(net,single(img),"upsampleHeatmap");
```

Display an overlay of the heatmap predicted by the network over the input image by using the `heatmapOverlay` helper function. This function is defined at the end of the example. Calculate a display range that reflects the range of heatmap values observed across the entire test set. Apply the display range for all heatmaps in this example. Set the minimum value of the `displayRange` to 0. Set the maximum value to the 80th percentile value of the heatmap for the test set image with the greatest mean anomaly score. Calculate the percentile value by using the `prctile` function.

```
[~,sampleIdx] = max(scores);
sampleMaxScore = read(subset(dsTest,sampleIdx));
heatmapMaxScore = activations(net,sampleMaxScore{1},"upsampleHeatmap");
displayRange = [0,prctile(heatmapMaxScore,80,"all")];

imshow(heatmapOverlay(img,map,displayRange))
```



To quantitatively confirm the result, display the mean anomaly score of the true positive test image as predicted by the network. The value is greater than the anomaly score threshold.

```
disp("Mean heatmap anomaly score of test image: "+scores(idxTruePositive(1)));
```

```
Mean heatmap anomaly score of test image: 1.1949
```

View Heatmap of Normal Image

Select and display an image of a correctly classified normal image. This result is a true negative classification.

```
idxTrueNegative = find(~(testSetTargetLabels | testSetOutputLabels));
dsExample = subset(dsTest,idxTrueNegative);
data = read(dsExample);
img = data{1};
imshow(img)
```

Obtain a heatmap of the normal image by extracting the activations of the `resize2dLayer` layer `upsampleHeatmap` of the network. Display an overlay of the heatmap predicted by the network over the input image by using the `heatmapOverlay` helper function. This function is defined at the end of the example. Many true negative test images, such as this test image, either have no visible anomalous regions or have low anomaly scores in a localized portion of the image.

```
map = activations(net,single(img),"upsampleHeatmap");
imshow(heatmapOverlay(img,map,displayRange))
```

Display the mean anomaly score of the true negative test image as predicted by the network. The value is less than the anomaly score threshold.

```
disp("Mean heatmap anomaly score of test image: "+scores(idxTrueNegative(1)));
```

```
Mean heatmap anomaly score of test image: 0.12476
```

View Heatmaps of False Negative Images

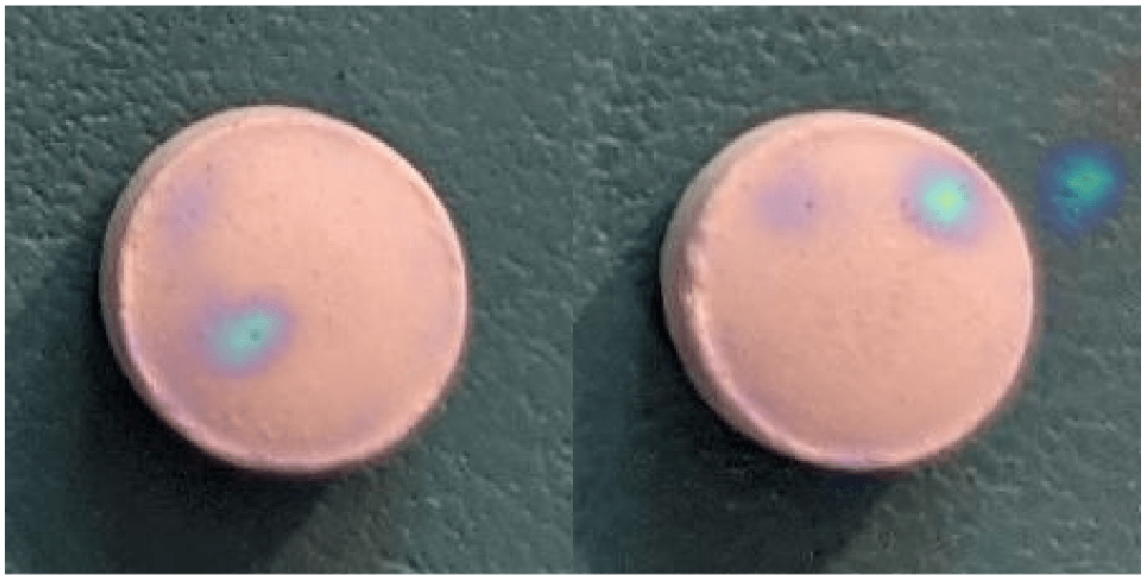
False negatives are images with pill defect anomalies that the network classifies as normal. Use the explanation from the network to gain insights into the misclassifications.

Find any false negative images from the test set. Obtain heatmap overlays of the false negative images by using the `transform` function. The operations of the transform are specified by an anonymous function that applies the `heatmapOverlay` helper function to the activations of the `resize2dLayer` layer `upsampleHeatmap` of the network. The `heatmapOverlay` helper function is defined at the end of the example. Display the false negative images as a montage. If there are no false negatives, the figure is empty.

```
falseNegativeIdx = find(testSetTargetLabels & ~testSetOutputLabels);
if ~isempty(falseNegativeIdx)
    fnExamples = subset(dsTest,falseNegativeIdx);
    fnExamplesWithHeatmapOverlays = transform(fnExamples,@(x) {heatmapOverlay(x{1},activations(ne
    fnExamples = readall(fnExamples);
    fnExamples = fnExamples(:,1);
    fnExamplesWithHeatmapOverlays = readall(fnExamplesWithHeatmapOverlays);
else
    [fnExamples,fnExamplesWithHeatmapOverlays] = deal([]);
end
montage(fnExamples)
```

Display the heatmap overlays as a montage. The network predicts visible anomaly scores around chip defects and specks of dirt, as expected.

```
montage(fnExamplesWithHeatmapOverlays)
```



Display the mean anomaly scores of the false negative test images as predicted by the network. The mean scores are less than the anomaly score threshold, resulting in misclassifications.

```
disp("Mean heatmap anomaly scores:"); scores(falseNegativeIdx)
```

```
Mean heatmap anomaly scores:

ans = 2×1 single column vector

    0.2603
    0.3277
```

View Heatmaps of False Positive Images

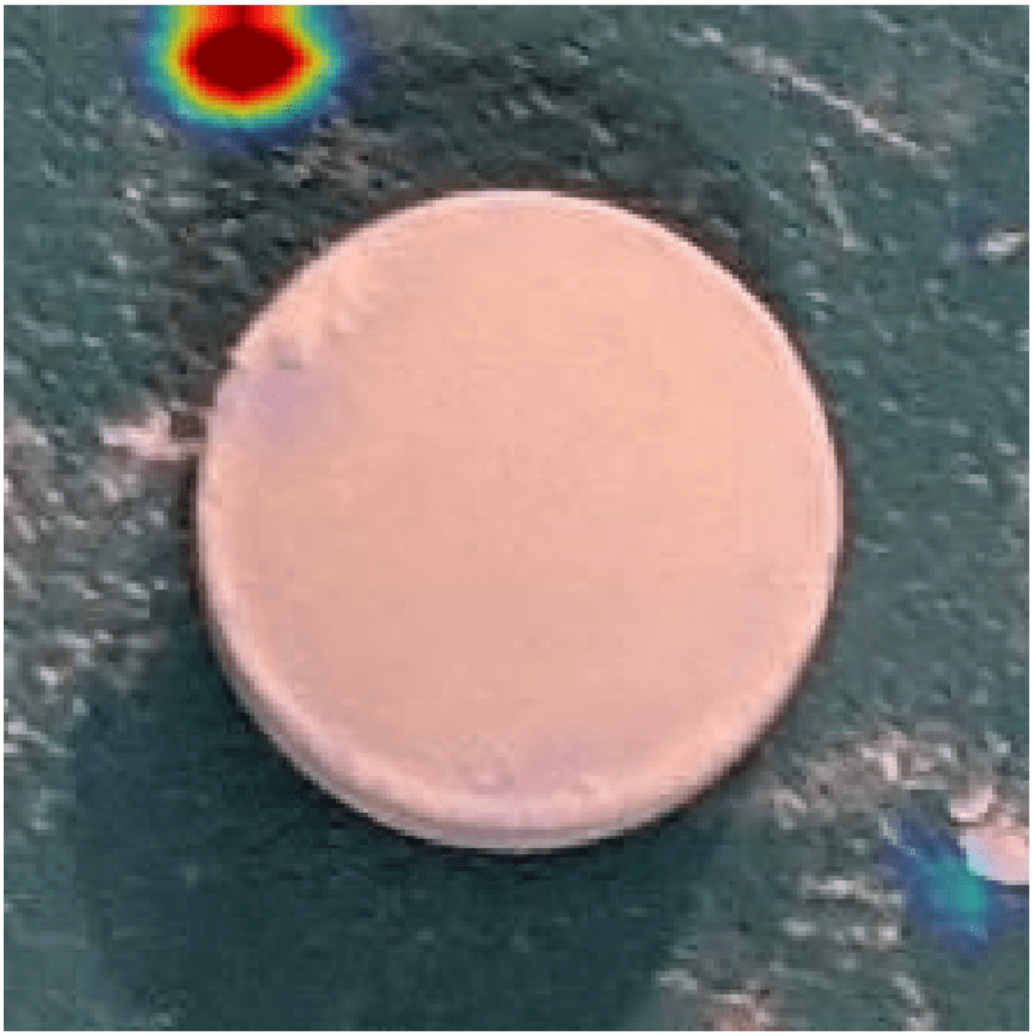
False positives are images without pill defect anomalies that the network classifies as anomalous. Use the explanation from the network to gain insights into the misclassifications.

Find any false positive images from the test set. Obtain heatmap overlays of the false positive images by using the `transform` function. The operations of the transform are specified by an anonymous function that applies the `heatmapOverlay` helper function to the activations of the `resize2dLayer` layer `upsampleHeatmap` of the network. The `heatmapOverlay` helper function is defined at the end of the example. Display the false positive images as a montage. If there are no false positives, the figure is empty.

```
falsePositiveIdx = find(~testSetTargetLabels & testSetOutputLabels);
if ~isempty(falsePositiveIdx)
    fpExamples = subset(dsTest,falsePositiveIdx);
    fpExamplesWithHeatmapOverlays = transform(fpExamples,@(x) {heatmapOverlay(x{1},activations(ne
    fpExamples = readall(fpExamples);
    fpExamples = fpExamples(:,1);
    fpExamplesWithHeatmapOverlays = readall(fpExamplesWithHeatmapOverlays);
else
    [fpExamples,fpExamplesWithHeatmapOverlays] = deal([]);
end
montage(fpExamples)
```

Display the heatmap overlays as a montage. The false positive images show regions that the network labeled as anomalous. You can use this explanation of the network behavior to gain insights into the classification problem. For example, if anomalous scores are localized to the image background, you can explore suppressing the background during preprocessing.

```
montage(fpExamplesWithHeatmapOverlays)
```



Display the mean anomaly scores of the false positive test images as predicted by the network. The mean scores are greater than the anomaly score threshold, resulting in misclassifications.

```
disp("Mean heatmap anomaly scores:"); scores(falsePositiveIdx)
```

```
Mean heatmap anomaly scores:

ans = single
    0.4467
```

Supporting Functions

The `freezeLayers` helper function freezes the network layers specified by the layer array `layers`.

```
function layers = freezeLayers(layers)
    for idx = 1:length(layers)
        if isprop(layers(idx),"Weights")
            layers(idx) = setLearnRateFactor(layers(idx),Weights=0);
            layers(idx) = setLearnRateFactor(layers(idx),Bias=0);
        end
    end
end
```

The `heatmapOverlay` helper function overlays a colored heatmap `hmap` with a display range specified by `displayRange` on the image `img`.

```
function out = heatmapOverlay(img,hmap,displayRange)

    % Normalize to the range [0, 1]
    img = mat2gray(img);
    hmap = rescale(hmap,InputMin=displayRange(1),InputMax=displayRange(2));

    % Convert heatmap to an RGB image using a colormap
    map = jet(256);
    hmapRGB = ind2rgb(gray2ind(hmap,size(map,1)),map);

    % Blend results
    hmapWeight = hmap;
    imgWeight = 1-hmapWeight;
    out = im2uint8(imgWeight.*img + hmapWeight.*hmapRGB);

end
```

The `addLabelData` helper function creates a one-hot encoded representation of label information in `data`.

```
function [data,info] = addLabelData(data,info)
    if info.Label == categorical("normal")
        onehotencoding = 0;
    else
        onehotencoding = 1;
    end
    data = {data,onehotencoding};
end
```

References

[1] Liznerski, Philipp, Lukas Ruff, Robert A. Vandermeulen, Billy Joe Franks, Marius Kloft, and Klaus-Robert Müller. "Explainable Deep One-Class Classification." Preprint, submitted March 18, 2021. https://arxiv.org/abs/2007.01760.

[2] Ruff, Lukas, Robert A. Vandermeulen, Billy Joe Franks, Klaus-Robert Müller, and Marius Kloft. "Rethinking Assumptions in Deep Anomaly Detection." Preprint, submitted May 30, 2020. https://arxiv.org/abs/2006.00339.

[3] Simonyan, Karen, and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." Preprint, submitted April 10, 2015. https://arxiv.org/abs/1409.1556.

[4] *ImageNet*. https://www.image-net.org.

See Also

`transform` | `minibatchqueue` | `vgg16` | `trainNetwork` | `predict` | `perfcurve` | `confusionmat` | `confusionchart`

Related Examples

- “Detect Image Anomalies Using Pretrained ResNet-18 Feature Embeddings” on page 3-59
- “Classify Defects on Wafer Maps Using Deep Learning” on page 3-24

More About

- “Datastores for Deep Learning” (Deep Learning Toolbox)
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)

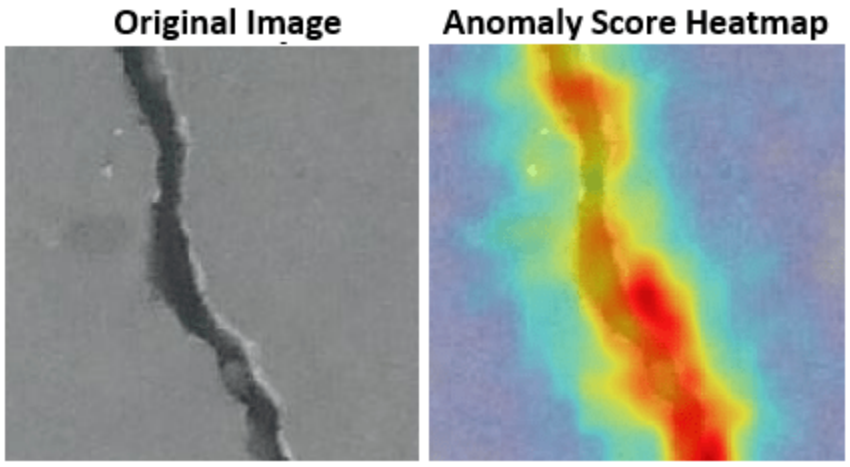
Detect Image Anomalies Using Pretrained ResNet-18 Feature Embeddings

This example shows how to train a similarity-based anomaly detector using one-class learning of feature embeddings extracted from a pretrained ResNet-18 convolutional neural network.

This example applies patch distribution modeling (PaDiM) [1 on page 3-0] to train an anomaly detection classifier. During training, you fit a Gaussian distribution that models the mean and covariance of normal image features. During testing, the classifier labels images whose features deviate from the Gaussian distribution by more than a certain threshold as anomalous. PaDiM is a similarity-based method because the similarity between test images and the normal image distribution drives classification. The PaDiM method has several practical advantages.

- PaDiM extracts features from a pretrained CNN without requiring that you retrain the network. Therefore, you can run the example efficiently without special hardware requirements such as a GPU.
- PaDiM is a *one-class learning* approach. The classification model is trained using only normal images. Training does not require images with anomalies, which can be rare, expensive, or unsafe to obtain for certain applications.
- PaDiM is an *explainable classification* method. The PaDiM classifier generates an anomaly score for each spatial patch. You can visualize the scores as a heatmap to localize anomalies and gain insight into the model.

The PaDiM method is suitable for image data sets that can be cropped to match the input size of the pretrained CNN. The input size of the CNN depends on the data used to train the network. For applications requiring more flexibility in image size, an alternative approach might be more appropriate. For an example of such an approach, see “Detect Image Anomalies Using Explainable One-Class Classification Neural Network”.



Download Concrete Crack Images for Classification Data Set

This example uses the Concrete Crack Images for Classification data set. The data set contains images of two classes: `Negative` images (or normal images) without cracks present in the road and `Positive` images (or anomaly images) with cracks. The data set provides 20,000 images of each class. The size of the data set is 235 MB.

Set `dataDir` as the desired location of the data set.

```
dataDir = fullfile(tempdir,"ConcreteCrackDataset");
if ~exist(dataDir,"dir")
    mkdir(dataDir);
end
```

To download the data set, go to this link: https://md-datasets-cache-zipfiles-prod.s3.eu-west-1.amazonaws.com/5y9wdsg2zt-2.zip. Extract the ZIP file to obtain a RAR file, then extract the contents of the RAR file into the directory specified by the `dataDir` variable. When extracted successfully, `dataDir` contains two subdirectories: `Negative` and `Positive`.

Load and Preprocess Data

Create an `imageDatastore` that reads and manages the image data. Label each image as `Positive` or `Negative` according to the name of its directory.

```
imdsPositive = imageDatastore(fullfile(dataDir,"Positive"),LabelSource="foldernames");
imdsNegative = imageDatastore(fullfile(dataDir,"Negative"),LabelSource="foldernames");
```

Display an example of each class. Display a negative, or good, image without crack anomalies on the left. In the good image, imperfections and deviations in texture are small. Display a positive, or anomalous, image on the right. The anomalous image shows a large black crack oriented vertically.

```
samplePositive = preview(imdsPositive);
sampleNegative = preview(imdsNegative);
montage({sampleNegative,samplePositive})
title("Road Images Without (Left) and with (Right) Cracks")
```

Partition Data into Training, Calibration, and Test Sets

To simulate a more typical semisupervised workflow, create a training set of 250 images from the `Negative` class only. Allocate 100 `Negative` images and 100 `Positive` images to a calibration set.

This example uses a calibration set to pick a threshold for the classifier. The classifier labels images with anomaly scores above the threshold as anomalous. Using separate calibration and test sets avoids information leaking from the test set into the design of the classifier. Allocate 1000 `Negative` images and 1000 `Positive` images to a test set.

```
numTrainNormal = 250;
numCal = 100;
numTest = 1000;

[imdsTestPos,imdsCalPos] = splitEachLabel(imdsPositive,numTest,numCal);
[imdsTrainNeg,imdsTestNeg,imdsCalNeg] = splitEachLabel(imdsNegative,numTrainNormal,numTest,numCa

trainFiles = imdsTrainNeg.Files;
calibrationFiles = cat(1,imdsCalPos.Files,imdsCalNeg.Files);
testFiles = cat(1,imdsTestPos.Files,imdsTestNeg.Files);

imdsTrain = imageDatastore(trainFiles,LabelSource="foldernames");
imdsCal = imageDatastore(calibrationFiles,LabelSource="foldernames");
imdsTest = imageDatastore(testFiles,LabelSource="foldernames");
```

Define an anonymous function, `addLabelFcn`, that creates a one-hot encoded representation of label information from an input image. Then, transform the datastores by using the `transform` function such that the datastores return a cell array of image data and a corresponding one-hot encoded array. The `transform` function applies the operations specified by `addLabelFcn`.

```
addLabelFcn = @(x,info) deal({x,onehotencode(info.Label,1)},info);
tdsTrain = transform(imdsTrain,addLabelFcn,IncludeInfo=true);
tdsCal = transform(imdsCal,addLabelFcn,IncludeInfo=true);
tdsTest = transform(imdsTest,addLabelFcn,IncludeInfo=true);
```

Resize and Crop Images

Define an anonymous function, `resizeAndCropImageFcn`, that applies the `resizeAndCropForConcreteAnomalyDetector` helper function to the input images. The `resizeAndCropForConcreteAnomalyDetector` helper function resizes and center crops input images, and is attached to the example as a supporting file. Transform the datastores by using the `transform` function with the operations specified by `resizeAndCropImageFcn`. This operation crops each image in the training, calibration, and test datastores to a size of 244-by-224 to match the input size of the pretrained CNN.

```
resizeImageSize = [256 256];
targetImageSize = [224 224];
resizeAndCropImageFcn = @(x,info) deal({resizeAndCropForConcreteAnomalyDetector(x{1},resizeImage
tdsTrain = transform(tdsTrain,resizeAndCropImageFcn);
tdsCal = transform(tdsCal,resizeAndCropImageFcn);
tdsTest = transform(tdsTest,resizeAndCropImageFcn);
```

Batch Training Data

Create a `minibatchqueue` (Deep Learning Toolbox) object that manages the mini-batches of training data. The `minibatchqueue` object automatically converts data to a `dlarray` (Deep Learning Toolbox) object that enables automatic differentiation in deep learning applications.

Specify the mini-batch data extraction format as `"SSCB"` (spatial, spatial, channel, batch).

```
minibatchSize = 128;
trainQueue = minibatchqueue(tdsTrain, ...
```

```
    PartialMiniBatch="return", ...
    MiniBatchFormat=["SSCB","CB"], ...
    MiniBatchSize=minibatchSize);
```

Create PaDiM Model

This example applies the PaDiM method described in [1 on page 3-0]. The basic idea of PaDiM is to simplify 2-D images into a lower resolution grid of embedding vectors that encode features extracted from a subset of layers of a pretrained CNN. Each embedding vector generated from the lower resolution CNN layers corresponds to a spatial patch of pixels in the original resolution image. The training step generates feature embedding vectors for all training set images and fits a statistical Gaussian distribution to the training data. A trained PaDiM classifier model consists of the mean and covariance matrix describing the learned Gaussian distribution for normal training images.

Extract Image Features from Pretrained CNN

This example uses the ResNet-18 network [2 on page 3-0] to extract features of input images. ResNet-18 is a convolutional neural network with 18 layers and is pretrained on ImageNet [3 on page 3-0].

Extract features from three layers of ResNet-18 located at the end of the first, second, and third blocks. For an input image of size 224-by-224, these layers correspond to activations with spatial resolutions of 56-by-56, 28-by-28, and 14-by-14, respectively. For example, the `XTrainFeatures1` variable contains 56-by-56 feature vectors from the `bn2b_branch2b` layer for each training set image. The layer activations with higher and lower spatial resolutions provide a balance between greater visual detail and global context, respectively.

```
net = resnet18("Weights","imagenet");

feature1LayerName = "bn2b_branch2b";
feature2LayerName = "bn3b_branch2b";
feature3LayerName = "bn4b_branch2b";

XTrainFeatures1 = []; %#ok<*UNRCH>
XTrainFeatures2 = [];
XTrainFeatures3 = [];

reset(trainQueue);
shuffle(trainQueue);
idx = 1;
while hasdata(trainQueue)
    [X,T] = next(trainQueue);

    XTrainFeatures1 = cat(4,XTrainFeatures1,activations(net,extractdata(X),feature1LayerName));
    XTrainFeatures2 = cat(4,XTrainFeatures2,activations(net,extractdata(X),feature2LayerName));
    XTrainFeatures3 = cat(4,XTrainFeatures3,activations(net,extractdata(X),feature3LayerName));
    idx = idx+size(X,4);
end
```

Concatenate Feature Embeddings

Combine the features extracted from the three ResNet-18 layers by using the `formAlignedEmbeddings` helper function defined at the end of this example. The `formAlignedEmbeddings` helper function upsamples the feature vectors extracted from the second and third blocks of ResNet-18 to match the spatial resolution of the first block and concatenates the three feature vectors.

```
XTrainEmbeddings = concatenateEmbeddings(XTrainFeatures1,XTrainFeatures2,XTrainFeatures3);
```

The variable `XTrainEmbeddings` is a numeric array containing feature embedding vectors for the training image set. The first two spatial dimensions correspond to the number of spatial patches in each image. The 56-by-56 spatial patches match the size of the `bn2b_branch2b` layer of ResNet-18. The third dimension corresponds to the channel data, or the length of the feature embedding vector for each patch. The fourth dimension corresponds to the number of training images.

```
whos XTrainEmbeddings
```

```
  Name                  Size                         Bytes  Class     Attributes

  XTrainEmbeddings      56x56x448x250           1404928000  single
```

Randomly Downsample Feature Embedding Channel Dimension

Reduce the dimensionality of the embedding vector by randomly selecting a subset of 100 out of 448 elements in the channel dimension to keep. As shown in [1 on page 3-0], this random dimensionality reduction step increases classification efficiency without decreasing accuracy.

```
selectedChannels = 100;
totalChannels = 448;
rIdx = randi(totalChannels,[1 selectedChannels]);
XTrainEmbeddings = XTrainEmbeddings(:,:,rIdx,:);
```

Compute Mean and Covariance of Gaussian Distribution

Model the training image patch embedding vectors as a Gaussian distribution by calculating the mean and covariance matrix across training images.

Reshape the embedding vector to have a single spatial dimension of length `H*W`.

```
[H, W, C, B] = size(XTrainEmbeddings);
XTrainEmbeddings = reshape(XTrainEmbeddings,[H*W C B]);
```

Calculate the mean of the embedding vector along the third dimension, corresponding to the average of the 250 training set images. In this example, the `means` variable is a 3136-by-100 matrix, with average feature values for each of the 56-by-56 spatial patches and 100 channel elements.

```
means = mean(XTrainEmbeddings,3);
```

For each embedding vector, calculate the covariance matrix between the 100 channel elements. Include a regularization constant based on the identity matrix to make `covars` a full rank and invertible matrix. In this example, the `covars` variable is a 3136-by-100-by-100 matrix.

```
covars = zeros([H*W C C]);
identityMatrix = eye(C);
for idx = 1:H*W
    covars(idx,:,:) = cov(squeeze(XTrainEmbeddings(idx,:,:))') + 0.01* identityMatrix;
end
```

Choose Anomaly Score Threshold for Classification

An important part of the semisupervised anomaly detection workflow is deciding on an anomaly score threshold for separating normal images from anomaly images. This example uses the calibration set to calculate the threshold.

In this example, the anomaly score metric is the Mahalanobis distance between the feature embedding vector and the learned Gaussian distribution for normal images. The anomaly score for each calibration image patch forms an anomaly score map that localizes predicted anomalies.

Calculate Anomaly Scores for Calibration Set

Calculate feature embedding vectors for the calibration set images. First, create a `minibatchqueue` (Deep Learning Toolbox) object to manage the mini-batches of calibration observations. Specify the mini-batch data extraction format as `"SSCB"` (spatial, spatial, channel, batch).

```
minibatchSize = 20;
calibrationQueue = minibatchqueue(tdsCal, ...
    MiniBatchFormat=["SSCB","CB"], ...
    MiniBatchSize=minibatchSize, ...
    OutputEnvironment="auto");
```

Perform the following steps to compute the anomaly scores for the calibration set images.

- Extract features of the calibration images from the same three layers of ResNet-18 used in training.
- Combine the features from the three layers into an overall embedding variable `XCalEmbeddings` by using the `formAlignedEmbeddings` helper function. The helper function is defined at the end of this example.
- Downsample the embedding vectors to the same 100 channel elements used during training, specified by `rIdx`.
- Reshape the embedding vectors into an `H*W`-by-`C`-by-`B` array, where `B` is the number of images in the mini-batch.
- Calculate the Mahalanobis distance between each embedding feature vector and the learned Gaussian distribution by using the `calculateDistance` helper function. The helper function is defined at the end of this example.
- Create an anomaly score map for each image by using the `createAnomalyScoreMap` helper function. The helper function is defined at the end of this example.

```
maxScoresCal = zeros(tdsCal.numpartitions,1);
minScoresCal = zeros(tdsCal.numpartitions,1);
meanScoresCal = zeros(tdsCal.numpartitions,1);
idx = 1;

while hasdata(calibrationQueue)
    XCal = next(calibrationQueue);

    XCalFeatures1 = activations(net,extractdata(XCal),feature1LayerName);
    XCalFeatures2 = activations(net,extractdata(XCal),feature2LayerName);
    XCalFeatures3 = activations(net,extractdata(XCal),feature3LayerName);

    XCalEmbeddings = concatenateEmbeddings(XCalFeatures1,XCalFeatures2,XCalFeatures3);

    XCalEmbeddings = XCalEmbeddings(:,:,rIdx,:);
    [H, W, C, B] = size(XCalEmbeddings);
    XCalEmbeddings = reshape(permute(XCalEmbeddings,[1 2 3 4]),[H*W C B]);

    distances = calculateDistance(XCalEmbeddings,H,W,B,means,covars);

    anomalyScoreMap = createAnomalyScoreMap(distances,H,W,B,targetImageSize);
```

```
    % Calculate max, min, and mean values of the anomaly score map
    maxScoresCal(idx:idx+size(XCal,4)-1) = squeeze(max(anomalyScoreMap,[],[1 2 3]));
    minScoresCal(idx:idx+size(XCal,4)-1) = squeeze(min(anomalyScoreMap,[],[1 2 3]));
    meanScoresCal(idx:idx+size(XCal,4)-1) = squeeze(mean(anomalyScoreMap,[1 2 3]));

    idx = idx+size(XCal,4);
    clear XCalFeatures1 XCalFeatures2 XCalFeatures3 anomalyScoreMap distances XCalEmbeddings XCa
end
```

Create Anomaly Score Histograms

Assign the known ground truth labels `"Positive"` and `"Negative"` to the calibration set images.

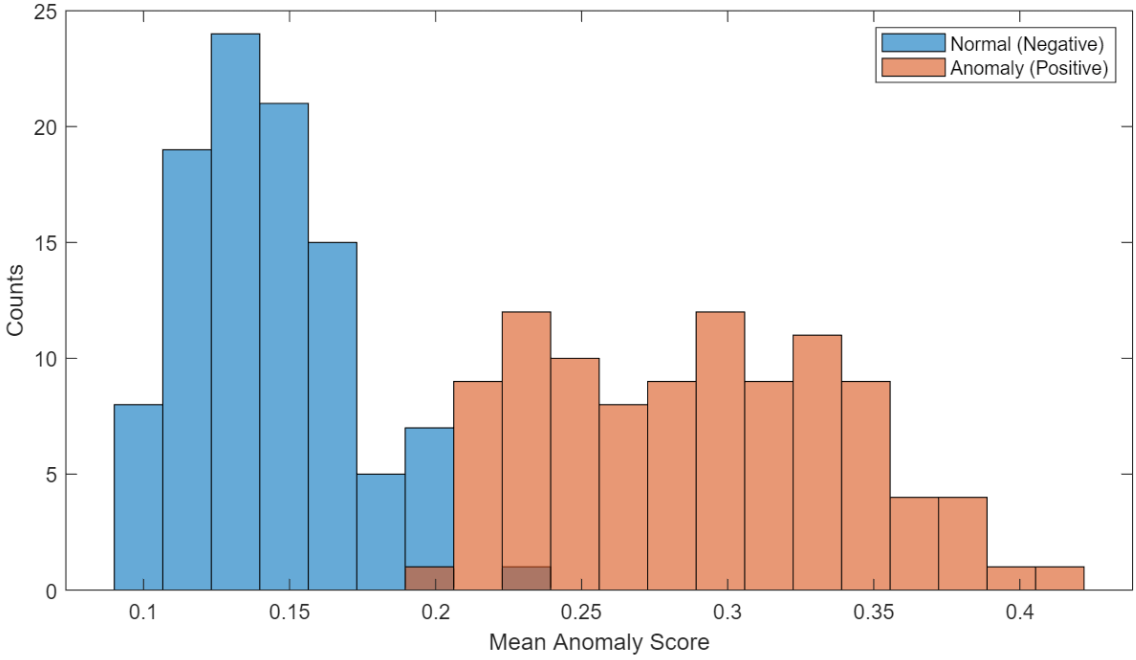
```
labelsCal = tdsCal.UnderlyingDatastores{1}.Labels ~= "Negative";
```

Use the minimum and maximum values of the calibration data set to normalize the mean scores to the range [0, 1].

```
maxScore = max(maxScoresCal,[],"all");
minScore = min(minScoresCal,[],"all");

scoresCal =  mat2gray(meanScoresCal, [minScore maxScore]);
```

Plot a histogram of the mean anomaly scores for the normal and anomaly classes. The distributions are well separated by the model-predicted anomaly score.

```
[~,edges] = histcounts(scoresCal,20);
hGood = histogram(scoresCal(labelsCal==0),edges);
hold on
hBad = histogram(scoresCal(labelsCal==1),edges);
hold off
legend([hGood,hBad],"Normal (Negative)","Anomaly (Positive)")
xlabel("Mean Anomaly Score");
ylabel("Counts");
```

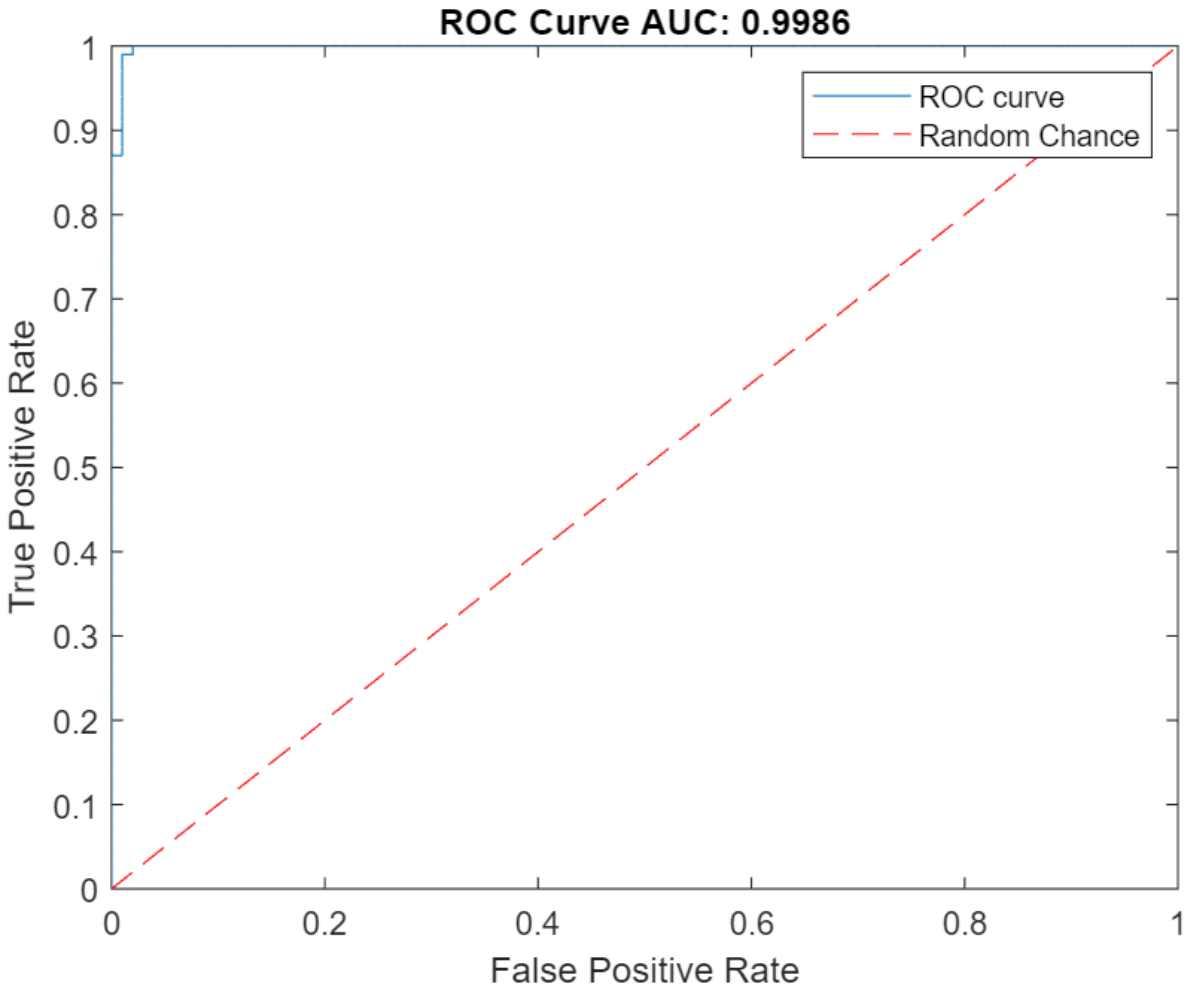


Calculate Threshold Value

Create a receiver operating characteristic (ROC) curve to calculate the anomaly threshold. Each point on the ROC curve represents the false positive rate (*x*-coordinate) and true positive rate (*y*-coordinate) when the calibration set images are classified using a different threshold value. An optimal threshold maximizes the true positive rate and minimizes the false positive rate. Using ROC curves and related metrics allows you to select a threshold based on the tradeoff between false positives and false negatives. These tradeoffs depend on the application-specific implications of misclassifying images as false positives versus false negatives.

Create the ROC curve by using the `perfcurve` (Statistics and Machine Learning Toolbox) function. The solid blue line represents the ROC curve. The red dashed line represents a random classifier corresponding to a 50% success rate. Display the area under the curve (AUC) metric for the calibration set in the title of the figure. A perfect classifier has an ROC curve with a maximum AUC of 1.

```
[xroc,yroc,troc,auc] = perfcurve(labelsCal,scoresCal,true);
figure
lroc = plot(xroc,yroc);
hold on
lchance = plot([0 1],[0 1],"r--");
hold off
xlabel("False Positive Rate")
ylabel("True Positive Rate")
title("ROC Curve AUC: "+auc);
legend([lroc,lchance],"ROC curve","Random Chance")
```



This example uses the maximum Youden Index metric to select the anomaly score threshold from the ROC curve. This corresponds to the threshold value that maximizes the distance between the blue model ROC curve and the red random chance ROC curve.

```
[~,ind] = max(yroc-xroc);
anomalyThreshold = troc(ind)
```

```
anomalyThreshold = 0.2082
```

Evaluate Classification Model

Calculate Anomaly Score Map for Test Set

Calculate feature embedding vectors for the test set images. First, create a `minibatchqueue` (Deep Learning Toolbox) object to manage the mini-batches of test observations. Specify the mini-batch data extraction format as `"SSCB"` (spatial, spatial, channel, batch).

```
testQueue = minibatchqueue(tdsTest, ...
    MiniBatchFormat=["SSCB","CB"], ...
    MiniBatchSize=minibatchSize, ...
    OutputEnvironment="auto");
```

Perform the following steps to compute the anomaly scores for the test set images.

- Extract features of the test images from the same three layers of ResNet-18 used in training.
- Combine the features from the three layers into an overall embedding variable `XTestEmbeddings` by using the `formAlignedEmbeddings` helper function. The helper function is defined at the end of this example.
- Downsample the embedding vectors to the same 100 channel elements used during training, specified by `rIdx`.
- Reshape the embedding vectors into an `H*W`-by-`C`-by-`B` array, where `B` is the number of images in the mini-batch.
- Calculate the Mahalanobis distance between each embedding feature vector and the learned Gaussian distribution by using the `calculateDistance` helper function. The helper function is defined at the end of this example.
- Create an anomaly score map for each image by using the `createAnomalyScoreMap` helper function. The helper function is defined at the end of this example.
- Concatenate the anomaly score maps across mini-batches. The `anomalyScoreMapsTest` variable specifies score maps for all test set images.

```
idx = 1;

XTestImages = [];
anomalyScoreMapsTest = [];

while hasdata(testQueue)
    XTest = next(testQueue);

    XTestFeatures1 = activations(net,extractdata(XTest),feature1LayerName);
    XTestFeatures2 = activations(net,extractdata(XTest),feature2LayerName);
    XTestFeatures3 = activations(net,extractdata(XTest),feature3LayerName);

    XTestEmbeddings = concatenateEmbeddings(XTestFeatures1,XTestFeatures2,XTestFeatures3);

    XTestEmbeddings = XTestEmbeddings(:,:,rIdx,:);
    [H, W, C, B] = size(XTestEmbeddings);
    XTestEmbeddings = reshape(XTestEmbeddings,[H*W C B]);

    distances = calculateDistance(XTestEmbeddings,H,W,B,means,covars);

    anomalyScoreMap = createAnomalyScoreMap(distances,H,W,B,targetImageSize);
    XTestImages = cat(4,XTestImages,gather(XTest));
    anomalyScoreMapsTest = cat(4,anomalyScoreMapsTest,gather(anomalyScoreMap));

    idx = idx+size(XTest,4);
    clear XTestFeatures1 XTestFeatures2 XTestFeatures3 anomalyScoreMap distances XTestEmbeddings
end
```

Classify Test Images

Calculate an overall mean anomaly score for each test image. Normalize the anomaly scores to the same range used to pick the threshold, defined by `minScore` and `maxScore`.

```
scoresTest = squeeze(mean(anomalyScoreMapsTest,[1 2 3]));
scoresTest = mat2gray(scoresTest, [minScore maxScore]);
```

Predict class labels for each test set image by comparing the mean anomaly score map value to the `anomalyThreshold` value.

```
predictedLabels = scoresTest > anomalyThreshold;
```

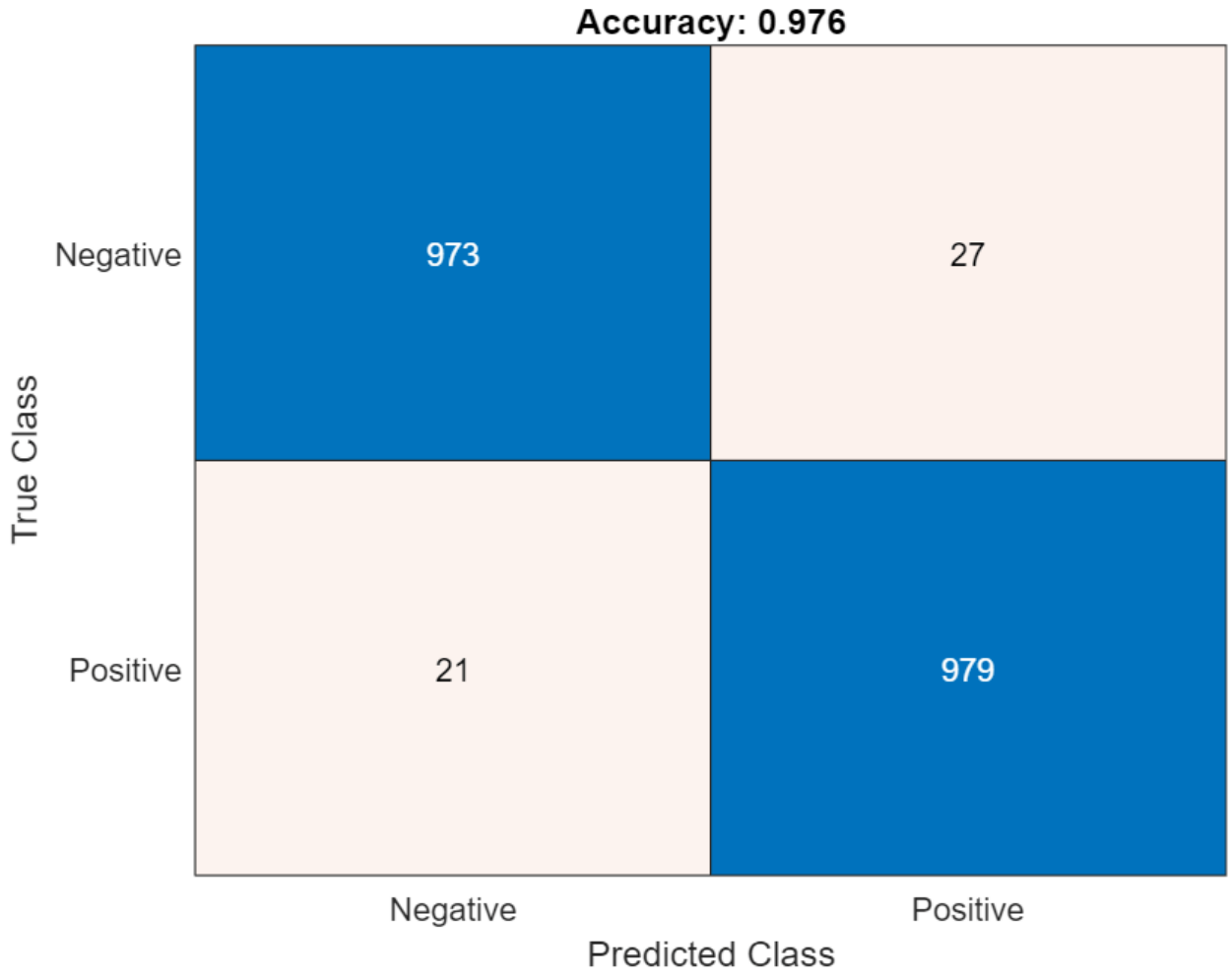
Calculate Classification Accuracy

Assign the known ground truth labels `"Positive"` or `"Negative"` to the test set images.

```
labelsTest = tdsTest.UnderlyingDatastores{1}.Labels ~= "Negative";
```

Calculate the confusion matrix and the classification accuracy for the test set. The classification model in this example is accurate and predicts a small percentage of false positives and false negatives.

```
targetLabels = logical(labelsTest);
M = confusionmat(targetLabels,predictedLabels);
confusionchart(M,["Negative","Positive"])
acc = sum(diag(M)) / sum(M,"all");
title("Accuracy: "+acc);
```



Explain Classification Decisions

You can visualize the anomaly score map predicted by the PaDiM model as a heatmap overlaid on the image. You can use this localization of predicted anomalies to help explain why an image is classified as normal or anomalous. This approach is useful for identifying patterns in false negatives and false positives. You can use these patterns to identify strategies to improve the classifier performance.

Calculate Heatmap Display Range

Instead of scaling the heatmap for each image individually, visualize heatmap data using the same display range for all images in a data set. Doing so yields uniformly cool heatmaps for normal images and warm colors in anomalous regions for anomaly images.

Calculate a display range that reflects the range of anomaly score values observed in the calibration set. Apply the display range for all heatmaps in this example. Set the minimum value of the `displayRange` to 0. Set the maximum value of the display range by calculating the maximum score for each of the 200 calibration images, then selecting the 80th percentile of the maximums. Calculate the percentile value by using the `prctile` function.

```
maxScoresCal = mat2gray(maxScoresCal);
scoreMapRange = [0 prctile(maxScoresCal,80,"all")];
```

View Heatmap of Anomaly

Select an image of a correctly classified anomaly. This result is a true positive classification. Display the image.

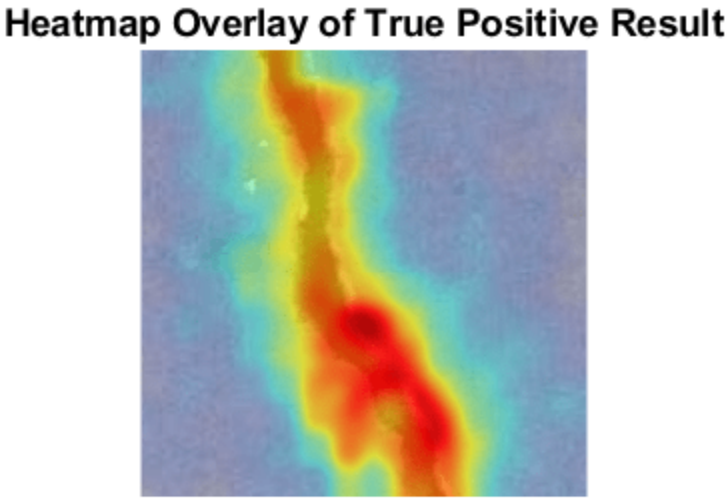
```
idxTruePositive = find(targetLabels & predictedLabels);
dsTruePositive = subset(tdsTest,idxTruePositive);
dataTruePositive = preview(dsTruePositive);
imgTruePositive = dataTruePositive{1};
imshow(imgTruePositive)
title("True Positive Test Image")
```

Obtain an anomaly score map of the true positive anomaly image. Normalize the anomaly scores to the minimum and maximum values of the calibration data set to match the range used to pick the threshold.

```
anomalyTestMapsRescaled = mat2gray(anomalyScoreMapsTest, [minScore maxScore]);
scoreMapTruePositive = anomalyTestMapsRescaled(:,:,1,idxTruePositive(1));
```

Display the heatmap as an overlay over the image by using the `anomalyMapOverlayForConcreteAnomalyDetector` helper function. This function is attached to the example as a supporting file.

```
imshow(anomalyMapOverlayForConcreteAnomalyDetector(imgTruePositive,scoreMapTruePositive,ScoreMapR
title("Heatmap Overlay of True Positive Result")
```



To quantitatively confirm the result, display the mean anomaly score of the true positive test image as predicted by the classifier. The value is greater than the anomaly score threshold.

```
disp("Mean anomaly score of test image: "+scoresTest(idxTruePositive(1)))
```

```
Mean anomaly score of test image: 0.25415
```

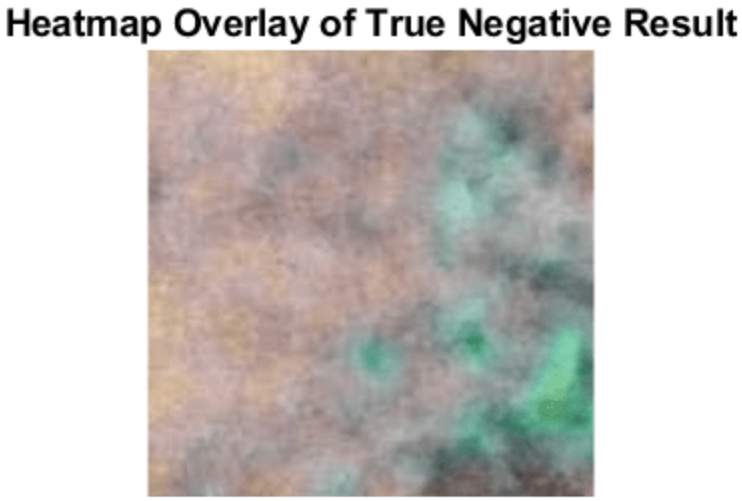
View Heatmap of Normal Image

Select and display an image of a correctly classified normal image. This result is a true negative classification.

```
idxTrueNegative = find(~(targetLabels | predictedLabels));
dsTrueNegative = subset(tdsTest,idxTrueNegative);
dataTrueNegative = preview(dsTrueNegative);
imgTrueNegative = dataTrueNegative{1};
imshow(imgTrueNegative)
title("True Negative Test Image")
```

Obtain a heatmap of the normal image. Display the heatmap as an overlay over the image by using the `anomalyMapOverlayForConcreteAnomalyDetector` helper function. This function is attached to the example as a supporting file. Many true negative test images, such as this test image, have either small anomaly scores across the entire image or large anomaly scores in a localized portion of the image.

```
scoreMapTrueNegative = anomalyTestMapsRescaled(:,:,1,idxTrueNegative(1));
imshow(anomalyMapOverlayForConcreteAnomalyDetector(imgTrueNegative,scoreMapTrueNegative,ScoreMapR
title("Heatmap Overlay of True Negative Result")
```



To quantitatively confirm the result, display the mean anomaly score of the true positive test image as predicted by the classifier. The value is less than the anomaly score threshold.

```
disp("Mean anomaly score of test image: "+scoresTest(idxTrueNegative(1)))
```

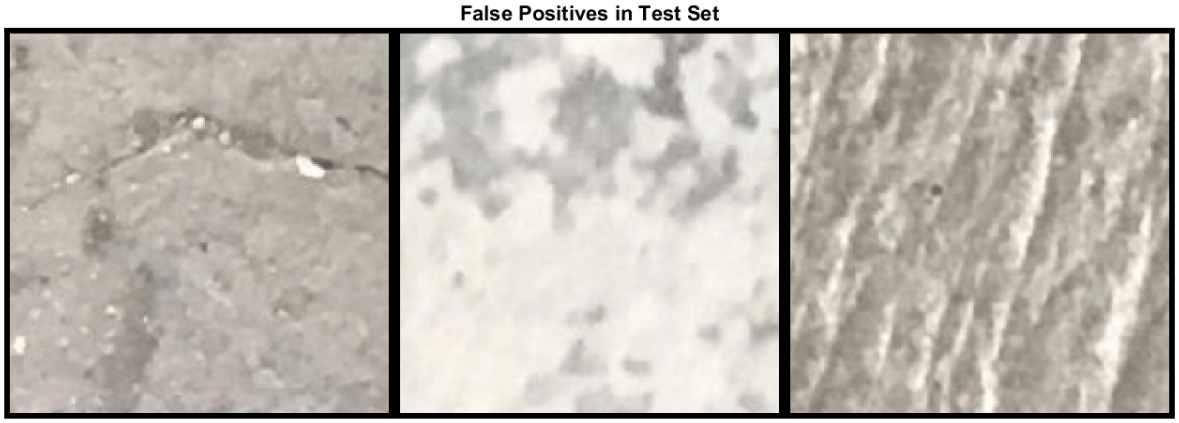
```
Mean anomaly score of test image: 0.12314
```

View Heatmaps of False Positive Images

False positives are images without crack anomalies that the network classifies as anomalous. Use the explanation from the PaDiM model to gain insight into the misclassifications.

Find false positive images from the test set. Display three false positive images as a montage.

```
idxFalsePositive = find(~targetLabels & predictedLabels);
dataFalsePositive = readall(subset(tdsTest,idxFalsePositive));
numelFalsePositive = length(idxFalsePositive);
numImages = min(numelFalsePositive,3);
if numelFalsePositive>0
    montage(dataFalsePositive(1:numImages,1),Size=[1,numImages],BorderSize=10);
    title("False Positives in Test Set")
end
```

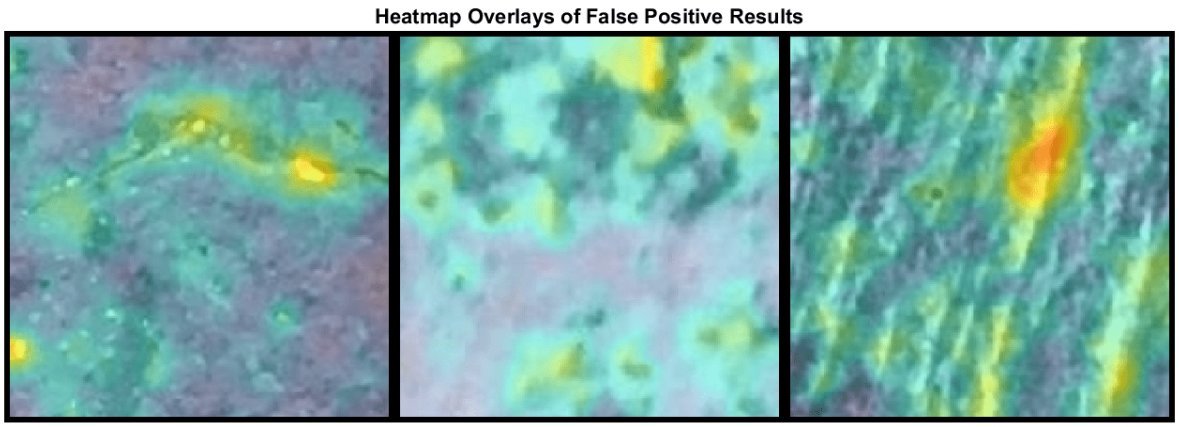


Obtain heatmaps of the false positive images.

```
hmapOverlay = cell(1,numImages);
for idx = 1:numImages
    img = dataFalsePositive{idx,1};
    scoreMapFalsePositive = anomalyTestMapsRescaled(:,:,1,idxFalsePositive(idx));
    hmapOverlay{idx} = anomalyMapOverlayForConcreteAnomalyDetector(img,scoreMapFalsePositive,Sco
end
```

Display the heatmap overlays as a montage. The false positive images show features such as rocks that have similar visual characteristics to cracks. The anomaly scores are high in these localized regions. However, the training data set only labels images with cracks as anomalous, so the ground truth label for these images is `Negative`. Training a classifier that recognizes rocks and other non-crack defects as anomalous requires training data with non-crack defects labeled as anomalous.

```
if numelFalsePositive>0
    montage(hmapOverlay,Size=[1,numImages],BorderSize=10)
    title("Heatmap Overlays of False Positive Results")
end
```



Display the mean anomaly scores of the false positive test images as predicted by the PaDiM model. The mean scores are greater than the anomaly score threshold, resulting in misclassifications.

```
disp("Mean anomaly scores:"); scoresTest(idxFalsePositive(1:numImages))
```

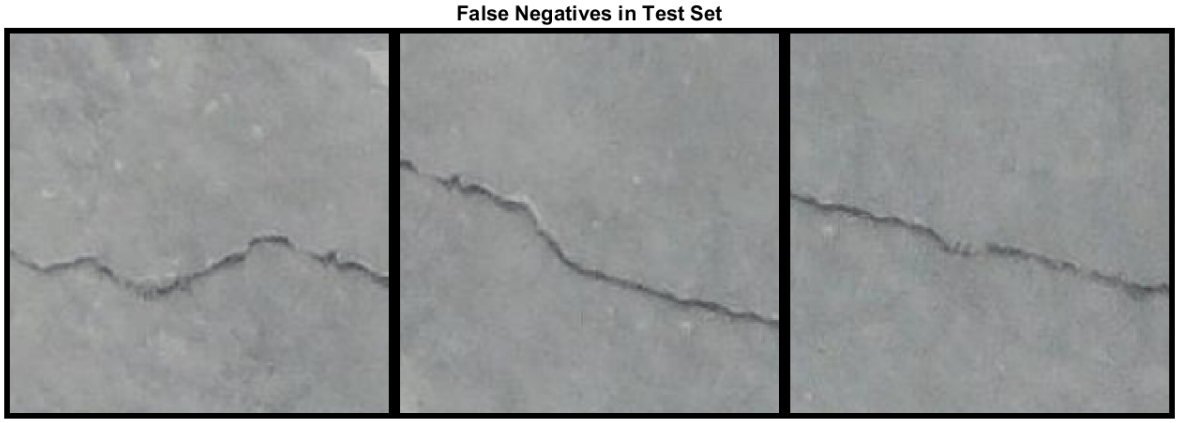
```
Mean anomaly scores:

ans = 3×1

    0.2125
    0.2395
    0.2651
```

View Heatmaps of False Negative Images

False negatives are images with crack anomalies that the network classifies as normal. Use the explanation from the PaDiM model to gain insights into the misclassifications.

Find any false negative images from the test set. Display three false negative images as a montage.

```
idxFalseNegative = find(targetLabels & ~predictedLabels);
dataFalseNegative = readall(subset(tdsTest,idxFalseNegative));
numelFalseNegative = length(idxFalseNegative);
numImages = min(numelFalseNegative,3);
if numelFalseNegative>0
    montage(dataFalseNegative(1:numImages,1),Size=[1,numImages],BorderSize=10);
    title("False Negatives in Test Set")
end
```

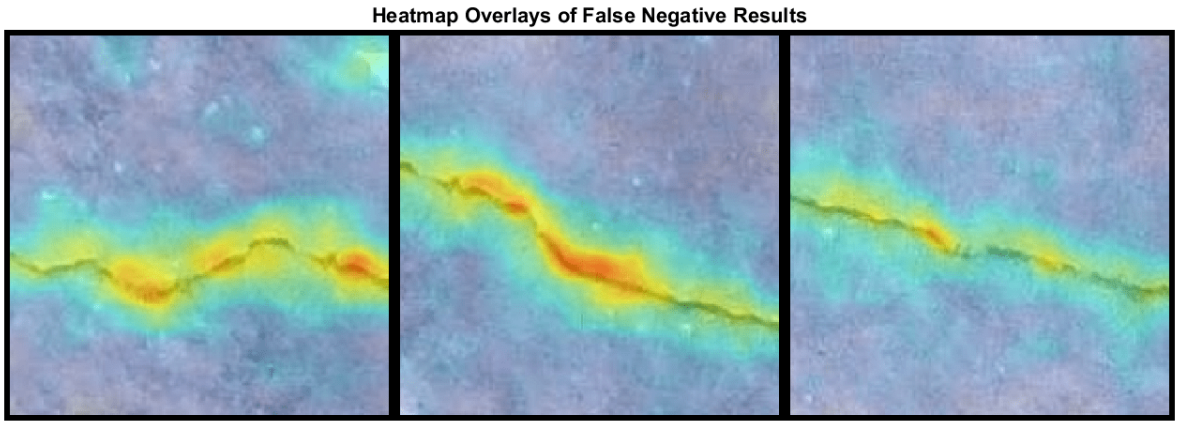


Obtain heatmaps of the false negative images.

```
hmapOverlay = cell(1,numImages);
for idx = 1:numImages
    img = dataFalseNegative{idx,1};
    scoreMapFalseNegative = anomalyTestMapsRescaled(:,:,1,idxFalseNegative(idx));
    hmapOverlay{idx} = anomalyMapOverlayForConcreteAnomalyDetector(img,scoreMapFalseNegative,Sco
end
```

Display the heatmap overlays as a montage. The PaDiM model predicts large anomaly scores around cracks, as expected.

```
if numelFalseNegative>0
    montage(hmapOverlay,Size=[1,numImages],BorderSize=10)
    title("Heatmap Overlays of False Negative Results")
end
```



Display the mean anomaly scores of the false negative test images as predicted by the PaDiM model. The mean scores are less than the anomaly score threshold, resulting in misclassifications.

```
disp("Mean anomaly scores:"); scoresTest(idxFalsePositive(1:numImages))
```

```
Mean anomaly scores:

ans = 3×1

    0.2125
    0.2395
    0.2651
```

Supporting Functions

The `concatenateEmbeddings` helper function combines features extracted from three layers of ResNet-18 into one feature embedding vector. The features from the second and third blocks of ResNet-18 are resized to match the spatial resolution of the first block.

```
function XEmbeddings = concatenateEmbeddings(XFeatures1,XFeatures2,XFeatures3)
    XFeatures2Resize = imresize(XFeatures2,2,"nearest");
    XFeatures3Resize = imresize(XFeatures3,4,"nearest");
    XEmbeddings = cat(3,XFeatures1,XFeatures2Resize,XFeatures3Resize);
end
```

The `calculateDistance` helper function calculates the Mahalanobis distance between each embedding feature vector specified by `XEmbeddings` and the learned Gaussian distribution for the corresponding patch with mean specified by `means` and covariance matrix specified by `covars`.

```
function distances = calculateDistance(XEmbeddings,H,W,B,means,covars)
    distances = zeros([H*W 1 B]);
```

```
    for dIdx = 1:H*W
        distances(dIdx,1,:) = pdist2((squeeze(means(dIdx,:))),(squeeze(XEmbeddings(dIdx,:,:))'),'
    end
end
```

The `createAnomalyScoreMap` helper function creates an anomaly score map for each image with embeddings vectors specified by `XEmbeddings`. The `createAnomalyScoreMap` function reshapes and resizes the anomaly score map to match the size and resolution of the original input images.

```
function anomalyScoreMap = createAnomalyScoreMap(distances,H,W,B,targetImageSize)
    anomalyScoreMap = reshape(distances,[H W 1 B]);
    anomalyScoreMap = imresize(anomalyScoreMap,targetImageSize,"bilinear");
    for mIdx = 1:size(anomalyScoreMap,4)
        anomalyScoreMap(:,:,1,mIdx) = imgaussfilt(anomalyScoreMap(:,:,1,mIdx),4,FilterSize=33);
    end
end
```

References

[1] Defard, Thomas, Aleksandr Setkov, Angelique Loesch, and Romaric Audigier. “PaDiM: A Patch Distribution Modeling Framework for Anomaly Detection and Localization.” In Pattern Recognition. ICPR International Workshops and Challenges, 475–89. Lecture Notes in Computer Science. Cham, Switzerland: Springer International Publishing, 2021. https://doi.org/10.1007/978-3-030-68799-1_35.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition.” In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–78. Las Vegas, NV, USA: IEEE, 2016. https://doi.org/10.1109/CVPR.2016.90.

[3] *ImageNet*. https://www.image-net.org.

See Also

`imageDatastore` | `activations` | `resnet18` | `perfcurve` | `confusionmat` | `confusionchart`

Related Examples

- “Detect Image Anomalies Using Explainable One-Class Classification Neural Network” on page 3-40
- “Classify Defects on Wafer Maps Using Deep Learning” on page 3-24

Train Object Detectors in Experiment Manager

This example shows how to use Experiment Manager app to find optimal training options for object detectors by sweeping through a range of hyperparameter values.

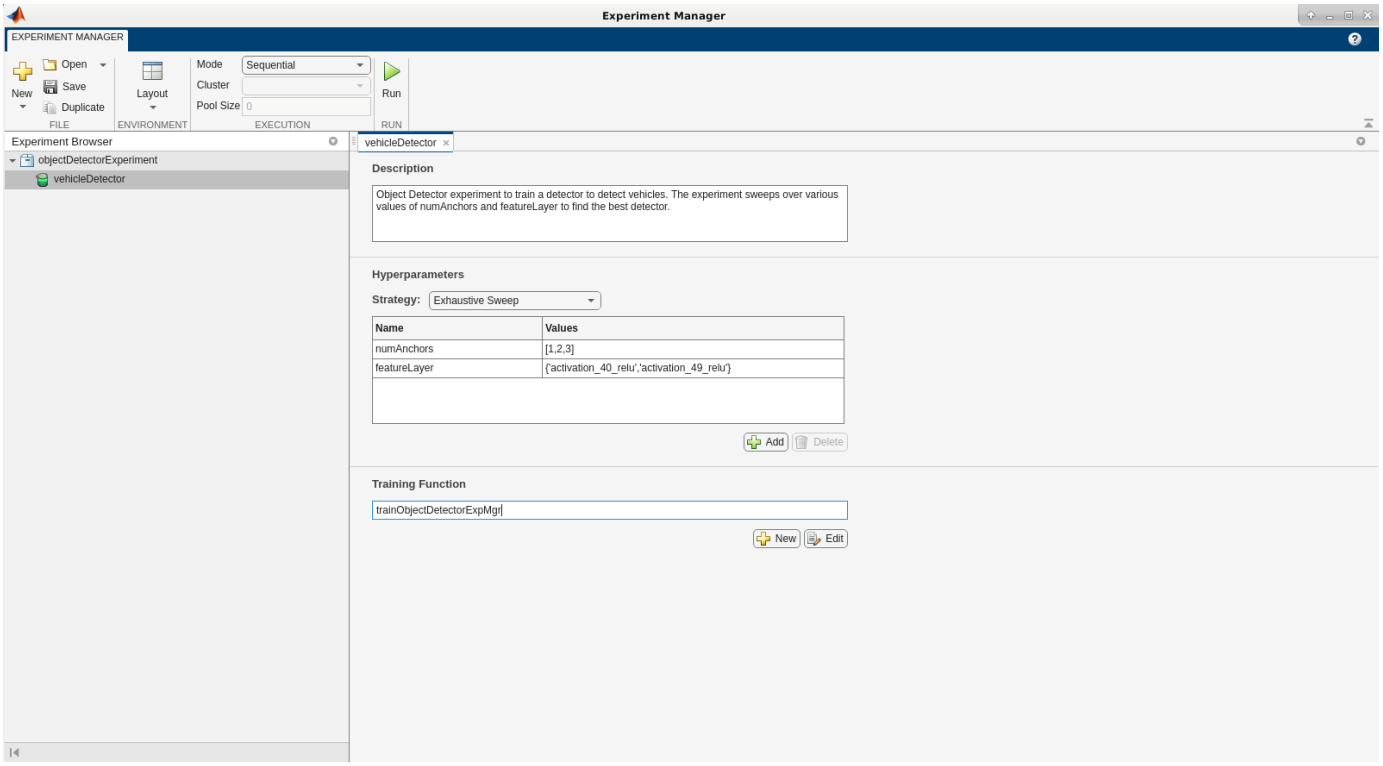
Overview

The Experiment Manager app enables you to create deep learning experiments to train object detectors under multiple initial conditions and compare the results. In this example, you will use the Experiment Manager app to train a YOLO v2 object detector to detect vehicles from traffic images. You will sweep over the number-of-anchors and different choices of feature extraction layer to obtain the best performing object detector. Experiment Manager trains the object detector using every combination of hyperparameter values specified in the hyperparameter table. Note that in this experiment you will run trials over different values of `numAnchors` and `featureLayer` for simplicity. To find the optimum object detector, specify experiment trials to sweep across additional hyperparameters such as learning rate, mini-batch size, and image size.

For more information about the Experiment Manager see, Experiment Manager (Deep Learning Toolbox).

Open Experiment

First, open the example. Experiment Manager loads the project with a custom object detector experiment that you can inspect and run.



The Hyperparameter section allows you to set the hyperparameters you wish to sweep over.

The Training Function section allows you to specify a custom training script to use for the experiment. This example uses `vehicleDetectorExperiment` function which loads in the data,

model, and performs training. Details of the script are described in this section. The complete function is listed under the Supporting Functions section.

Load in the data using the utility function `splitVehicleData` . This example uses a small vehicle dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona, and used with permission. Each image contains one or two labeled instances of a vehicle.

```
output.trainedNet = [];
output.ap = [];
output.executionEnvironment = "auto";
[trainingData,validationData,testData] = splitVehicleData;
```

Apply augmentations and preprocessing on the training dataset. For the validation and test dataset only preprocessing is needed.

```
inputSize = [224 224];
augmentedTrainingData = transform(trainingData,@augmentData);
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))
preprocessedValidationData = transform(validationData,@(data)preprocessData(data,inputSize));
preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
```

For this example, you will sweep over the number of anchor boxes for training. “Anchor Boxes for Object Detection” on page 17-32 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. Multiple anchor boxes enable the object detector to detect objects of different sizes. The shape, scale, and number of anchor boxes impact the efficiency and accuracy of the detectors. A large number of anchor boxes decrease the runtime performance of the detector. The `estimateAnchorBoxes` function uses the hyperparameter (`params.numAnchors`) passed by the Experiment Manager during each trial. This changes the number of anchor boxes estimated during each trial. For more information about the anchor box estimation see, “Estimate Anchor Boxes From Training Data” on page 3-319.

```
aboxes = estimateAnchorBoxes(preprocessedTrainingData,params.numAnchors);
```

Set the training options.

```
opts = trainingOptions('rmsprop',...
    'InitialLearnRate',0.0001,...
    'MiniBatchSize',16,...
    'MaxEpochs',20,...
    'LearnRateSchedule','piecewise',...
    'LearnRateDropPeriod',3,...
    'VerboseFrequency',30, ...
    'L2Regularization',0.001);
```

Create a YOLOv2 object detector using `yolov2Layers` with Resnet50 backbone. For this example, you will also sweep over different feature extraction layers. Different feature extraction layers correspond to different amounts of downsampling. There 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. Set `featureLayer` to `params.featureLayer` to accept the input from Experiment Manager. You can visualize the network to identify different feature layers using `analyzeNetwork` or D|eepNetworkDesigner| from Deep Learning Toolbox&trade.

```
numClasses = 1;
inputSize = [224 224 3];
```

```
network = resnet50();
featureLayer = params.featureLayer;
lgraph = yolov2Layers(inputSize,numClasses,aboxes,network,featureLayer);
```

The network will be trained using the `trainYOLOv2ObjectDetector` function with the ExperimentMonitor name-value pair set to `monitor`. Setting this name value-pair allows the trainer to feed the training statistics back to Experiment Monitor at regular intervals.

```
detector = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,...
          opts,'ExperimentMonitor',monitor);
```

Assess the trained object detector on the validation set by computing the average precision score. Precision is a ratio of true positive instances to all positive instances of objects in the detector, based on the ground truth.

```
results = detect(detector,preprocessedTestData,'MiniBatchSize',4);
[ap,~,~] = evaluateDetectionPrecision(results,preprocessedTestData);
```

Package the trained detector and average precision score into the output struct.

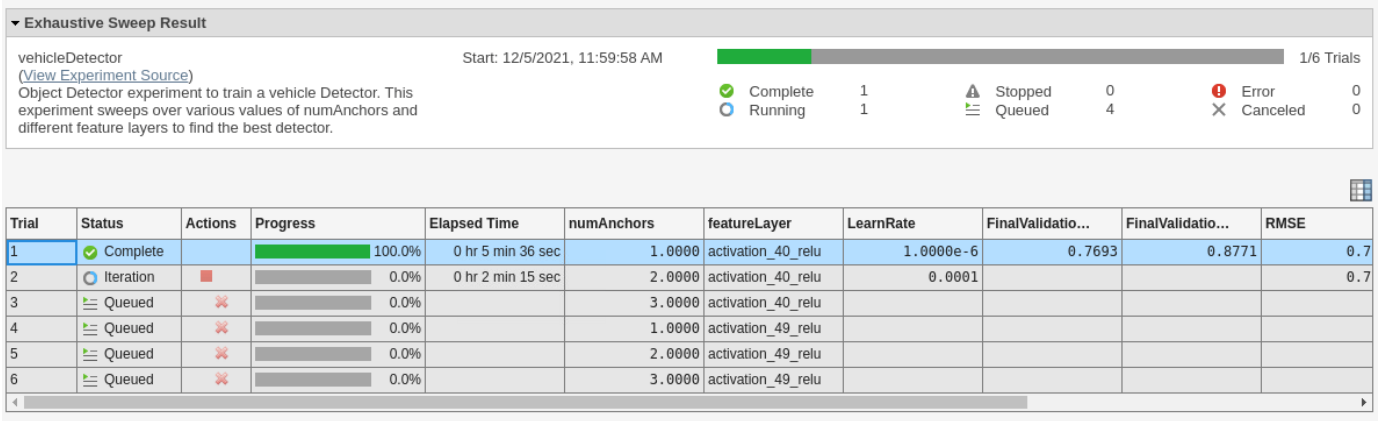
```
output.trainedNet = detector;
output.ap = ap;
```

Run Experiment

Click the Run button on the Experiment Manager toolstrip to start the training trials.

When you run the experiment, Experiment Manager trains the network defined by the training function six times. Each trial uses a unique combination of `numAnchors` and `featureLayer` specified in the hyperparameter table. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbo&trade, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs.

A table of results displays the training loss for each trial.



Trial	Status	Actions	Progress	Elapsed Time	numAnchors	featureLayer	LearnRate	FinalValidatio...	FinalValidatio...	RMSE
1	Complete		100.0%	0 hr 5 min 36 sec	1.0000	activation_40_relu	1.0000e-6	0.7693	0.8771	0.7
2	Iteration		0.0%	0 hr 2 min 15 sec	2.0000	activation_40_relu	0.0001			0.7
3	Queued		0.0%		3.0000	activation_40_relu				
4	Queued		0.0%		1.0000	activation_49_relu				
5	Queued		0.0%		2.0000	activation_49_relu				
6	Queued		0.0%		3.0000	activation_49_relu				

Export results

Export the best-trained detector to the workspace:

- Select the trial with the lowest loss score.

- On the Experiment Manager toolstrip, click Export.
- In the dialog window, enter the name of a workspace variable for the exported training output. The default name is trainingOutput.

Visualize the results of the trained detector by calling the `runDetectorOnTestImage` function.

```
runDetectorOnTestImage(trainingOutput)
```

Appendix 1: Training Function

The `vehicleDetectorExperiment` function specifies the training data, network architecture, training options, and training procedure used by the experiment.

This function takes in two arguments,

- `params` is a structure with fields from the Experiment Manager hyperparameter table.
- `monitor` is an `experiments.Monitor` (Deep Learning Toolbox) object that you can use to track the progress of the training, update information fields in the results table, record values of the metrics used by the training, and produce training plots.

The output of this function is a struct that contains the trained detector network, the execution environment, and Average Precision metrics for the trained network. Experiment Manager saves this output, so you can export it to the MATLAB workspace when the training is complete.

```
function output = trainObjectDetectorExpMgr(params,monitor)
    output.trainedNet = [];
    output.ap = [];
    output.executionEnvironment = "auto";

    % Load data and split it into training, validation and test sets
    [trainingData,validationData,testData] = splitVehicleData;

    % Augment and preprocess the data
    inputSize = [224 224];
    augmentedTrainingData = transform(trainingData,@augmentData);
    preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSi
    preprocessedValidationData = transform(validationData,@(data)preprocessData(data,inputSize))
    preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
```

```
    % Setup the training options
    opts = trainingOptions('rmsprop',...
        'InitialLearnRate',0.001,...
        'MiniBatchSize',16,...
        'MaxEpochs',20,...
        'LearnRateSchedule','piecewise',...
        'LearnRateDropPeriod',5,...
        'VerboseFrequency',30, ...
        'L2Regularization',0.001,...
        'ValidationData',preprocessedValidationData,...
        'ValidationFrequency',50);

    % Construct the YOLO v2 detector
    numClasses = 1;
    inputSize = [224 224 3];
    network = resnet50();
    featureLayer = params.featureLayer;
    % Estimate anchor boxes by using numAnchors parameter from the Experiment Manager
    aboxes = estimateAnchorBoxes(preprocessedTrainingData,params.numAnchors);
    lgraph = yolov2Layers(inputSize,numClasses,aboxes,network,featureLayer);
    % Train YOLOv2 detector
    [detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,opts,'ExperimentM

    % Capture Average Precision result with the output
    results = detect(detector,preprocessedTestData,'MiniBatchSize',4);
    [ap,~,~] = evaluateDetectionPrecision(results,preprocessedTestData);

    output.trainedNet = detector;
    output.ap = ap;
    output.info = info;
end
```

Appendix 2: Data Preprocessing Functions

`augmentData` function

The `augmentData` function returns augmented images for training.

```
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);
```

```
    % Sanitize box data, if needed.
    A{2} = helperSanitizeBoxes(A{2}, sz);
    % 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
```

preprocessData function

The `preprocessData` function rescales the images and the bounding boxes according to the target size.

```
function data = preprocessData(data,targetSize)
    % Resize image and bounding boxes to the targetSize.
    sz = size(data{1},[1 2]);
    scale = targetSize(1:2)./sz;
    data{1} = imresize(data{1},targetSize(1:2));
    % Sanitize box data, if needed.
    data{2} = helperSanitizeBoxes(data{2},sz);
    % Resize boxes to new image size.
    data{2} = bboxresize(data{2},scale);
end
```

```
% helperSanitizeBoxes Sanitize box data.
% This example helper is used to clean up invalid bounding box data. Boxes
% with values <= 0 are removed and fractional values are rounded to
% integers.
%
% If none of the boxes are valid, this function passes the data through to
% enable downstream processing to issue proper errors.

function boxes = helperSanitizeBoxes(boxes, ~)
    persistent hasInvalidBoxes
    valid = all(boxes > 0, 2);
    if any(valid)
        if ~all(valid) && isempty(hasInvalidBoxes)
            % Issue one-time warning about removing invalid boxes.
            hasInvalidBoxes = true;
            warning('Removing ground truth bouding box data with values <= 0.')
        end
        boxes = boxes(valid,:);
    end
end
```

Appendix 3: Data Loading Function

The `splitVehicleData` function loads the data and splits it into training, validation, and test datastores.

```
function [dsTrain,dsVal,dsTest] = splitVehicleData()
    outputDir = fullfile(tempdir,'vehicleImages');

    if ~exist(outputDir,'dir')
        % Unzip images and load the lables
        unzip('vehicleDatasetImages.zip', fullfile(tempdir));
    end
    data = load('vehicleDatasetGroundTruth.mat');
    vehicleDataset = data.vehicleDataset;

    % Load the list of image files
    vehicleDataset.imageFilename = fullfile(tempdir,vehicleDataset.imageFilename);
    rng(0);
    shuffledIndices = randperm(height(vehicleDataset));
    idx = floor(0.6 * length(shuffledIndices) );

    % Create a training, validation and test indices
    trainingIdx = 1:idx;
    validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
    testIdx = validationIdx(end)+1 : length(shuffledIndices);

    % Load data using imageDatastore and boxLabelDatastore
    imds = imageDatastore(vehicleDataset{:,'imageFilename'});
    blds = boxLabelDatastore(vehicleDataset(:,'vehicle'));
    allData = combine(imds,blds);
    dsTrain = subset(allData,trainingIdx);
    dsVal = subset(allData,validationIdx);
    dsTest = subset(allData,testIdx);
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.

Activity Recognition Using R(2+1)D Video Classification

This example first shows how to perform activity recognition using a pretrained R(2+1)D [1] on page 3-0 convolutional neural network based video classifier and then shows how to use transfer learning to train such a video classifier using video data.

Overview

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 inaccurate ground truth data for video data sets, the variety of gestures that actors in a video can perform, the heavily class imbalanced datasets, and the large amount of data required to train a robust classifier from scratch. Deep learning techniques, such as R(2+1)D [1] on page 3-0 and SlowFast [2 on page 3-0] have shown improved performance on smaller datasets using transfer learning with networks pretrained on large video activity recognition datasets, such as Kinetics-400 [4 on page 3-0].

Note: This example requires the Computer Vision Toolbox™ Model for R(2+1)D Video Classification. You can install the Computer Vision Toolbox Model for R(2+1)D Video Classification from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Perform Activity Recognition Using a Pretrained R(2+1)D Video Classifier

Download the pretrained R(2+1)D video classifier along with a video file on which to perform activity recognition. The size of the downloaded zip file is around 112 MB.

```
pretrainedFolder = fullfile(tempdir,"hmdb51","pretrained","r2plus1d");
if ~isfolder(pretrainedFolder)
    mkdir(pretrainedFolder);
end
zipFile = 'activityRecognition-R2Plus1D-HMDB51-21b.zip';

if ~isfile(fullfile(pretrainedFolder,zipFile))
    disp('Downloading the pretrained network...')
    downloadURL = ['https://ssd.mathworks.com/supportfiles/vision/data/' zipFile];
    zipFile = fullfile(pretrainedFolder,zipFile);
    websave(zipFile,downloadURL);
    unzip(zipFile,pretrainedFolder);
    disp('Downloaded.')
end
```

Load the pretrained R(2+1)D video classifier.

```
pretrainedDataFile = fullfile(pretrainedFolder,'r2plus1d-FiveClasses-hmdb51.mat');
pretrained = load(pretrainedDataFile);
r2plus1dPretrained = pretrained.data.r2plus1d;
```

Display the class label names of the pretrained video classifier.

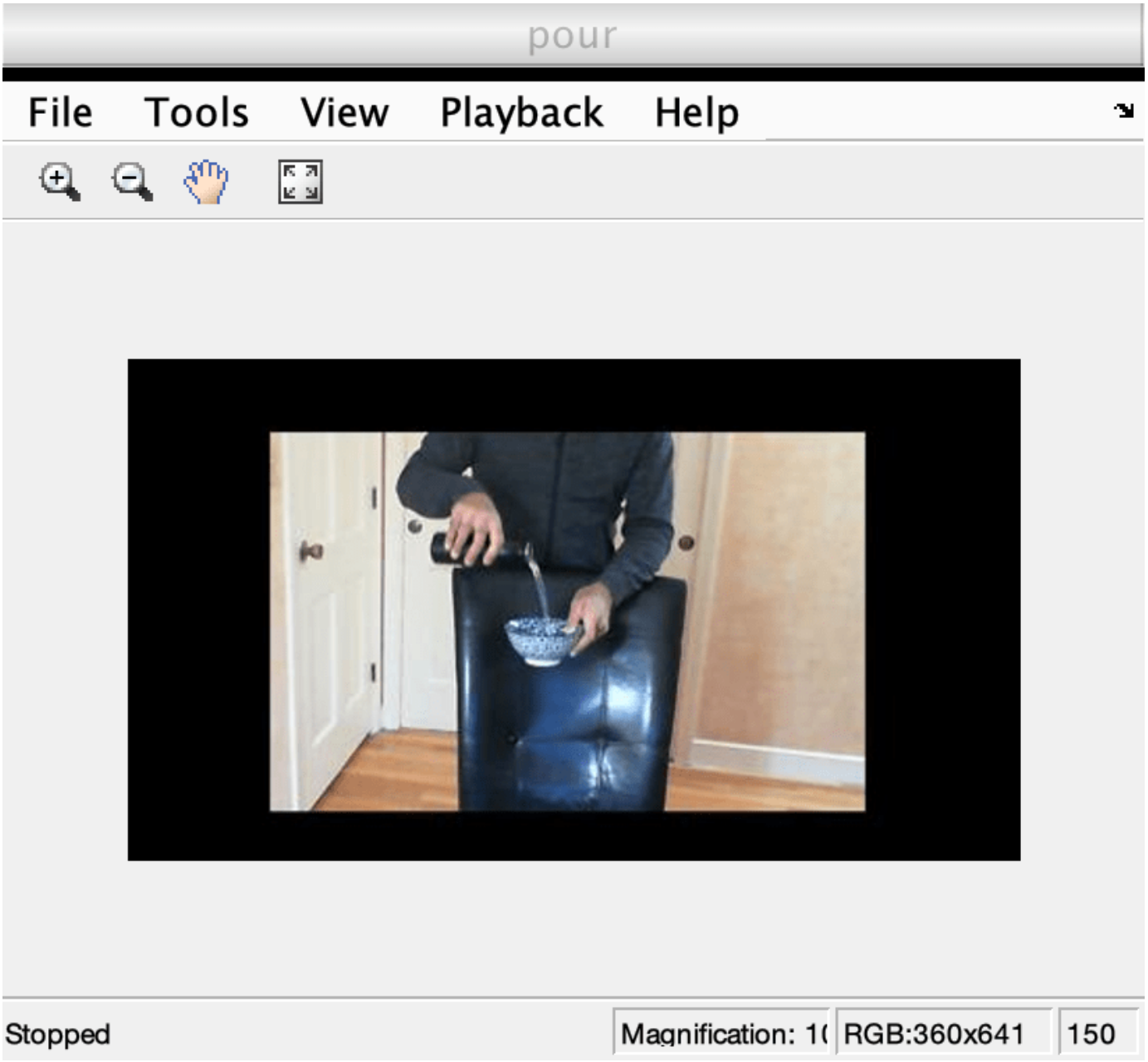
```
classes = r2plus1dPretrained.Classes
```

```
classes = 5×1 categorical
     kiss
```

```
    laugh
    pick
    pour
    pushup
```

Read and display the video `pour.avi` using `VideoReader` and `vision.VideoPlayer`.

```
videoFilename = fullfile(pretrainedFolder, "pour.avi");

videoReader = VideoReader(videoFilename);
videoPlayer = vision.VideoPlayer;
videoPlayer.Name = "pour";

while hasFrame(videoReader)
   frame = readFrame(videoReader);
   % Resize the frame for display.
   frame = imresize(frame, 1.5);
   step(videoPlayer,frame);
end
release(videoPlayer);
```



Choose 10 randomly selected video sequences to classify the video, to uniformly cover the entirety of the file to find the action class that is predominant in the video.

```
numSequences = 10;
```

Classify the video file using the `classifyVideoFile` function.

```
[actionLabel,score] = classifyVideoFile(r2plus1dPretrained, videoFilename, "NumSequences", numSe
```

```
actionLabel = categorical
     pour
```

```
score = single
    0.9727
```

Train a Video Classifier for Gesture Recognition

This section of the example shows how the video classifier shown above is trained using transfer learning. Set the `doTraining` variable to `false` to use the pretrained video classifier without having to wait for training to complete. Alternatively, if you want to train the video classifier, set the `doTraining` variable to `true`.

```
doTraining = false;
```

Download Training and Validation 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 = string(classes);
end
dataFolder = fullfile(downloadFolder, "hmdb51_org");
```

Split the data set into a training set for training the classifier, and a test set for evaluating the classifier. Use 80% of the data for the training set and the rest for the test set. Use `folders2labels` and `splitlabels` to create label information from folders and split the data based on each label into training and test data sets by randomly selecting a proportion of files from each label.

```
[labels,files] = folders2labels(fullfile(dataFolder,classes),...
    "IncludeSubfolders",true,...
    "FileExtensions",'.avi');

indices = splitlabels(labels,0.8,'randomized');

trainFilenames = files(indices{1});
testFilenames  = files(indices{2});
```

Load Dataset

This example uses a datastore to read the videos sequences and the corresponding labels from the video files.

Specify the number of video frames the datastore should be configured to output for each time data is read from the datastore.

```
numFrames = 32;
```

A value of 32 is used here to balance memory usage and classification time. Common values to consider are 8, 16, 32, 64, or 128. Using more frames helps capture additional temporal information, but requires more memory. Empirical analysis is required to determine the optimal number of frames. You might need to increase or decrease this value depending on your system resources.

Next, specify the height and width of the frames the datastore should be configured to output. The datastore automatically resizes the raw video frames to the specified size to enable batch processing of multiple video sequences.

```
frameSize = [112,112];
```

A value of [112 112] is used to capture longer temporal relationships in the video scene which help classify gestures with long time durations. Common values for the size are [112 112], [224 224], or [256 256]. Smaller sizes enable the use of more video frames at the cost of memory usage, processing time, and spatial resolution. 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`. As with the number of frames, empirical analysis is required to determine the optimal values.

Specify the number of channels as 3 for the RGB input data.

```
numChannels = 3;
```

Use the helper function, `createFileDatastore`, to configure two `FileDatastore` objects for loading the data, one for training and another for validation. The helper function is listed at the end of this example.

```
isDataForTraining = true;
dsTrain = createFileDatastore(trainFilenames,numFrames,numChannels,classes,isDataForTraining);

isDataForTraining = false;
dsVal = createFileDatastore(testFilenames,numFrames,numChannels,classes,isDataForTraining);
```

Configure R(2+1)D Video Classifier for Transfer Learning

In this example, you create a R(2+1)D video classifier based on the ResNet-3D architecture with 18 Spatio-Temporal residual layers, a 3D Convolution Neural Network Video Classifier pretrained on the Kinetics-400 dataset [4 on page 3-0].

Specify ResNet-3D with 18 Spatio-Temporal layers as the base network architecture for the R(2+1)D classifier.

```
baseNetwork = "resnet-3d-18";
```

Specify the input size for the R(2+1)D Video Classifier.

```
inputSize = [frameSize, numChannels, numFrames];
```

Create a R(2+1)D Video Classifier by specifying the classes for the HMDB51 dataset and the network input size.

```
r2plus1d = r2plus1dVideoClassifier(baseNetwork,string(classes),"InputSize",inputSize);
```

Specify a model name for the video classifier.

```
r2plus1d.ModelName = "R(2+1)D Activity Recognizer";
```

Augment and Preprocess Training Data

Data augmentation provides a way to use limited data sets for training. Augmentation on video data must be the same for a collection of frames based on the network input size. Minor changes, such as translation, cropping, or transforming an image, provide, new, distinct, and unique images that you can use to train a robust video classifier. Datastores are a convenient way to read and augment collections of data. Augment the training video data by using the `augmentVideo` supporting function, defined at the end of this example.

```
dsTrain = transform(dsTrain, @augmentVideo);
```

Preprocess the training video data to resize to the R(2+1)D Video Classifier input size, by using the `preprocessVideoClips`, defined at the end of this example. Specify the `InputNormalizationStatistics` property of the video classifier and input size to the preprocessing function as field values in a struct, `preprocessInfo`. The `InputNormalizationStatistics` property is used to rescale the video frames between 0 and 1, and then normalize the rescaled data using mean and standard deviation. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` to random crop or center crop the input data to the input size of the video classifier. 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.

```
preprocessInfo.Statistics = r2plus1d.InputNormalizationStatistics;
preprocessInfo.InputSize = inputSize;
preprocessInfo.SizingOption = "resize";
dsTrain = transform(dsTrain, @(data)preprocessVideoClips(data, preprocessInfo));
dsVal = transform(dsVal, @(data)preprocessVideoClips(data, preprocessInfo));
```

Define Model Loss Function

The `modelLoss` function, listed at the end of this example, takes as input the R(2+1)D Video Classifier `r2plus1d`, a mini-batch of input data `dlRGB`, and a mini-batch of ground truth label data `dlT`. The function returns the training loss value, the gradients of the loss with respect to the learnable parameters of the classifier, and the mini-batch accuracy of the classifier.

The loss is calculated by computing the cross-entropy loss of the predictions from video classifier. The output predictions of the network are probabilities between 0 and 1 for each of the classes.

$$predictions = forward(r2plus1d, dlRGB);$$

$$loss = crossentropy(predictions)$$

The accuracy of the classifier is calculated by comparing the classifier `predictions` to the ground truth label of the inputs, `dlT`.

Specify Training Options

Train with a mini-batch size of 5 for 900 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:

- A minimum learning rate of 1e-4.
- A maximum learning rate of 1e-3.
- Cosine number of iterations of 200, 300, and 400, 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:

- 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. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

```
params.Classes = classes;
params.MiniBatchSize = 5;
params.NumIterations = 900;
params.SaveBestAfterIteration = 600;
params.CosineNumIterations = [200, 300, 400];
params.MinLearningRate = 1e-4;
params.MaxLearningRate = 1e-3;
params.Momentum = 0.9;
params.Velocity = [];
params.L2Regularization = 0.0005;
params.ProgressPlot = false;
params.Verbose = true;
params.ValidationData = dsVal;
params.DispatchInBackground = false;
params.NumWorkers = 12;
```

Train R(2+1)D Video Classifier

Train the R(2+1)D video classifier using the video data.

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`.

- 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 video data and the labels to `dlarray` objects with the underlying type single.
- To enable processing the time dimension of the the video data using the R(2+1)D Video Classifier specify the temporal sequence dimension, `"T"`. Specify the dimension labels `"SSCTB"` (spatial, spatial, channel, temporal, batch) for the video data, and `"CB"` for the label data.

The `minibatchqueue` object uses the supporting function `batchVideo`, listed at the end of this example, to batch the RGB video data.

```
params.ModelFilename = "r2plus1d-FiveClasses-hmdb51.mat";
if doTraining
    epoch = 1;
    bestLoss = realmax;
    accTrain = [];
    lossTrain = [];

    iteration = 1;
    start = tic;
    trainTime = start;
    shuffled = shuffleTrainDs(dsTrain);

    % Number of outputs is two: One for RGB frames, and one for ground truth labels.
    numOutputs = 2;
    mbq = createMiniBatchQueue(shuffled, numOutputs, params);

    % 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.
        [dlVideo,dlT] = next(mbq);

        % Evaluate the model loss and gradients using dlfeval.
        [loss,gradients,acc,state] = ...
            dlfeval(@modelLoss,r2plus1d,dlVideo,dlT);

        % Accumulate the loss and accuracies.
        lossTrain = [lossTrain, loss];
        accTrain = [accTrain, acc];

        % Update the network state.
        r2plus1d.State = state;

        % Update the gradients and parameters for the classifier
        % using the SGDM optimizer.
        [r2plus1d,params.Velocity,learnRate] = ...
            updateLearnables(r2plus1d,gradients,params,params.Velocity,iteration);
```

```
        if ~hasdata(mbq) || iteration == params.NumIterations
            % Current epoch is complete. Do validation and update progress.
            trainTime = toc(trainTime);

            [validationTime,cmat,lossValidation,accValidation] = ...
                doValidation(params, r2plus1d);

            accTrain = mean(accTrain);
            lossTrain = mean(lossTrain);

            % Update the training progress.
            displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
                accTrain,accValidation,lossTrain,lossValidation,trainTime,validationTime);
            updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accTrain,accValida

            % Save the trained video classifier and the parameters, that gave
            % the best validation loss so far. Use the saveData supporting function,
            % listed at the end of this example.
            bestLoss = saveData(r2plus1d,bestLoss,iteration,cmat,lossTrain,lossValidation,accTra
        end

        if ~hasdata(mbq) && iteration < params.NumIterations
            % Current epoch is complete. Initialize the training loss, accuracy
            % values, and minibatchqueue for the next epoch.
            accTrain = [];
            lossTrain = [];

            epoch = epoch + 1;
            trainTime = tic;
            shuffled = shuffleTrainDs(dsTrain);
            mbq = createMiniBatchQueue(shuffled, numOutputs, params);
        end

        iteration = iteration + 1;
    end

    % Display a message when training is complete.
    endVerboseOutput(params);

    disp("Model saved to: " + params.ModelFilename);
end
```

Evaluate Trained Video Classifier

Use the test data set to evaluate the accuracy of the trained video classifier.

Load the best model saved during training or use the pretrained model.

```
if doTraining
    transferLearned = load(params.ModelFilename);
    r2plus1dPretrained = transferLearned.data.r2plus1d;
end
```

Create a `minibatchqueue` object to load batches of the test data.

```
numOutputs = 2;
mbq = createMiniBatchQueue(dsVal, numOutputs, params);
```

For each batch of evaluation data, make predictions using the R(2+1)D Video Classifier, and compute the prediction accuracy using a confusion matrix.

```
numClasses = numel(params.Classes);
cmat = sparse(numClasses,numClasses);

while hasdata(mbq)
    [dlVideo,dlT] = next(mbq);

    % Computer the predictions of the trained R(2+1)D
    % Video Classifier.
    dlYPred = predict(r2plus1dPretrained,dlVideo);
    dlYPred = squeezeIfNeeded(dlYPred, dlT);

    % Aggregate the confusion matrix by using the maximum
    % values of the prediction scores and the ground truth labels.
    [~,TTest] = max(dlT,[],1);
    [~,YPred] = max(dlYPred,[],1);
    cmat = aggregateConfusionMetric(cmat,TTest,YPred);
end
```

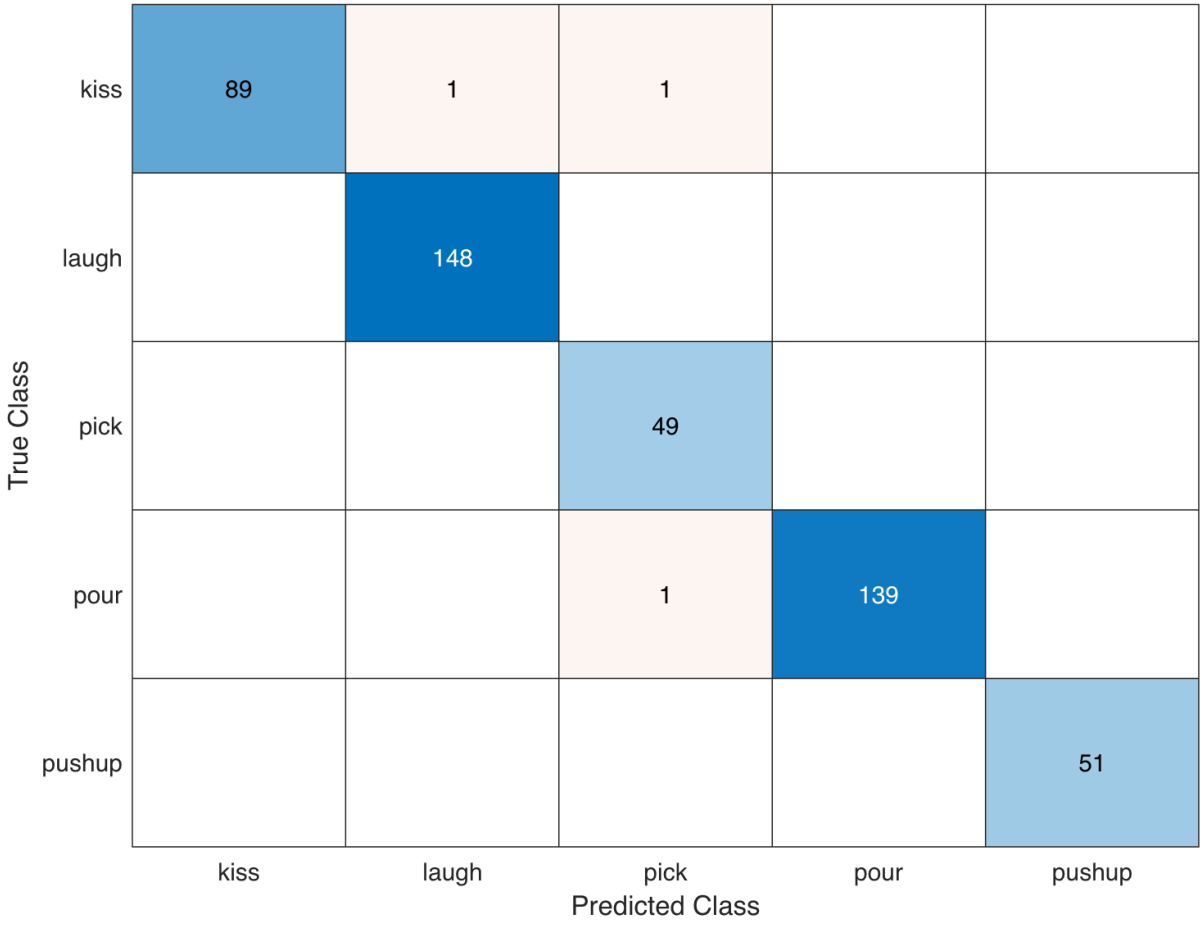
Compute the average clip classification accuracy for the trained R(2+1)D Video Classifier.

```
evalClipAccuracy = sum(diag(cmat))./sum(cmat,"all")
```

```
evalClipAccuracy = 0.9937
```

Display the confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```



The R(2+1)D video classifier that is pretrained on the Kinetics-400 dataset, provides strong performance for human activity recognition on transfer learning. The above training was run on 24GB Titan-X GPU for about 30 minutes. When training from scratch on a small activity recognition video dataset, the training time and convergence takes much longer than the pretrained video classifier. Transer learning using the Kinetics-400 pretrained R(2+1)D video classifier also avoids overfitting the classifier when ran for larger number of epochs. To learn more about video recognition using deep learning, see “Getting Started with Video Classification Using Deep Learning” on page 17-4.

Supporting Functions

`createFileDatastore`

The `createFileDatastore` function creates a `FileDatastore` object using the given folder name. 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(trainingFolder,numFrames,numChannels,classes,isDataForT
    readFcn = @(f,u)readVideo(f,u,numFrames,numChannels,classes,isDataForTraining);
    datastore = fileDatastore(trainingFolder,...
```

```
        'IncludeSubfolders',true,...
        'FileExtensions','.avi',...
        'ReadFcn',readFcn,...
        'ReadMode','partialfile');
end
```

shuffleTrainDs

The `shuffleTrainDs` function shuffles the files present in the training datastore, `dsTrain`.

```
function shuffled = shuffleTrainDs(dsTrain)
shuffled = copy(dsTrain);
transformed = isa(shuffled, 'matlab.io.datastore.TransformedDatastore');
if transformed
    files = shuffled.UnderlyingDatastores{1}.Files;
else
    files = shuffled.Files;
end
n = numel(files);
shuffledIndices = randperm(n);
if transformed
    shuffled.UnderlyingDatastores{1}.Files = files(shuffledIndices);
else
    shuffled.Files = files(shuffledIndices);
end

reset(shuffled);

end
```

readVideo

The `readVideo` function reads video frames, and the corresponding 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. During testing, all the frames are sequentially read. The video frames are resized to the required classifier network input size for training, and for testing and validation.

```
function [data,userdata,done] = readVideo(filename,userdata,numFrames,numChannels,classes,isDataI
    if isempty(userdata)
        userdata.reader      = VideoReader(filename);
        userdata.batchesRead = 0;

        userdata.label = getLabel(filename,classes);

        totalFrames = floor(userdata.reader.Duration * userdata.reader.FrameRate);
        totalFrames = min(totalFrames, userdata.reader.NumFrames);
        userdata.totalFrames = totalFrames;
        userdata.datatype = class(read(userdata.reader,1));
    end
    reader      = userdata.reader;
    totalFrames = userdata.totalFrames;
    label       = userdata.label;
    batchesRead = userdata.batchesRead;

    if isDataForTraining
        video = readForTraining(reader, numFrames, totalFrames);
```

```
    else
        video = readForValidation(reader, userdata.datatype, numChannels, numFrames, totalFrames)
    end

    data = {video, label};

    batchesRead = batchesRead + 1;

    userdata.batchesRead = batchesRead;

    if numFrames > totalFrames
        numBatches = 1;
    else
        numBatches = floor(totalFrames/numFrames);
    end
    % Set the done flag to true, if the reader has read all the frames or
    % if it is training.
    done = batchesRead == numBatches || isDataForTraining;
end
```

readForTraining

The `readForTraining` function reads the video frames for training the video classifier. The function reads the specific number of frames as per the network input size, with a randomly chosen starting frame. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForTraining(reader, numFrames, totalFrames)
    if numFrames >= totalFrames
        startIdx = 1;
        endIdx = totalFrames;
    else
        startIdx = randperm(totalFrames - numFrames + 1);
        startIdx = startIdx(1);
        endIdx = startIdx + numFrames - 1;
    end
    video = read(reader,[startIdx,endIdx]);
    if numFrames > totalFrames
        % Add more frames to fill in the network input size.
        additional = ceil(numFrames/totalFrames);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

readForValidation

The `readForValidation` function reads the video frames for evaluating the trained video classifier. The function reads the specific number of frames sequentially as per the network input size. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForValidation(reader, datatype, numChannels, numFrames, totalFrames)
    H = reader.Height;
    W = reader.Width;
    toRead = min([numFrames,totalFrames]);
    video = zeros([H,W,numChannels,toRead], datatype);
    frameIndex = 0;
```

```
    while hasFrame(reader) && frameIndex < numFrames
        frame = readFrame(reader);
        frameIndex = frameIndex + 1;
        video(:,:,:,frameIndex) = frame;
    end

    if frameIndex < numFrames
        video = video(:,:,:,1:frameIndex);
        additional = ceil(numFrames/frameIndex);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

`getLabel`

The `getLabel` function obtains the label name from the full path of a filename. The label for a file is the folder in which it exists. For example, for a file path such as `"/path/to/dataset/clapping/video_0001.avi"`, the label name is `"clapping"`.

```
function label = getLabel(filename,classes)
    folder = fileparts(string(filename));
    [~,label] = fileparts(folder);
    label = categorical(string(label), string(classes));
end
```

`augmentVideo`

The `augmentVideo` function augments the video frames for training the video classifier. The function augments a video sequence with the same augmentation technique provided by the `augmentTransform` function.

```
function data = augmentVideo(data)
    numClips = size(data,1);
    for ii = 1:numClips
        video = data{ii,1};
        % HxWxC
        sz = size(video,[1,2,3]);
        % One augment fcn per clip
        augmentFcn = augmentTransform(sz);
        data{ii,1} = augmentFcn(video);
    end
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
```

preprocessVideoClips

The `preprocessVideoClips` function preprocesses the training video data to resize to the R(2+1)D Video Classifier input size. It takes the `InputNormalizationStatistics` and the `InputSize` properties of the video classifier in a struct, `info`. The `InputNormalizationStatistics` property is used to rescale the video frames between 0 and 1, and then normalize the rescaled data using mean and standard deviation. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` as values for `SizingOption` to random crop or center crop the input data to the input size of the video classifier.

```
function data = preprocessVideoClips(data, info)
    inputSize = info.InputSize(1:2);
    sizingOption = info.SizingOption;
    switch sizingOption
        case "resize"
            sizingFcn = @(x)imresize(x,inputSize);
        case "randomcrop"
            sizingFcn = @(x)cropVideo(x,@randomCropWindow2d,inputSize);
        case "centercrop"
            sizingFcn = @(x)cropVideo(x,@centerCropWindow2d,inputSize);
    end
    numClips = size(data,1);

    minValue = info.Statistics.Min;
    maxValue = info.Statistics.Max;
    minValue = reshape(minValue, 1, 1, 3);
    maxValue = reshape(maxValue, 1, 1, 3);

    meanValue = info.Statistics.Mean;
    stdValue  = info.Statistics.StandardDeviation;
    meanValue = reshape(meanValue, 1, 1, 3);
    stdValue  = reshape(stdValue, 1, 1, 3);

    for ii = 1:numClips
        video = data{ii,1};
        resized = sizingFcn(video);

        % Cast the input to single.
        resized = single(resized);

        % Rescale the input between 0 and 1.
        resized = rescale(resized,0,1,"InputMin",minValue,"InputMax",maxValue);

        % Normalize using mean and standard deviation.
        resized = resized - meanValue;
        resized = resized ./ stdValue;
        data{ii,1} = resized;
    end

    function outData = cropVideo(data, cropFcn, inputSize)
        imsz = size(data,[1,2]);
        cropWindow = cropFcn(imsz, inputSize);
        numBatches = size(data,4);
```

```
        sz = [inputSize, size(data,3), numBatches];
        outData = zeros(sz, 'like', data);
        for b = 1:numBatches
            outData(:,:,:,b) = imcrop(data(:,:,:,b), cropWindow);
        end
    end
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

inputFormat(1:numOutputs-1) = "SSCTB";
outputFormat = "CB";
mbq = minibatchqueue(datastore, numOutputs, ...
    "MiniBatchSize", params.MiniBatchSize, ...
    "MiniBatchFcn", @batchVideo, ...
    "MiniBatchFormat", [inputFormat,outputFormat]);
end
```

batchVideo

The `batchVideo` function batches the video, and the label data from cell arrays. It uses `onehotencode` function to encode ground truth categorical labels into one-hot arrays. The one-hot encoded array contains a `1` in the position corresponding to the class of the label, and `0` in every other position.

```
function [video,labels] = batchVideo(video, labels)
% Batch dimension: 5
video = cat(5,video{:});

% Batch dimension: 2
labels = cat(2,labels{:});

% Feature dimension: 1
labels = onehotencode(labels,1);
end
```

modelLoss

The `modelLoss` function takes as input a mini-batch of RGB data `dlRGB`, and the corresponding target `dlT`, 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 `modelLoss` function using the `dlfeval` function in the training loop.

```
function [loss,gradientsRGB,acc,stateRGB] = modelLoss(r2plus1d,dlRGB,dlT)
[dlYPredRGB,stateRGB] = forward(r2plus1d,dlRGB);
dlYPred = squeezeIfNeeded(dlYPredRGB, dlT);

loss = crossentropy(dlYPred,dlT);

gradientsRGB = dlgradient(loss,r2plus1d.Learnables);

% Calculate the accuracy of the predictions.
[~,TTest] = max(dlT,[],1);
[~,YPred] = max(dlYPred,[],1);

acc = gather(extractdata(sum(TTest == YPred)./numel(TTest)));
end
```

`squeezeIfNeeded`

```
function dlYPred = squeezeIfNeeded(dlYPred, T)
if ~isequal(size(T), size(dlYPred))
    dlYPred = squeeze(dlYPred);
    dlYPred = dlarray(dlYPred,dims(T));
end
end
```

`updateLearnables`

The `updateLearnables` function updates the provided `dlnetwork` object with gradients and other parameters using SGDM optimization function `sgdmupdate`.

```
function [r2plus1d,velocity,learnRate] = updateLearnables(r2plus1d,gradients,params,velocity,ite
    % Determine the learning rate using the cosine-annealing learning rate schedule.
    learnRate = cosineAnnealingLearnRate(iteration, params);

    % Apply L2 regularization to the weights.
    learnables = r2plus1d.Learnables;
    idx = learnables.Parameter == "Weights";
    gradients(idx,:) = dlupdate(@(g,w) g + params.L2Regularization*w, gradients(idx,:), learnable

    % Update the network parameters using the SGDM optimizer.
    [r2plus1d, velocity] = sgdmupdate(r2plus1d, 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 `TTest`.

```
function cmat = aggregateConfusionMetric(cmat,TTest,YPred)
TTest = gather(extractdata(TTest));
YPred = gather(extractdata(YPred));
[m,n] = size(cmat);
cmat = cmat + full(sparse(TTest,YPred,1,m,n));
end
```

doValidation

The `doValidation` function validates the video classifier using the validation data.

```
function [validationTime, cmat, lossValidation, accValidation] = doValidation(params, r2plus1d)

    validationTime = tic;

    numOutputs = 2;
    mbq = createMiniBatchQueue(params.ValidationData, numOutputs, params);

    lossValidation = [];
    numClasses = numel(params.Classes);
    cmat = sparse(numClasses,numClasses);
    while hasdata(mbq)

        [dlVideo,T] = next(mbq);

        % Pass the video input through the R(2+1)D Video Classifier.
        dlT = predict(r2plus1d,dlVideo);
        dlT = squeezeIfNeeded(dlT, T);

        % Calculate the cross-entropy loss.
        loss = crossentropy(dlT,T);

        % Calculate the accuracy of the predictions.
        [~,TTest] = max(T,[],1);
        [~,YPred] = max(dlT,[],1);

        lossValidation = [lossValidation,loss];
        cmat = aggregateConfusionMetric(cmat,TTest,YPred);
    end
    lossValidation = mean(lossValidation);
    accValidation = sum(diag(cmat))./sum(cmat,"all");

    validationTime = toc(validationTime);
end
```

saveData

The `saveData` function saves the given R(2+1)D Video Classifier, accuracy, loss, and other training parameters to a MAT-file.

```
function bestLoss = saveData(r2plus1d,bestLoss,iteration,cmat,lossTrain,lossValidation,accTrain,a
if iteration >= params.SaveBestAfterIteration
    lossValidtion = extractdata(gather(lossValidation));
    if lossValidtion < bestLoss
        params = rmfield(params, 'Velocity');
        bestLoss = lossValidtion;
        r2plus1d = gatherFromGPUToSave(r2plus1d);
        data.BestLoss = bestLoss;
        data.TrainingLoss = extractdata(gather(lossTrain));
        data.TrainingAccuracy = accTrain;
        data.ValidationAccuracy = accValidation;
        data.ValidationConfmat= cmat;
        data.r2plus1d = r2plus1d;
        data.Params = params;
        save(params.ModelFilename, 'data');
    end
end
end
```

gatherFromGPUToSave

The `gatherFromGPUToSave` function gathers data from the GPU in order to save the video classifier to disk.

```
function r2plus1d = gatherFromGPUToSave(r2plus1d)
if ~canUseGPU
    return;
end
r2plus1d.Learnables = gatherValues(r2plus1d.Learnables);
r2plus1d.State = gatherValues(r2plus1d.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 ~isfolder(hmdbFolder)
    error("Download 'hmdb51_org.rar' file using the supporting function 'downloadHMDB51' before r
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 ~isfolder(dataLoc)
    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');

% Download the RAR file and save it to the download folder.
if ~isfile(rarFileName)
    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
```

updateProgressPlot

The `updateProgressPlot` function updates the progress plot with loss and accuracy information during training.

```
function updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accuracyTrain,accurac
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
```

initializeVerboseOutput

```
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 | Validation | Mini-Batch | Validation
    disp("|       |           |  (hh:mm:ss)  |  Accuracy  |  Accuracy  |    Loss    |    Loss
    disp("|=========================================================================================
end
end
```

displayVerboseOutputEveryEpoch

```
function displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
        accTrain,accValidation,lossTrain,lossValidation,trainTime,validationTime)
    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);

        lossTrain = gather(extractdata(lossTrain));
        lossTrain = compose('%.4f',lossTrain);

        accTrain = composePadAccuracy(accTrain);
        learnRate = compose('%.13f',learnRate);

        disp("| " + ...
            pad(string(epoch),5,'both') + " | " + ...
            pad(string(iteration),9,'both') + " | " + ...
            pad(string(D),12,'both') + " | " + ...
            pad(string(accTrain),10,'both') + " | " + ...
            pad(string(accValidation),10,'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
    function acc = composePadAccuracy(acc)
        acc = compose('%.2f',acc*100) + "%";
        acc = pad(string(acc),6,'left');
    end
end
```

endVerboseOutput

The `endVerboseOutput` function displays the end of verbose output during training.

```
function endVerboseOutput(params)
if params.Verbose
    disp("|========================================================================================
end
end
```

References

[1] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, Manohar Paluri. "A Closer Look at Spatiotemporal Convolutions for Action Recognition". Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 6450-6459.

[2] Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. "SlowFast Networks for Video Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), 2019.

[3] Loshchilov, Ilya, and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts." *International Conferencee on Learning Representations 2017*. Toulon, France: ICLR, 2017.

[4] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, Andrew Zisserman. "The Kinetics Human Action Video Dataset." *arXiv preprint arXiv:1705.06950*, 2017.

Activity Recognition from Video and Optical Flow Data Using Deep Learning

This example first shows how to perform activity recognition using a pretrained Inflated 3-D (I3D) two-stream convolutional neural network based video classifier and then shows how to use transfer learning to train such a video classifier using RGB and optical flow data from videos [1] on page 3-0 .

Overview

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 inaccurate ground truth data for video data sets, the variety of gestures that actors in a video can perform, the heavily class imbalanced datasets, and the large amount of data required to train a robust classifier from scratch. Deep learning techniques, such as I3D two-stream convolutional networks [1] on page 3-0 , R(2+1)D [4 on page 3-0], and SlowFast [5 on page 3-0] have shown improved performance on smaller datasets using transfer learning with networks pretrained on large video activity recognition datasets, such as Kinetics-400 [6 on page 3-0].

Note: This example requires the Computer Vision Toolbox™ Model for Inflated-3D Video Classification. You can install the Computer Vision Toolbox Model for Inflated-3D Video Classification from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Perform Activity Recognition Using a Pretrained Inflated-3D Video Classifier

Download the pretrained Inflated-3D video classifier along with a video file on which to perform activity recognition. The size of the downloaded zip file is around 89 MB.

```
downloadFolder = fullfile(tempdir,"hmdb51","pretrained","I3D");
if ~isfolder(downloadFolder)
    mkdir(downloadFolder);
end

filename = "activityRecognition-I3D-HMDB51-21b.zip";

zipFile = fullfile(downloadFolder,filename);
if ~isfile(zipFile)
    disp('Downloading the pretrained network...');
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + filename;
    websave(zipFile,downloadURL);
    unzip(zipFile,downloadFolder);
end
```

Load the pretrained Inflated-3D video classifier.

```
pretrainedDataFile = fullfile(downloadFolder,"inflated3d-FiveClasses-hmdb51.mat");
pretrained = load(pretrainedDataFile);
inflated3dPretrained = pretrained.data.inflated3d;
```

Display the class label names of the pretrained video classifier.

```
classes = inflated3dPretrained.Classes
```

```
classes = 5×1 categorical
     kiss
     laugh
     pick
     pour
     pushup
```

Read and display the video `pour.avi` using `VideoReader` and `vision.VideoPlayer`.

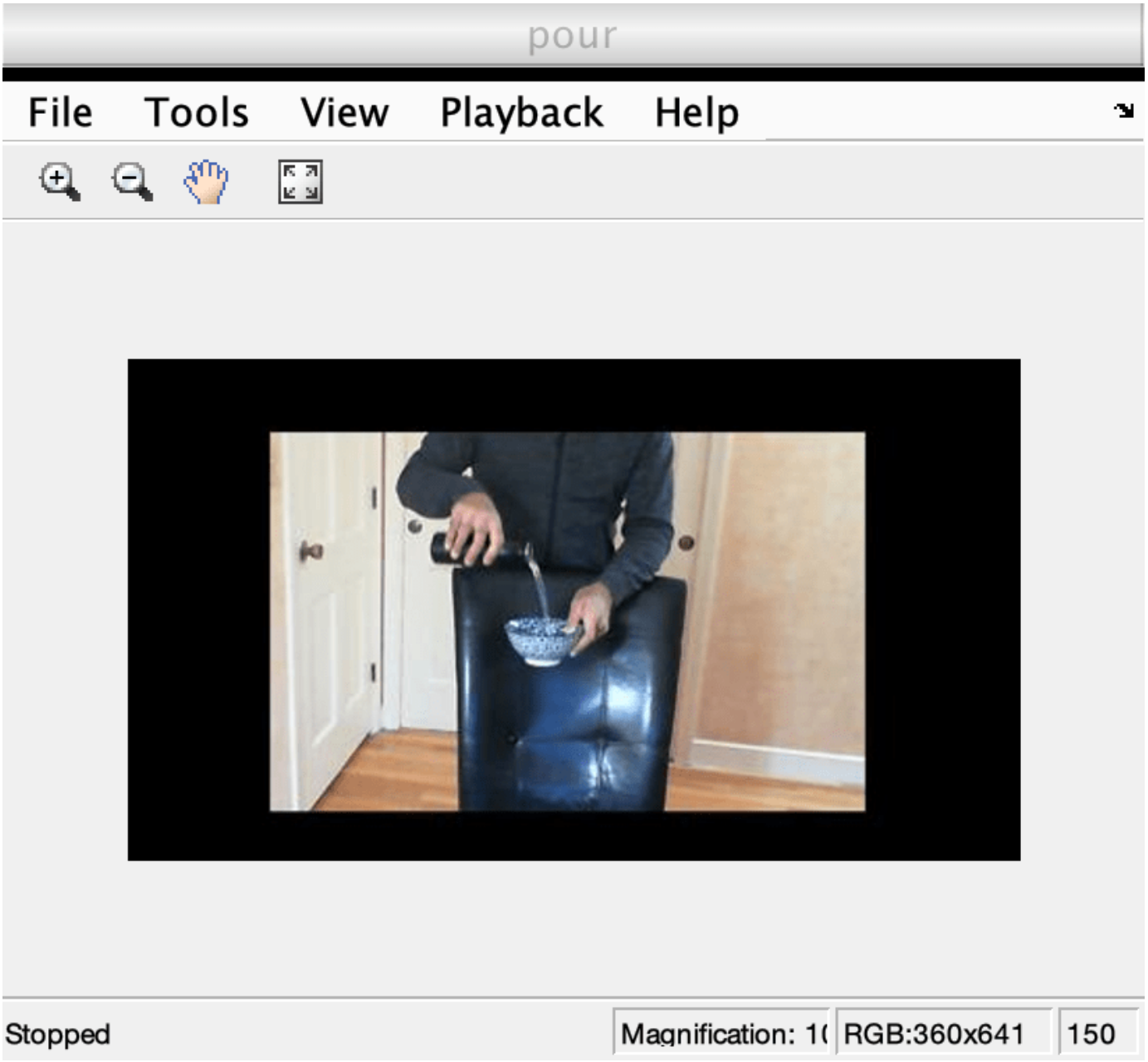
```
videoFilename = fullfile(downloadFolder, "pour.avi");

videoReader = VideoReader(videoFilename);
videoPlayer = vision.VideoPlayer;
videoPlayer.Name = "pour";

while hasFrame(videoReader)
   frame = readFrame(videoReader);
   % Resize the frame for display.
   frame = imresize(frame, 1.5);
   step(videoPlayer,frame);
end
release(videoPlayer);
```

Choose 10 randomly selected video sequences to classify the video, to uniformly cover the entirety of the file to find the action class that is predominant in the video.

```
numSequences = 10;
```

Classify the video file using the `classifyVideoFile` function.

```
[actionLabel,score] = classifyVideoFile(inflated3dPretrained, videoFilename, "NumSequences", numS
```




```
actionLabel = categorical
     pour


score = single
    0.4482
```

Train a Video Classifier for Gesture Recognition

This section of the example shows how the video classifier shown above is trained using transfer learning. Set the `doTraining` variable to `false` to use the pretrained video classifier without having

to wait for training to complete. Alternatively, if you want to train the video classifier, set the `doTraining` variable to `true`.

```
doTraining = false;
```

Download Training and Validation Data

This example trains an Inflated-3D (I3D) Video Classifier 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;
end
dataFolder = fullfile(downloadFolder, "hmdb51_org");
```

Split the data set into a training set for training the classifier, and a test set for evaluating the classifier. Use 80% of the data for the training set and the rest for the test set. Use `folders2labels` and `splitlabels` to create label information from folders and split the data based on each label into training and test data sets by randomly selecting a proportion of files from each label.

```
[labels,files] = folders2labels(fullfile(dataFolder,string(classes)),...
    "IncludeSubfolders",true,...
    "FileExtensions",'.avi');

indices = splitlabels(labels,0.8,'randomized');

trainFilenames = files(indices{1});
testFilenames  = files(indices{2});
```

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
```

Load Dataset

This example uses a datastore to read the videos scenes, the corresponding optical flow data, and the corresponding labels from the video files.

Specify the number of video frames the datastore should be configured to output for each time data is read from the datastore.

```
numFrames = 64;
```

A value of 64 is used here to balance memory usage and classification time. Common values to consider are 16, 32, 64, or 128. Using more frames helps capture additional temporal information, but requires more memory. You might need to lower this value depending on your system resources. Empirical analysis is required to determine the optimal number of frames.

Next, specify the height and width of the frames the datastore should be configured to output. The datastore automatically resizes the raw video frames to the specified size to enable batch processing of multiple video sequences.

```
frameSize = [112,112];
```

A value of [112 112] is used to capture longer temporal relationships in the video scene which help classify activities with long time durations. Common values for the size are [112 112], [224 224], or [256 256]. Smaller sizes enable the use of more video frames at the cost of memory usage, processing time, and spatial resolution. The minimum height and width of the video frames in the HMDB51 data set are 240 and 176, respectively. 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`. As with the number of frames, empirical analysis is required to determine the optimal values.

Specify the number of channels as 3 for the RGB video subnetwork, and 2 for the optical flow subnetwork of the I3D video classifier. 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;
```

Use the helper function, `createFileDatastore`, to configure two `FileDatastore` objects for loading the data, one for training and another for validation. The helper function is listed at the end of this example. Each datastore reads a video file to provide RGB data and the corresponding label information.

```
isDataForTraining = true;
dsTrain = createFileDatastore(trainFilenames,numFrames,rgbChannels,classes,isDataForTraining);

isDataForTraining = false;
dsVal = createFileDatastore(testFilenames,numFrames,rgbChannels,classes,isDataForTraining);
```

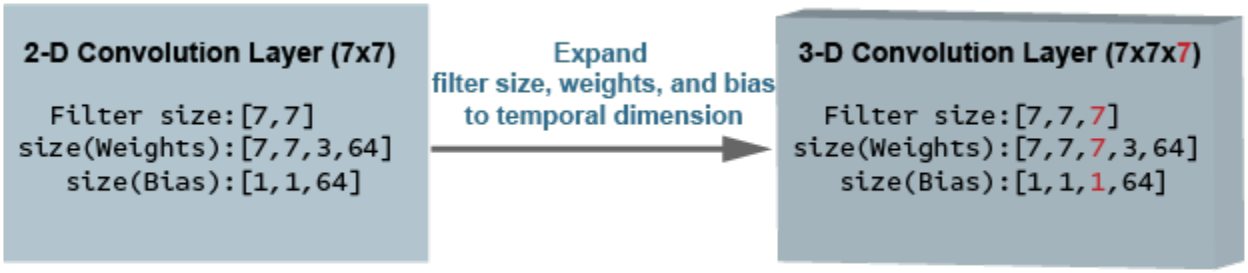
Define Network Architecture

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).

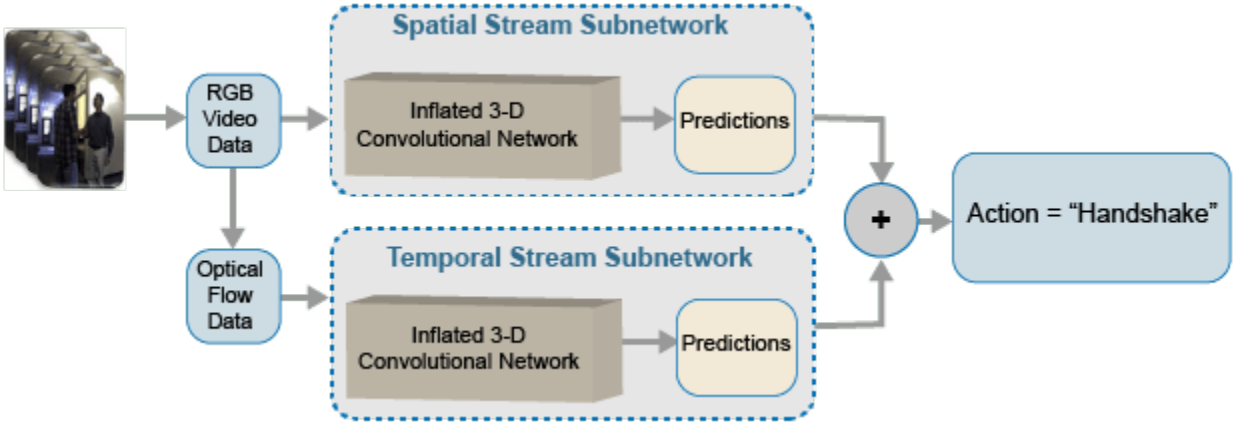


Two-Stream I3D Network

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.



Configure Inflated-3D (I3D) Video Classifier for Transfer Learning

In this example, you create an I3D video classifier based on the GoogLeNet architecture, a 3D Convolution Neural Network Video Classifier pretrained on the Kinetics-400 dataset.

Specify GoogLeNet as the backbone convolution neural network architecture for the I3D video classifier that contains two subnetworks, one for video data and another for optical flow data.

```
baseNetwork = "googlenet-video-flow";
```

Specify the input size for the Inflated-3D Video Classifier.

```
inputSize = [frameSize, rgbChannels, numFrames];
```

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 to normalize the input data.

```
oflowMin = squeeze(inputStats.oflowMin)';
oflowMax = squeeze(inputStats.oflowMax)';
rgbMin   = squeeze(inputStats.rgbMin)';
rgbMax   = squeeze(inputStats.rgbMax)';

stats.Video.Min               = rgbMin;
stats.Video.Max               = rgbMax;
stats.Video.Mean              = [];
stats.Video.StandardDeviation = [];

stats.OpticalFlow.Min               = oflowMin(1:flowChannels);
stats.OpticalFlow.Max               = oflowMax(1:flowChannels);
stats.OpticalFlow.Mean              = [];
stats.OpticalFlow.StandardDeviation = [];
```

Create the I3D Video Classifier by using the `inflated3dVideoClassifier` function.

```
i3d = inflated3dVideoClassifier(baseNetwork,string(classes),...
    "InputSize",inputSize,...
    "InputNormalizationStatistics",stats);
```

Specify a model name for the video classifier.

```
i3d.ModelName = "Inflated-3D Activity Recognizer Using Video and Optical Flow";
```

Augment and Preprocess Training Data

Data augmentation provides a way to use limited data sets for training. Augmentation on video data must be the same for a collection of frames, i.e. a video sequence, based on the network input size. Minor changes, such as translation, cropping, or transforming an image, provide, new, distinct, and unique images that you can use to train a robust video classifier. Datastores are a convenient way to read and augment collections of data. Augment the training video data by using the `augmentVideo` supporting function, defined at the end of this example.

```
dsTrain = transform(dsTrain, @augmentVideo);
```

Preprocess the training video data to resize to the Inflated-3D Video Classifier input size, by using the `preprocessVideoClips`, defined at the end of this example. Specify the `InputNormalizationStatistics` property of the video classifier and input size to the preprocessing function as field values in a struct, `preprocessInfo`. The `InputNormalizationStatistics` property is used to rescale the video frames and optical flow data between -1 and 1. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` to random crop or center crop the input data to the input size of the video classifier. 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.

```
preprocessInfo.Statistics = i3d.InputNormalizationStatistics;
preprocessInfo.InputSize = inputSize;
preprocessInfo.SizingOption = "resize";
dsTrain = transform(dsTrain, @(data)preprocessVideoClips(data, preprocessInfo));
dsVal = transform(dsVal, @(data)preprocessVideoClips(data, preprocessInfo));
```

Define Model Gradients Function

Create the supporting function `modelGradients`, listed at the end of this example. The `modelGradients` function takes as input the I3D video classifier `i3d`, a mini-batch of input data `dlRGB` and `dlFlow`, 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 classifier, and the mini-batch accuracy of the classifier.

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 classifier 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 600 iterations. Specify the iteration after which to save the video classifier with the best validation accuracy by using the `SaveBestAfterIteration` parameter.

Specify the cosine-annealing learning rate schedule [3 on page 3-0] parameters:

- A minimum learning rate of 1e-4.
- A maximum learning rate of 1e-3.
- Cosine number of iterations of 100, 200, and 300, 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:

- 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. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

```
params.Classes = classes;
params.MiniBatchSize = 20;
params.NumIterations = 600;
params.SaveBestAfterIteration = 400;
params.CosineNumIterations = [100, 200, 300];
params.MinLearningRate = 1e-4;
params.MaxLearningRate = 1e-3;
params.Momentum = 0.9;
params.VelocityRGB = [];
params.VelocityFlow = [];
params.L2Regularization = 0.0005;
params.ProgressPlot = true;
params.Verbose = true;
params.ValidationData = dsVal;
params.DispatchInBackground = false;
params.NumWorkers = 4;
```

Train I3D Video Classifier

Train the I3D video classifier using the RGB video data and optical flow data.

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 video data or optical flow data and the labels to `dlarray` objects with the underlying type single.
- To enable processing the time dimension of the the video data using the I3D Video Classifier specify the temporal sequence dimension, `"T"`. Specify the dimension labels `"SSCTB"` (spatial, spatial, channel, temporal, batch) for the video data, and `"CB"` for the label data.

The `minibatchqueue` object uses the supporting function `batchVideoAndFlow`, listed at the end of this example, to batch the RGB video and optical flow data.

```
params.ModelFilename = "inflated3d-FiveClasses-hmdb51.mat";
if doTraining
    epoch     = 1;
    bestLoss  = realmax;

    accTrain     = [];
    accTrainRGB  = [];
    accTrainFlow = [];
    lossTrain    = [];

    iteration = 1;
    start     = tic;
    trainTime = start;
```

```
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);

% 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.
    [dlVideo,dlFlow,dlY] = next(mbq);

    % Evaluate the model gradients and loss using dlfeval.
    [gradRGB,gradFlow,loss,acc,accRGB,accFlow,stateRGB,stateFlow] = ...
        dlfeval(@modelGradients,i3d,dlVideo,dlFlow,dlY);

    % Accumulate the loss and accuracies.
    lossTrain    = [lossTrain, loss];
    accTrain     = [accTrain, acc];
    accTrainRGB  = [accTrainRGB, accRGB];
    accTrainFlow = [accTrainFlow, accFlow];

    % Update the network state.
    i3d.VideoState       = stateRGB;
    i3d.OpticalFlowState = stateFlow;

    % Update the gradients and parameters for the RGB and optical flow
    % subnetworks using the SGDM optimizer.
    [i3d.VideoLearnables,params.VelocityRGB] = ...
        updateLearnables(i3d.VideoLearnables,gradRGB,params,params.VelocityRGB,iteration);
    [i3d.OpticalFlowLearnables,params.VelocityFlow,learnRate] = ...
        updateLearnables(i3d.OpticalFlowLearnables,gradFlow,params,params.VelocityFlow,itera

    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, i3d);

        accTrain     = mean(accTrain);
        accTrainRGB  = mean(accTrainRGB);
        accTrainFlow = mean(accTrainFlow);
        lossTrain    = mean(lossTrain);

        % Update the training progress.
        displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
            accTrain,accTrainRGB,accTrainFlow,...
            accValidation,accValidationRGB,accValidationFlow,...
            lossTrain,lossValidation,trainTime,validationTime);
        updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accTrain,accValida
```

```
            % Save the trained video classifier and the parameters, that gave
            % the best validation loss so far. Use the saveData supporting function,
            % listed at the end of this example.
            bestLoss = saveData(i3d,bestLoss,iteration,cmat,lossTrain,lossValidation,...
                accTrain,accValidation,params);
        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: " + params.ModelFilename);
end
```

Evaluate Trained Network

Use the test data set to evaluate the accuracy of the trained video classifier.

Load the best model saved during training or use the pretrained model.

```
if doTraining
    transferLearned = load(params.ModelFilename);
    inflated3dPretrained = transferLearned.data.inflated3d;
end
```

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 networks, take the average of the predictions, and compute the prediction accuracy using a confusion matrix.

```
numClasses = numel(classes);
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 I3D Video Classifier to get the separate predictions.
    [dlYPredRGB,dlYPredFlow] = predict(inflated3dPretrained,dlRGB,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
```

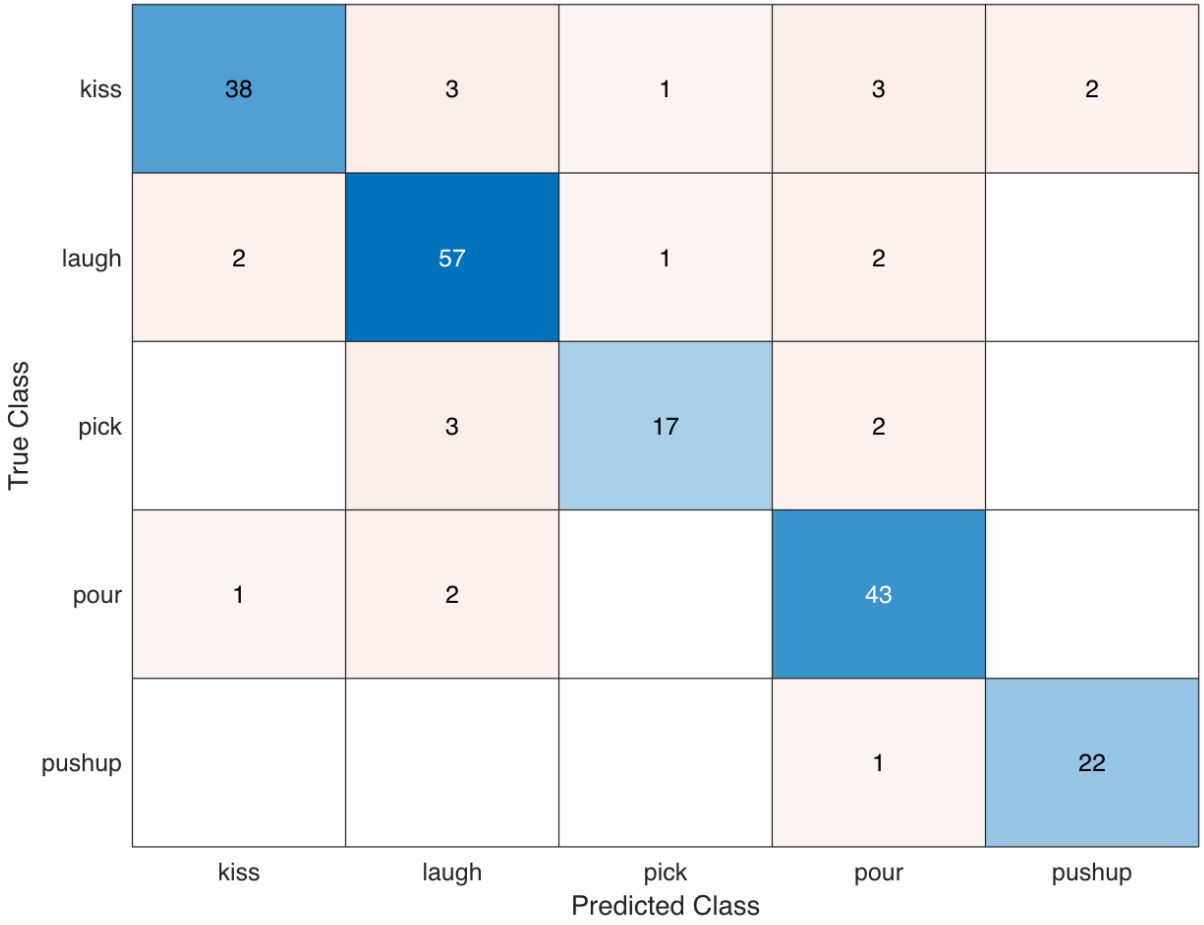
Compute the average classification accuracy for the trained networks.

```
accuracyEval = sum(diag(cmat))./sum(cmat,"all")
```

```
accuracyEval = 0.8850
```

Display the confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```



The Inflated-3D video classifier that is pretrained on the Kinetics-400 dataset, provides better performance for human activity recognition on transfer learning. The above training was run on 24GB Titan-X GPU for about 100 minutes. When training from scratch on a small activity recognition video dataset, the training time and convergence takes much longer than the pretrained video classifier. Transer learning using the Kinetics-400 pretrained Inflated-3D video classifier also avoids overfitting the classifier when ran for larger number of epochs. However, the SlowFast Video Classifier and R(2+1)D Video Classifier that are pretrained on the Kinetics-400 dataset provide better performance and faster convergence during training compared to the Inflated-3D Video Classifier. To learn more about video recognition using deep learning, see “Getting Started with Video Classification Using Deep Learning” on page 17-4.

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 [labelName, filename] = getLabelFilename(filename)
    fileNameSplit = split(filename,'/');
    labelName = fileNameSplit{end-1};
    filename = fileNameSplit{end};
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(trainingFolder,numFrames,numChannels,classes,isDataForT
    readFcn = @(f,u)readVideo(f,u,numFrames,numChannels,classes,isDataForTraining);
    datastore = fileDatastore(trainingFolder,...
        'IncludeSubfolders',true,...
        'FileExtensions','.avi',...
        'ReadFcn',readFcn,...
        'ReadMode','partialfile');
end
```

`shuffleTrainDs`

The `shuffleTrainDs` function shuffles the files present in the training datastore `dsTrain`.

```
function shuffled = shuffleTrainDs(dsTrain)
shuffled = copy(dsTrain);
transformed = isa(shuffled, 'matlab.io.datastore.TransformedDatastore');
if transformed
    files = shuffled.UnderlyingDatastores{1}.Files;
else
    files = shuffled.Files;
end
n = numel(files);
shuffledIndices = randperm(n);
if transformed
    shuffled.UnderlyingDatastores{1}.Files = files(shuffledIndices);
else
```

```
    shuffled.Files = files(shuffledIndices);
end

reset(shuffled);
end
```

readVideo

The `readVideo` function reads video frames, and the corresponding 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. During testing, all the frames are sequentially read. The video frames are resized to the required classifier network input size for training, and for testing and validation.

```
function [data,userdata,done] = readVideo(filename,userdata,numFrames,numChannels,classes,isDataF
    if isempty(userdata)
        userdata.reader      = VideoReader(filename);
        userdata.batchesRead = 0;

        userdata.label = getLabel(filename,classes);

        totalFrames = floor(userdata.reader.Duration * userdata.reader.FrameRate);
        totalFrames = min(totalFrames, userdata.reader.NumFrames);
        userdata.totalFrames = totalFrames;
        userdata.datatype = class(read(userdata.reader,1));
    end
    reader      = userdata.reader;
    totalFrames = userdata.totalFrames;
    label       = userdata.label;
    batchesRead = userdata.batchesRead;

    if isDataForTraining
        video = readForTraining(reader, numFrames, totalFrames);
    else
        video = readForValidation(reader, userdata.datatype, numChannels, numFrames, totalFrames)
    end

    data = {video, label};

    batchesRead = batchesRead + 1;

    userdata.batchesRead = batchesRead;

    if numFrames > totalFrames
        numBatches = 1;
    else
        numBatches = floor(totalFrames/numFrames);
    end
    % Set the done flag to true, if the reader has read all the frames or
    % if it is training.
    done = batchesRead == numBatches || isDataForTraining;
end
```

readForTraining

The `readForTraining` function reads the video frames for training the video classifier. The function reads the specific number of frames as per the network input size, with a randomly chosen starting

frame. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForTraining(reader, numFrames, totalFrames)
    if numFrames >= totalFrames
        startIdx = 1;
        endIdx = totalFrames;
    else
        startIdx = randperm(totalFrames - numFrames + 1);
        startIdx = startIdx(1);
        endIdx = startIdx + numFrames - 1;
    end
    video = read(reader,[startIdx,endIdx]);
    if numFrames > totalFrames
        % Add more frames to fill in the network input size.
        additional = ceil(numFrames/totalFrames);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

`readForValidation`

The `readForValidation` function reads the video frames for evaluating the trained video classifier. The function reads the specific number of frames sequentially as per the network input size. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForValidation(reader, datatype, numChannels, numFrames, totalFrames)
    H = reader.Height;
    W = reader.Width;
    toRead = min([numFrames,totalFrames]);
    video = zeros([H,W,numChannels,toRead], datatype);
    frameIndex = 0;
    while hasFrame(reader) && frameIndex < numFrames
        frame = readFrame(reader);
        frameIndex = frameIndex + 1;
        video(:,:,:,frameIndex) = frame;
    end

    if frameIndex < numFrames
        video = video(:,:,:,1:frameIndex);
        additional = ceil(numFrames/frameIndex);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

`getLabel`

The `getLabel` function obtains the label name from the full path of a filename. The label for a file is the folder in which it exists. For example, for a file path such as `"/path/to/dataset/clapping/video_0001.avi"`, the label name is `"clapping"`.

```
function label = getLabel(filename,classes)
    folder = fileparts(string(filename));
    [~,label] = fileparts(folder);
```

```
    label = categorical(string(label), string(classes));
end
```

augmentVideo

The `augmentVideo` function uses the augment transform function provided by the `augmentTransform` supporting function to apply the same augmentation across a video sequence.

```
function data = augmentVideo(data)
    numSequences = size(data,1);
    for ii = 1:numSequences
        video = data{ii,1};
        % HxWxC
        sz = size(video,[1,2,3]);
        % One augmentation per sequence
        augmentFcn = augmentTransform(sz);
        data{ii,1} = augmentFcn(video);
    end
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
```

preprocessVideoClips

The `preprocessVideoClips` function preprocesses the training video data to resize to the Inflated-3D Video Classifier input size. It takes the `InputNormalizationStatistics` and the `InputSize` properties of the video classifier in a struct, `info`. The `InputNormalizationStatistics` property is used to rescale the video frames and optical flow data between -1 and 1. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` as values for `SizingOption` to random crop or center crop the input data to the input size of the video classifier.

```
function preprocessed = preprocessVideoClips(data, info)
inputSize = info.InputSize(1:2);
sizingOption = info.SizingOption;
switch sizingOption
    case "resize"
        sizingFcn = @(x)imresize(x,inputSize);
    case "randomcrop"
        sizingFcn = @(x)cropVideo(x,@randomCropWindow2d,inputSize);
    case "centercrop"
        sizingFcn = @(x)cropVideo(x,@centerCropWindow2d,inputSize);
```

```
end
numClips = size(data,1);

rgbMin   = info.Statistics.Video.Min;
rgbMax   = info.Statistics.Video.Max;
oflowMin = info.Statistics.OpticalFlow.Min;
oflowMax = info.Statistics.OpticalFlow.Max;

numChannels = length(rgbMin);
rgbMin   = reshape(rgbMin, 1, 1, numChannels);
rgbMax   = reshape(rgbMax, 1, 1, numChannels);

numChannels = length(oflowMin);
oflowMin = reshape(oflowMin, 1, 1, numChannels);
oflowMax = reshape(oflowMax, 1, 1, numChannels);

preprocessed = cell(numClips, 3);
for ii = 1:numClips
    video   = data{ii,1};
    resized = sizingFcn(video);
    oflow   = computeFlow(resized,inputSize);

    % Cast the input to single.
    resized = single(resized);
    oflow   = single(oflow);

    % Rescale the input between -1 and 1.
    resized = rescale(resized,-1,1,"InputMin",rgbMin,"InputMax",rgbMax);
    oflow   = rescale(oflow,-1,1,"InputMin",oflowMin,"InputMax",oflowMax);

    preprocessed{ii,1} = resized;
    preprocessed{ii,2} = oflow;
    preprocessed{ii,3} = data{ii,2};
end
end

function outData = cropVideo(data, cropFcn, inputSize)
imsz = size(data,[1,2]);
cropWindow = cropFcn(imsz, inputSize);
numFrames = size(data,4);
sz = [inputSize, size(data,3), numFrames];
outData = zeros(sz, 'like', data);
for f = 1:numFrames
    outData(:,:,:,f) = imcrop(data(:,:,:,f), cropWindow);
end
end
```

`computeFlow`

The `computeFlow` function takes as input a video sequence, `videoFrames`, and computes the the corresponding optical flow data `opticalFlowData` using `opticalFlowFarneback`. The optical flow data contains two channels, which correspond to the *x*- and *y*- components of velocity.

```
function opticalFlowData = computeFlow(videoFrames, inputSize)
opticalFlow = opticalFlowFarneback;
numFrames = size(videoFrames,4);
sz = [inputSize, 2, numFrames];
opticalFlowData = zeros(sz, 'like', videoFrames);
```

```
for f = 1:numFrames
    gray = rgb2gray(videoFrames(:,:,:,f));
    flow = estimateFlow(opticalFlow,gray);

    opticalFlowData(:,:,:,f) = cat(3,flow.Vx,flow.Vy);
end
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
inputFormat(1:numOutputs-1) = "SSCTB";
outputFormat = "CB";
mbq = minibatchqueue(datastore, numOutputs, ...
    "MiniBatchSize", params.MiniBatchSize, ...
    "MiniBatchFcn", @batchVideoAndFlow, ...
    "MiniBatchFormat", [inputFormat,outputFormat]);
end
```

batchVideoAndFlow

The `batchVideoAndFlow` function batches the video, optical flow, and label data from cell arrays. It uses `onehotencode` function to encode ground truth categorical labels into one-hot arrays. The one-hot encoded array contains a `1` in the position corresponding to the class of the label, and `0` in every other position.

```
function [video,flow,labels] = batchVideoAndFlow(video, flow, labels)
% Batch dimension: 5
video = cat(5,video{:});
flow = cat(5,flow{:});

% Batch dimension: 2
labels = cat(2,labels{:});

% Feature dimension: 1
labels = onehotencode(labels,1);
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] = modelGradients

% Pass video input as RGB and optical flow data through the two-stream
% network.
[dlYPredRGB,dlYPredFlow,stateRGB,stateFlow] = forward(i3d,dlRGB,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(rgbLoss,i3d.VideoLearnables);
gradientsFlow = dlgradient(flowLoss,i3d.OpticalFlowLearnables);

% 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
```

`updateLearnables`

The `updateLearnables` function updates the provided `learnables` with gradients and other parameters using SGDM optimization function `sgdmupdate`.

```
function [learnables,velocity,learnRate] = updateLearnables(learnables,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 = learnables.Parameter == "Weights";
    gradients(idx,:) = dlupdate(@(g,w) g + params.L2Regularization*w, gradients(idx,:), learnable

    % Update the network parameters using the SGDM optimizer.
    [learnables, velocity] = sgdmupdate(learnables, gradients, velocity, learnRate, params.Momen
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
```

doValidation

The `doValidation` function validates the video classifier using the validation data.

```
function [validationTime, cmat, lossValidation, accValidation, accValidationRGB, accValidationFl

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(i3d,dlX1,dlX2,dlY);

    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 video classifier for RGB and optical flow data.

```
function [loss,YTest,YPred,YPredRGB,YPredFlow] = predictValidation(i3d,dlRGB,dlFlow,Y)

% Pass the video input through the two-stream Inflated-3D video classifier.
[dlYPredRGB,dlYPredFlow] = predict(i3d,dlRGB,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
```

saveData

The `saveData` function saves the given Inflated-3d Video Classifier, accuracy, loss, and other training parameters to a MAT-file.

```
function bestLoss = saveData(inflated3d,bestLoss,iteration,cmat,lossTrain,lossValidation,...
                accTrain,accValidation,params)
if iteration >= params.SaveBestAfterIteration
    lossValidtion = extractdata(gather(lossValidation));
    if lossValidtion < bestLoss
        params = rmfield(params, 'VelocityRGB');
        params = rmfield(params, 'VelocityFlow');
        bestLoss = lossValidtion;
        inflated3d = gatherFromGPUToSave(inflated3d);
        data.BestLoss = bestLoss;
        data.TrainingLoss = extractdata(gather(lossTrain));
        data.TrainingAccuracy = accTrain;
```

```
        data.ValidationAccuracy = accValidation;
        data.ValidationConfmat= cmat;
        data.inflated3d = inflated3d;
        data.Params = params;
        save(params.ModelFilename, 'data');
    end
end
end
```

gatherFromGPUToSave

The `gatherFromGPUToSave` function gathers data from the GPU in order to save the video classifier to disk.

```
function classifier = gatherFromGPUToSave(classifier)
if ~canUseGPU
    return;
end
p = string(properties(classifier));
p = p(endsWith(p, ["Learnables","State"]));
for jj = 1:numel(p)
    prop = p(jj);
    classifier.(prop) = gatherValues(classifier.(prop));
end
    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 ~isfolder(hmdbFolder)
    error("Download 'hmdb51_org.rar' file using the supporting function 'downloadHMDB51' before r
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 ~isfolder(dataLoc)
    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');

% Download the RAR file and save it to the download folder.
if ~isfile(rarFileName)
    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
```

updateProgressPlot

The `updateProgressPlot` function updates the progress plot with loss and accuracy information during training.

```
function updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accuracyTrain,accurac
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
```

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 Accurac
    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,lossTrain,
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

    function acc = composePadAccuracy(acc)
        acc = compose('%.2f',acc*100) + "%";
        acc = pad(string(acc),6,'left');
    end

end
```

endVerboseOutput

The `endVerboseOutput` function displays the end of verbose output during training.

```
function endVerboseOutput(params)
if params.Verbose
    disp("|========================================================================================
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 Conferencee on Learning Representations 2017*. Toulon, France: ICLR, 2017.

[4] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, Manohar Paluri. "A Closer Look at Spatiotemporal Convolutions for Action Recognition". Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 6450-6459.

[5] Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. "SlowFast Networks for Video Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), 2019.

[6] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, Andrew Zisserman. "The Kinetics Human Action Video Dataset." *arXiv preprint arXiv:1705.06950*, 2017.

Evaluate a Video Classifier

This example shows how to evaluate a pretrained SlowFast Video Classifier using a collection of videos. To learn more about how to train a video classifier network for your dataset, see “Gesture Recognition using Videos and Deep Learning” on page 3-147.

Download Pretrained Video Classifier

Download the pretrained SlowFast video classifier. The size of the downloaded zip file is around 245 MB.

```
downloadFolder = fullfile(tempdir,"gesture");
zipFile = "slowFastPretrained_fourClasses.zip";

if ~isfile(fullfile(downloadFolder,zipFile))
    disp("Downloading the pretrained network...");
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + zipFile;
    zipFile = fullfile(downloadFolder,zipFile);
    websave(zipFile,downloadURL);
    unzip(zipFile,downloadFolder);
    disp('Downloaded.');
end
```

Load the pretrained SlowFast video classifier.

```
pretrainedDataFile = fullfile(downloadFolder,"slowFastPretrained_fourClasses.mat");
pretrained = load(pretrainedDataFile);
slowFastClassifier = pretrained.data.slowFast;
```

Display the class label names of the pretrained video classifier.

```
classes = slowFastClassifier.Classes
```

```
classes = 4×1 categorical
     clapping
     noAction
     somethingElse
     wavingHello
```

Load Videos For Evaluation

This examples uses videos stored in folders with names that correspond to their labels. Extract video sequences of specific scene labels and store them in a folder named `"videoScenes"`. To learn how to extract video sequences from a collection of videos and the corresponding ground truth scene label information, see “Extract Training Data for Video Classification” on page 3-140.

Use the supporting function `downloadGroundTruthData`, defined at the end of this example, to download the video scenes and the corresponding ground truth scene label information.

```
groundTruthFolder = fullfile(downloadFolder,"groundTruthFolder");
trainingFolder = fullfile(downloadFolder,"videoScenes");
groundTruthAlreadyExtracted = downloadGroundTruthData(groundTruthFolder,trainingFolder,classes);
```

Use the supporting function `extractVideoScenes`, defined at the end of this example, to extract the video scenes from ground truth scene label information. Extract the video scene data only when the ground truth data was not extracted before.

```
if ~groundTruthAlreadyExtracted
    extractVideoScenes(groundTruthFolder,trainingFolder,classes);
end
```

Classify Video Files

Use `folders2labels` to create label information from folders and to list all the files in the `"videoScenes"` folder.

```
[trueLabels, filenames] = folders2labels(trainingFolder,...
    "FileExtensions",".avi", ...
    "IncludeSubfolders",true);
```

Pre-allocate categorical label values with empty label names.

```
numFiles = numel(filenames);
emptyLabelNames = repmat("",numFiles,1);
predictedLabels = categorical(emptyLabelNames,string(classes));
```

Choose 5 randomly selected video sequences to classify the video, to uniformly cover the entirety of a file to find the action class that is predominant in the video.

```
numSequences = 5;
```

In order to classify and assign a label to each of the video files in the collection, use `classifyVideoFile` function.

```
for ii = 1:numFiles
    filename = filenames(ii);
    predictedLabels(ii) = classifyVideoFile(slowFastClassifier, filename,...
        "NumSequences", numSequences);
end
```

Compute Video Accuracy

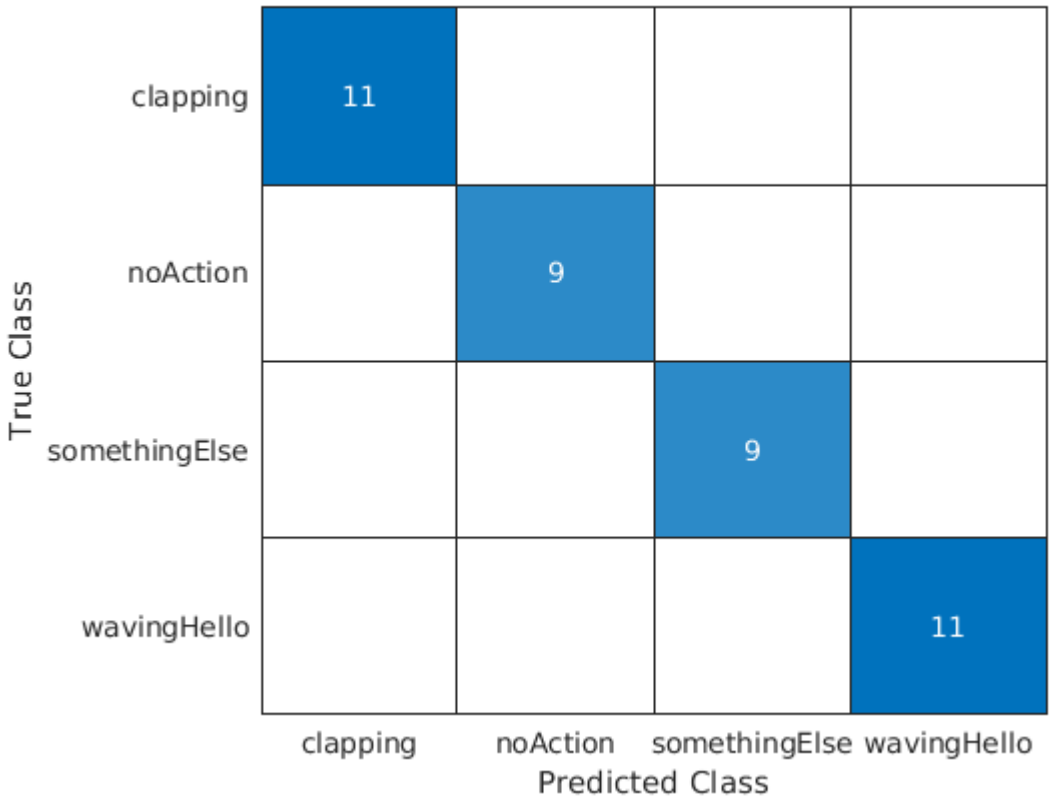
Compute the average video accuracy for the video classifier.

```
videoAccuracy = sum(trueLabels == predictedLabels)/numel(trueLabels)
```

```
videoAccuracy = 1
```

Display the confusion matrix.

```
figure
chart = confusionchart(trueLabels,predictedLabels);
```



Supporting Functions

downloadGroundTruthData

The `downloadGroundTruthData` function downloads ground truth video data and its corresponding collection of scene labels.

```
function groundTruthAlreadyExtracted = downloadGroundTruthData(groundTruthFolder,trainingFolder,c

% If the video scenes are already extracted, no need to download
% the dataset and extract video scenes.
if isfolder(trainingFolder)
    classFolders = fullfile(trainingFolder, string(classes));
    groundTruthAlreadyExtracted = true;
    for ii = 1:numel(classFolders)
        if ~isfolder(classFolders(ii))
            groundTruthAlreadyExtracted = false;
            break;
        end
    end
    if groundTruthAlreadyExtracted
        return;
    end
end
if ~isfolder(groundTruthFolder)
    mkdir(groundTruthFolder);
end
```

```
downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/videoClipsAndSceneLabels.zip";
filename = fullfile(groundTruthFolder,"videoClipsAndSceneLabels.zip");
if ~isfile(filename)
    disp("Downloading the video clips and the corresponding scene labels to " + groundTruthFolde
    websave(filename,downloadURL);
end

% Unzip the contents to the download folder.
unzip(filename,groundTruthFolder);
end
```

`extractVideoScenes`

The `extractVideoScenes` function extracts training video data from a collection of videos and its corresponding collection of scene labels, by using the functions `sceneTimeRanges` and `writeVideoScenes`.

```
function extractVideoScenes(groundTruthFolder,trainingFolder)
labelDataFiles = dir(fullfile(groundTruthFolder,"*_labelData.mat"));
labelDataFiles = fullfile(groundTruthFolder,{labelDataFiles.name}');
numGtruth = numel(labelDataFiles);

% Load the label data information and create ground truth objects.
gTruth = groundTruth.empty(numGtruth,0);
for ii = 1:numGtruth
    ld = load(labelDataFiles{ii});
    videoFilename = fullfile(groundTruthFolder,ld.videoFilename);
    gds = groundTruthDataSource(videoFilename);
    gTruth(ii) = groundTruth(gds,ld.labelDefs,ld.labelData);
end

% Gather all the scene time ranges and the corresponding scene labels
% using the sceneTimeRanges function.
[timeRanges, sceneLabels] = sceneTimeRanges(gTruth);

% Specify the subfolder names for each duration as the scene label names.
foldernames = sceneLabels;

% Delete the folder if it already exists.
if isfolder(trainingFolder)
    rmdir(trainingFolder, 's');
end
% Video files are written to the folders specified by the folderNames input.
writeVideoScenes(gTruth,timeRanges,trainingFolder,foldernames);
end
```

Extract Training Data for Video Classification

This example shows how to extract labeled scenes from a collection of videos where each video contains multiple scene labels. The extracted scenes and their associated labels can be used for training or validating a video classifier. For more information on scene labels, see “Get Started with the Video Labeler” on page 9-44.

Download Training Videos and Scene Labels

This example uses a small collection of video files that were labeled using the Video Labeler app. Specify a location to store the videos and scene label data.

```
downloadFolder = fullfile(tempdir,'sceneLabels');
if ~isfolder(downloadFolder)
    mkdir(downloadFolder);
end
```

Download the training data using `websave` and unzip the contents to the `downloadFolder`.

```
downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/videoClipsAndSceneLabels.zip";

filename = fullfile(downloadFolder,"videoClipsAndSceneLabels.zip");
if ~exist(filename,'file')
    disp("Downloading the video clips and the corresponding scene labels to " + downloadFolder);
    websave(filename,downloadURL);
end

% Unzip the contents to the download folder.
unzip(filename,downloadFolder);
```

Create a groundTruth objects to represent the labeled video files using the supporting function, `createGroundTruthForVideoCollection`, listed at the end of this example.

```
gTruth = createGroundTruthForVideoCollection(downloadFolder);
```

Gather Video Scene Time Ranges and Labels

Gather all the scene time ranges and the corresponding scene labels using the `sceneTimeRanges` function.

```
[timeRanges, sceneLabels] = sceneTimeRanges(gTruth);
```

Here `timeRanges` and `sceneLabels` are M-by-1 cell arrays, where M is the number of ground truth objects. Each time range is a T-by-2 duration matrix, where T is the number of time ranges. Each row of the matrix corresponds to a time range in the ground truth data where a scene label was applied, specified in the form `[rangeStart, rangeEnd]`. For example, the first ground truth object corresponding to video file `video_0001.avi`, contains 4 scenes with labels `"noAction"`, `"wavingHello"`, `"clapping"`, and `"somethingElse"`.

```
[~,name,ext] = fileparts(string(gTruth(1).DataSource.Source));
firstVideoFilename = name + ext
```

```
firstVideoFilename =
"video_0001.avi"
```

```
firstTimeRange = timeRanges{1}
```

```
firstTimeRange = 4×2 duration
       15 sec   28.033 sec
      7.3 sec   15.033 sec
        0 sec   7.3333 sec
       28 sec   37.033 sec
```

```
firstSceneLabel = sceneLabels{1}
```

```
firstSceneLabel = 4×1 categorical
     noAction
     wavingHello
     clapping
     somethingElse
```

Write Extracted Video Scenes

Use `writeVideoScenes` to write the extracted video scenes to disk and organize the written files based on the labels. Saving video files to folders with scene labels as names helps with obtaining the label information easily when training a video classifier. To learn more about training a video classifier using the extracted video data, see “Gesture Recognition using Videos and Deep Learning” on page 3-147.

Select a folder in the download folder to write video scenes.

```
rootFolder = fullfile(downloadFolder,"videoScenes");
```

Video files are written to the folders specified by the `folderNames` input. Use the scene label names as folder names.

```
folderNames = sceneLabels;
```

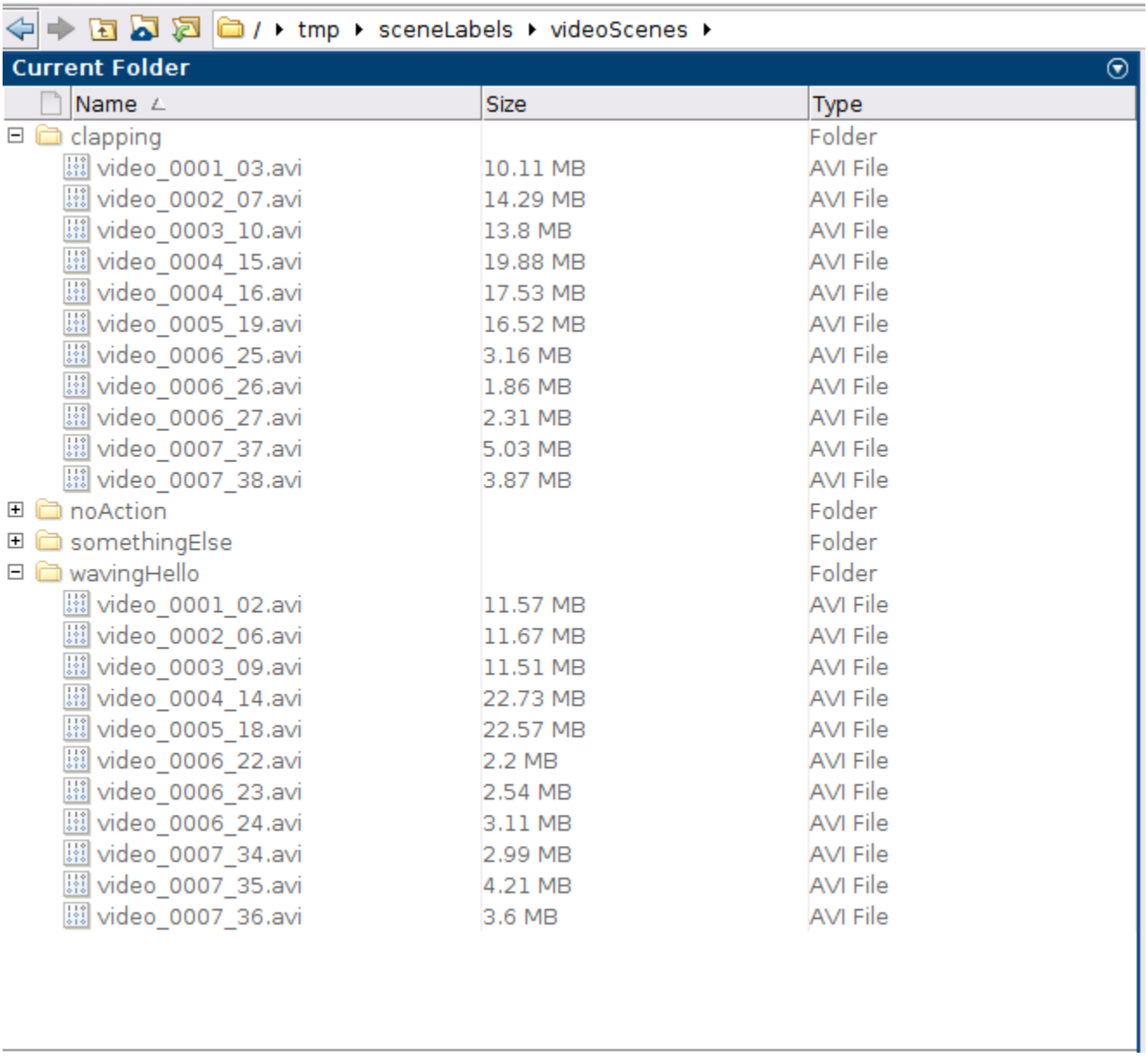
Write video scenes to the `"videoScenes"` folder.

```
filenames = writeVideoScenes(gTruth,timeRanges,rootFolder,folderNames);
```

```
[==================================================] 100%
Elapsed time: 00:01:47
Estimated time remaining: 00:00:00
```

The output `filenames` is an M-by-1 cell array of character strings that specifies the full path to the saved video scenes in each groundTruth object.

Note that the video files corresponding to a scene label are written to folders named by the scene label. For example, video scenes corresponding to the scene label `"clapping"` are written to the folder `"videoScenes/clapping"`, and video scenes corresponding to the scene label `"wavingHello"` are written to the folder `"videoScenes/wavingHello"`.



The extracted video scenes can now be used for training and validating a video classifier. For more information about using the extracted data for training a video classifier, see “Gesture Recognition using Videos and Deep Learning” on page 3-147. For more information about using the extracted data for evaluating a video classifier, see “Evaluate a Video Classifier” on page 3-136.

Supporting Functions

`createGroundTruthForVideoCollection`

The `createGroundTruthForVideoCollection` function creates ground truth data for a given collection of videos and the corresponding label information.

```
function gTruth = createGroundTruthForVideoCollection(downloadFolder)
labelDataFiles = dir(fullfile(downloadFolder,"*_labelData.mat"));
```

```
labelDataFiles = fullfile(downloadFolder,{labelDataFiles.name}');
numGtruth = numel(labelDataFiles);

%Load the label data information and create ground truth objects.
gTruth = groundTruth.empty(numGtruth,0);
for ii = 1:numGtruth
    ld = load(labelDataFiles{ii});
    videoFilename = fullfile(downloadFolder,ld.videoFilename);
    gds = groundTruthDataSource(videoFilename);
    gTruth(ii) = groundTruth(gds,ld.labelDefs,ld.labelData);
end
end
```

Classify Streaming Webcam Video Using SlowFast Video Classifier

This example shows how to classify a streaming video from a webcam using a pretrained SlowFast Video Classifier. To learn more about how to train a video classifier network for your dataset, see “Gesture Recognition using Videos and Deep Learning” on page 3-147.

Download Pretrained Video Classifier

Download the pretrained SlowFast video classifier.

```
downloadFolder = fullfile(tempdir,"gesture");
zipFile = "slowFastPretrained_fourClasses.zip";
if ~isfile(fullfile(downloadFolder,zipFile))
    disp("Downloading the pretrained network...");
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + zipFile;
    zipFile = fullfile(downloadFolder,zipFile);
    websave(zipFile,downloadURL);
    unzip(zipFile,downloadFolder);
end
```

Load the pretrained SlowFast video classifier.

```
pretrainedDataFile = fullfile(downloadFolder,"slowFastPretrained_fourClasses.mat");
pretrained = load(pretrainedDataFile);
slowFastClassifier = pretrained.data.slowFast;
```

Display the class label names of the pretrained video classifier. Any gesture such as "clapping" and "wavingHello" on to the webcam will be recognized by the SlowFast Video Classifier.

```
classes = slowFastClassifier.Classes
```

```
classes = 4×1 categorical
     clapping
     noAction
     somethingElse
     wavingHello
```

Setup the Webcam and the Video Player

In this example, a webcam object is used to capture streaming video. A Video Player is used to display the streaming video along with the predicted class.

Create a webcam object using the `webcam` function.

```
cam = webcam;
```

Create a Video Player using `vision.VideoPlayer` function. Make sure to place the Video Player in a position where you can clearly see the streaming video when running the classification.

```
player = vision.VideoPlayer;
```

Classify the Webcam Streaming Video

Specify how frequently the classifier should be applied to incoming video frames.

```
classifyInterval = 10;
```

A value of 10 balances runtime performance against classification performance. Increase this value to improve runtime performance at the cost of missing gestures from the live video stream.

Obtain the sequence length of the SlowFast Video Classifier. Classify only after capturing at least `sequenceLength` number of frames from the webcam.

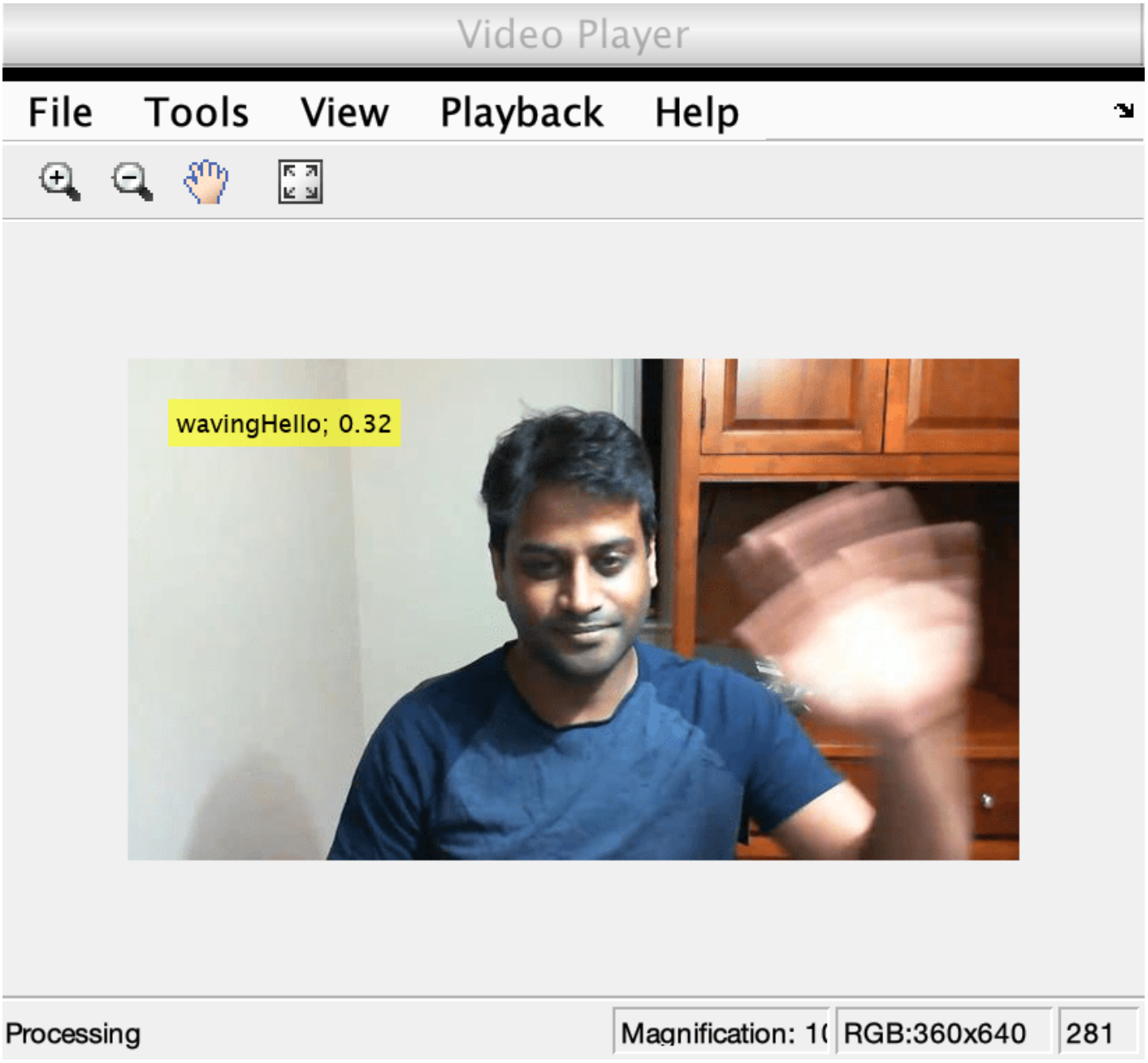
```
sequenceLength = slowFastClassifier.InputSize(4);
```

Specify the maximum number of frames to capture in a loop using the `maxNumFrames` variable. Make sure you wave one of your hands to recognize `"wavingHello"` label, and clap using both your hands for the classifier to recognize `"clapping"` label.

```
maxNumFrames = 280;
```

Capture the webcam snapshot in a loop. Update the streaming video sequence of the classifier using the `updateSequence` method, and classify the streaming sequence using the `classifySequence` method.

```
numFrames = 0;
text = "";

while numFrames <= maxNumFrames
    frame = snapshot(cam);

    numFrames = numFrames + 1;
    slowFastClassifier = updateSequence(slowFastClassifier,frame);
    if mod(numFrames, classifyInterval) == 0 && numFrames >= sequenceLength
        [label,scores] = classifySequence(slowFastClassifier);
        if ~isempty(label)
            text = string(label) + "; " + num2str(max(scores), "%0.2f");
        end
    end
    frame = insertText(frame,[30,30],text,'FontSize',18);
    step(player,frame);
end
```


Video Player
File
Tools
View
Playback
Help
wavingHello; 0.32
Processing
Magnification: 1(
RGB:360x640
281

Gesture Recognition using Videos and Deep Learning

This example first shows how to perform gesture recognition using a pretrained SlowFast [1] on page 3-0 video classifier and then shows how to use transfer learning to train a classifier on a custom gesture recognition data set.

Overview

Vision-based human gesture recognition involves predicting a gesture, such as waving hello, sign language gestures, or clapping, using a set of video frames. One of the appealing features of gesture recognition is that they make it possible for humans to communicate with computers and devices without the need for an external input equipment such as a mouse or a remote control. Gesture recognition from videos has many applications, such as control of consumer electronics and mechanical systems, robot learning, and computer games. For example, online prediction of multiple actions for incoming videos from multiple cameras can be important for robot learning. Compared to image classification, human gesture recognition using videos is challenging to model because of the inaccurate ground truth data for video data sets, the variety of gestures that actors in a video can perform, the heavily class imbalanced data sets, and the large amount of data required to train a robust classifier from scratch. Deep learning techniques, such as SlowFast two pathway convolutional networks [1] on page 3-0 , have shown improved performance on smaller data sets using transfer learning with networks pretrained on large video activity recognition data sets.

Note: This example requires the Computer Vision Toolbox™ Model for SlowFast Video Classification. You can install the Computer Vision Toolbox Model for SlowFast Video Classification from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Perform Gesture Recognition Using a Pretrained Video Classifier

Download the pretrained SlowFast video classifier along with a video file on which to perform gesture recognition. The size of the downloaded zip file is around 245 MB.

```
downloadFolder = fullfile(tempdir,"gesture");
if ~isfolder(downloadFolder)
    mkdir(downloadFolder);
end

zipFile = "slowFastPretrained_fourClasses.zip";

if ~isfile(fullfile(downloadFolder,zipFile))
    disp('Downloading the pretrained network...');
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + zipFile;
    zipFile = fullfile(downloadFolder,zipFile);
    websave(zipFile,downloadURL);
    unzip(zipFile,downloadFolder);
    disp("Downloaded.")
end
```

```
Downloading the pretrained network...

Downloaded.
```

Load the pretrained SlowFast video classifier.

```
pretrainedDataFile = fullfile(downloadFolder,"slowFastPretrained_fourClasses.mat");
pretrained = load(pretrainedDataFile);
slowFastClassifier = pretrained.data.slowFast;
```

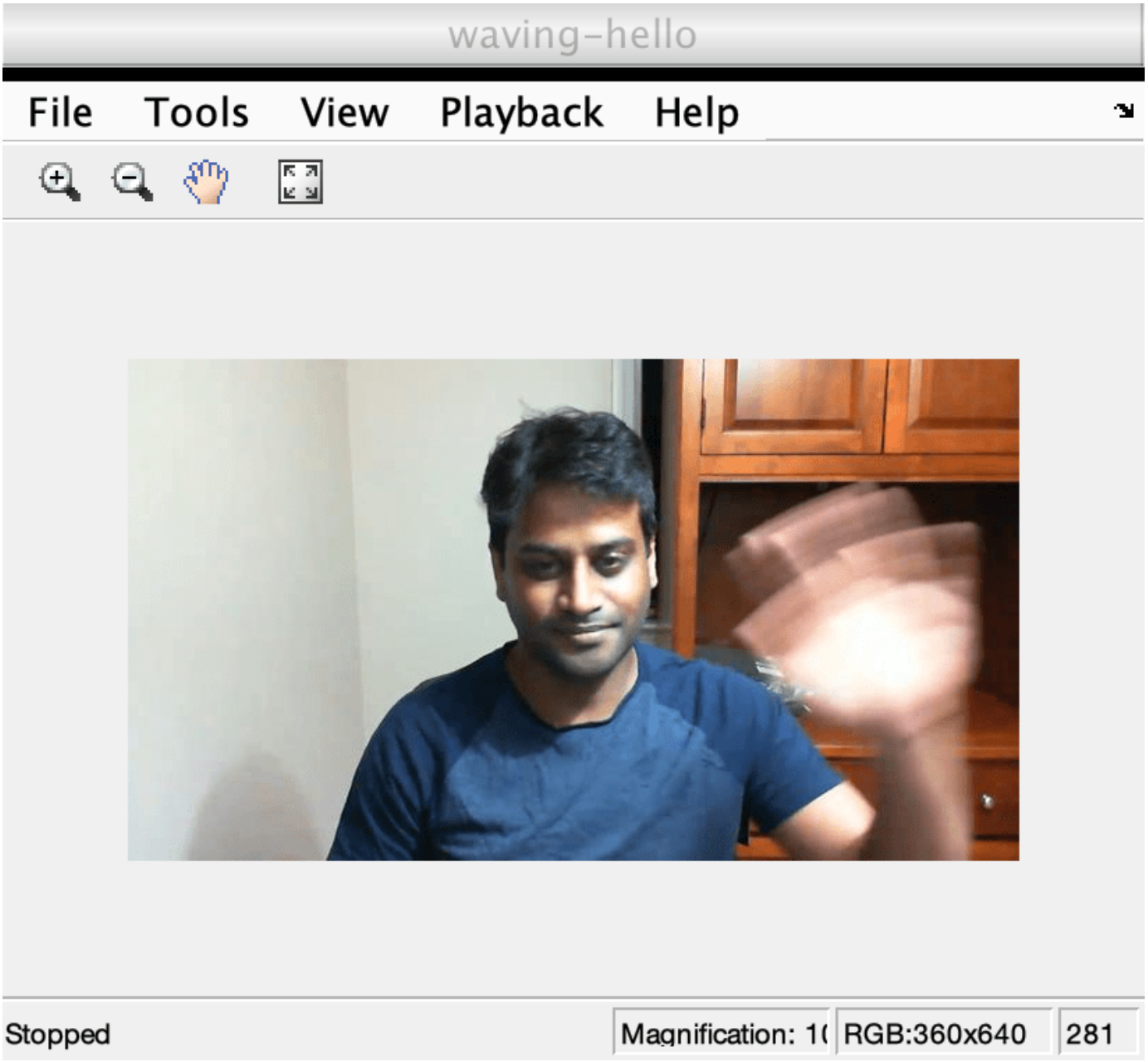
Display the class label names of the pretrained video classifier.

```
classes = slowFastClassifier.Classes
```

```
classes = 4×1 categorical
     clapping
     noAction
     somethingElse
     wavingHello
```

Read and display the video `waving-hello.avi` using `VideoReader` and `vision.VideoPlayer`.

```
videoFilename = fullfile(downloadFolder,"waving-hello.avi");

videoReader = VideoReader(videoFilename);
videoPlayer = vision.VideoPlayer;
videoPlayer.Name = "waving-hello";

while hasFrame(videoReader)
   frame = readFrame(videoReader);
   step(videoPlayer,frame);
end
release(videoPlayer);
```



Choose 10 randomly selected video sequences to classify the video, to uniformly cover the entirety of the file to find the action class that is predominant in the video.

```
numSequences = 10;
```

Classify the video file using the `classifyVideoFile` function.

```
[gestureLabel,score] = classifyVideoFile(slowFastClassifier,videoFilename,NumSequences=numSequenc
```

```
gestureLabel = categorical
     wavingHello
```

```
score = single
    0.4753
```

The classification can also be applied to a streaming video. To learn how to classify a streaming webcam video, see “Classify Streaming Webcam Video Using SlowFast Video Classifier” on page 3-144.

Train a Video Classifier for Gesture Recognition

This section of the example shows how the video classifier shown above is trained using transfer learning. Set the `doTraining` variable to `false` to use the pretrained video classifier without having to wait for training to complete. Alternatively, if you want to train the video classifier, set the `doTraining` variable to `true`.

```
doTraining = false;
```

Download Ground Truth Training Data

This example trains a SlowFast video classification network using downloadable gesture data set that contains four gestures: `"clapping","wavingHello","somethingElse", and "noAction"`. The data set contains videos that are labeled using a Video Labeler and the corresponding ground truth data.

Create directories to store the ground truth training data.

```
groundTruthFolder = fullfile(downloadFolder,"groundTruthFolder");
if ~isfolder(groundTruthFolder)
    mkdir(groundTruthFolder);
end
```

Download the data set and extract the zip archive into the `downloadFolder`.

```
zipFile = 'videoClipsAndSceneLabels.zip';

if ~isfile(fullfile(groundTruthFolder,zipFile))
    disp('Downloading the ground truth training data...');
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + zipFile;
    zipFile = fullfile(groundTruthFolder,zipFile);
    websave(zipFile,downloadURL);
    unzip(zipFile,groundTruthFolder);
end
```

Extract Training Video Sequences

To train a video classifier, you need a collection of videos and its corresponding collection of scene labels. Use the helper function extractVideoScenes, defined at the end of this example, to extract labeled video scenes from the ground truth data and write them to disk as separate video files. To learn more about extracting training data from videos, see “Extract Training Data for Video Classification” on page 3-140.

```
groundTruthFolder = fullfile(downloadFolder,"groundTruthFolder");
trainingFolder = fullfile(downloadFolder,"videoScenes");

extractVideoScenes(groundTruthFolder,trainingFolder,classes);
```

A total of 40 video scenes are extracted from the downloaded ground truth data.

Load data set

This example uses a datastore to read the videos scenes and labels extracted from the ground truth data.

Specify the number of video frames the datastore should be configured to output for each time data is read from the datastore.

```
numFrames = 16;
```

A value of 16 is used here to balance memory usage and classification time. Common values to consider are 8, 16, 32, 64, or 128. Using more frames helps capture additional temporal information, but requires more memory. Empirical analysis is required to determine the optimal number of frames.

Next, specify the height and width of the frames the datastore should be configured to output. The datastore automatically resizes the raw video frames to the specified size to enable batch processing of multiple video sequences.

```
frameSize = [112,112];
```

A value of [112 112] is used to capture longer temporal relationships in the video scene which help classify gestures with long time durations. Common values for the size are [112 112], [224 224], or [256 256]. Smaller sizes enable the use of more video frames at the cost of memory usage, processing time, and spatial resolution. As with the number of frames, empirical analysis is required to determine the optimal values.

Specify the number of channels as 3, as the videos are RGB.

```
numChannels = 3;
```

Use the helper function, `createFileDatastore`, to configure a `FileDatastore` for loading the data. The helper function is listed at the end of this example.

```
isDataForTraining = true;
dsTrain = createFileDatastore(trainingFolder,numFrames,numChannels,classes,isDataForTraining);
```

Configure SlowFast Video Classifier for Transfer Learning

Create a SlowFast video classifier for transfer learning by using the `slowFastVideoClassifier` function. The `slowFastVideoClassifier` function creates a SlowFast video classifier object that is pretrained on the Kinetics-400 data set [2 on page 3-0].

Specify ResNet-50 as the base network convolution neural network 3D architecture for the SlowFast classifier.

```
baseNetwork = "resnet50-3d";
```

Specify the input size for the SlowFast video classifier.

```
inputSize = [frameSize,numChannels,numFrames];
```

Create a SlowFast video classifier by specifying the classes for the gesture data set and the network input size.

```
slowFast = slowFastVideoClassifier(baseNetwork,string(classes),InputSize=inputSize);
```

Specify a model name for the video classifier.

```
slowFast.ModelName = "Gesture Recognizer Using Deep Learning";
```

Augment and Preprocess Training Data

Data augmentation provides a way to use limited data sets for training. Augmentation on video data must be the same for a collection of frames based on the network input size. Minor changes, such as translation, cropping, or transforming an image, provide, new, distinct, and unique images that you can use to train a robust video classifier. Datastores are a convenient way to read and augment collections of data. Augment the training video data by using the `augmentVideo` supporting function, defined at the end of this example.

```
dsTrain = transform(dsTrain,@augmentVideo);
```

Preprocess the training video data to resize to the SlowFast video classifier input size, by using the `preprocessVideoClips`, defined at the end of this example. Specify the `InputNormalizationStatistics` property of the video classifier and input size to the preprocessing function as field values in a struct, `preprocessInfo`. The `InputNormalizationStatistics` property is used to rescale the video frames between 0 and 1, and then normalize the rescaled data using mean and standard deviation. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` as values for `SizingOption` to random crop or center crop the input data to the input size of the video classifier.

```
preprocessInfo.Statistics   = slowFast.InputNormalizationStatistics;
preprocessInfo.InputSize    = inputSize;
preprocessInfo.SizingOption = "resize";

dsTrain = transform(dsTrain,@(data)preprocessVideoClips(data,preprocessInfo));
```

Define Model Gradients Function

The `modelGradients` function, listed at the end of this example, takes as input the SlowFast video classifier `slowFast`, a mini-batch of input data `dlRGB`, 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 classifier, and the mini-batch accuracy of the classifier.

The loss is calculated by computing the cross-entropy loss of the predictions from video classifier. The output predictions of the network are probabilities between 0 and 1 for each of the classes.

$$predictions = forward(slowFast, dlRGB);$$

$$loss = crossentropy(predictions)$$

The accuracy of the classifier is calculated by comparing the classifier `predictions` to the ground truth label of the inputs, `dlY`.

Specify Training Options

Train with a mini-batch size of 5 for 600 iterations. Specify the iteration after which to save the model with the best mini-batch loss by using the `SaveBestAfterIteration` parameter.

Specify the cosine-annealing learning rate schedule [3 on page 3-0] parameters:

- A minimum learning rate of 1e-4.
- A maximum learning rate of 1e-3.

- Cosine number of iterations of 200, 300, and 400, 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:

- 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. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

```
params.Classes = classes;
params.MiniBatchSize = 5;
params.NumIterations = 600;
params.CosineNumIterations = [100 200 300];
params.SaveBestAfterIteration = 400;
params.MinLearningRate = 1e-4;
params.MaxLearningRate = 1e-3;
params.Momentum = 0.9;
params.Velocity = [];
params.L2Regularization = 0.0005;
params.ProgressPlot = false;
params.Verbose = true;
params.DispatchInBackground = true;
params.NumWorkers = 12;
```

Train Video Classifier

Train the SlowFast video classifier using the video data.

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`.
- 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 video data and the labels to `dlarray` objects with the underlying type single.
- To enable processing the time dimension of the the video data using the SlowFast video classifier specify the temporal sequence dimension, `"T"`. Specify the dimension labels `"SSCTB"` (spatial, spatial, channel, temporal, batch) for the video data, and `"CB"` for the label data.

The `minibatchqueue` object uses the supporting function `batchVideo`, listed at the end of this example, to batch the RGB video data.

```
params.ModelFilename = "slowFastPretrained_fourClasses.mat";
if doTraining
    epoch = 1;
    bestLoss = realmax;
    accTrain = [];
    lossTrain = [];

    iteration = 1;
    start = tic;
    trainTime = start;
    shuffled = shuffleTrainDs(dsTrain);

    % Number of outputs is two: One for RGB frames, and one for ground truth labels.
    numOutputs = 2;
    mbq = createMiniBatchQueue(shuffled, numOutputs, params);

    % 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,dlY] = next(mbq);

        % Evaluate the model gradients and loss using dlfeval.
        [gradients,loss,acc,state] = ...
            dlfeval(@modelGradients,slowFast,dlX1,dlY);

        % Accumulate the loss and accuracies.
        lossTrain = [lossTrain, loss];
        accTrain = [accTrain, acc];

        % Update the network state.
        slowFast.State = state;

        % Update the gradients and parameters for the video classifier
        % using the SGDM optimizer.
        [slowFast,params.Velocity,learnRate] = ...
            updateLearnables(slowFast,gradients,params,params.Velocity,iteration);

        if ~hasdata(mbq) || iteration == params.NumIterations
            % Current epoch is complete. Do validation and update progress.
            trainTime = toc(trainTime);

            accTrain = mean(accTrain);
            lossTrain = mean(lossTrain);

            % Update the training progress.
            displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
                accTrain,lossTrain,trainTime);
            updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accTrain);

            % Save the trained video classifier and the parameters, that gave
            % the best training loss so far. Use the saveData supporting function,
```

```
            % listed at the end of this example.
            bestLoss = saveData(slowFast,bestLoss,iteration,lossTrain,params);
        end

        if ~hasdata(mbq) && iteration < params.NumIterations
            % Current epoch is complete. Initialize the training loss, accuracy
            % values, and minibatchqueue for the next epoch.
            accTrain = [];
            lossTrain = [];

            epoch = epoch + 1;
            trainTime = tic;
            shuffled = shuffleTrainDs(dsTrain);
            mbq = createMiniBatchQueue(shuffled, numOutputs, params);
        end

        iteration = iteration + 1;
    end

    % Display a message when training is complete.
    endVerboseOutput(params);

    disp("Model saved to: " + params.ModelFilename);
end
```

Evaluate the Trained Video Classifier

To evaluate the accuracy of the trained SlowFast video classifier, set the `isDataForTraining` variable to false and create a `fileDatastore`. Note that data augmentation is not applied to the evaluation data. Ideally, test and evaluation data should be representative of the original data and is left unmodified for unbiased evaluation.

```
isDataForTraining = false;
dsEval = createFileDatastore(trainingFolder,numFrames,numChannels,classes,isDataForTraining);
dsEval = transform(dsEval,@(data)preprocessVideoClips(data,preprocessInfo));
```

Load the best model saved during training or use the pretrained model.

```
if doTraining
    transferLearned = load(params.ModelFilename);
    slowFastClassifier = transferLearned.data.slowFast;
end
```

Create a `minibatchqueue` object to load batches of the test data.

```
numOutputs = 2;
mbq = createMiniBatchQueue(dsEval,numOutputs,params);
```

For each batch of evaluation data, make predictions using the SlowFast video classifier, and compute the prediction accuracy using a confusion matrix.

```
numClasses = numel(params.Classes);
cmat = sparse(numClasses,numClasses);

while hasdata(mbq)
    [dlVideo,dlY] = next(mbq);

    % Computer the predictions of the trained SlowFast
```

```
    % video classifier.
    dlYPred = predict(slowFastClassifier,dlVideo);
    dlYPred = squeezeIfNeeded(dlYPred,dlY);

    % Aggregate the confusion matrix by using the maximum
    % values of the prediction scores and the ground truth labels.
    [~,YTest] = max(dlY,[],1);
    [~,YPred] = max(dlYPred,[],1);
    cmat = aggregateConfusionMetric(cmat,YTest,YPred);
end
```

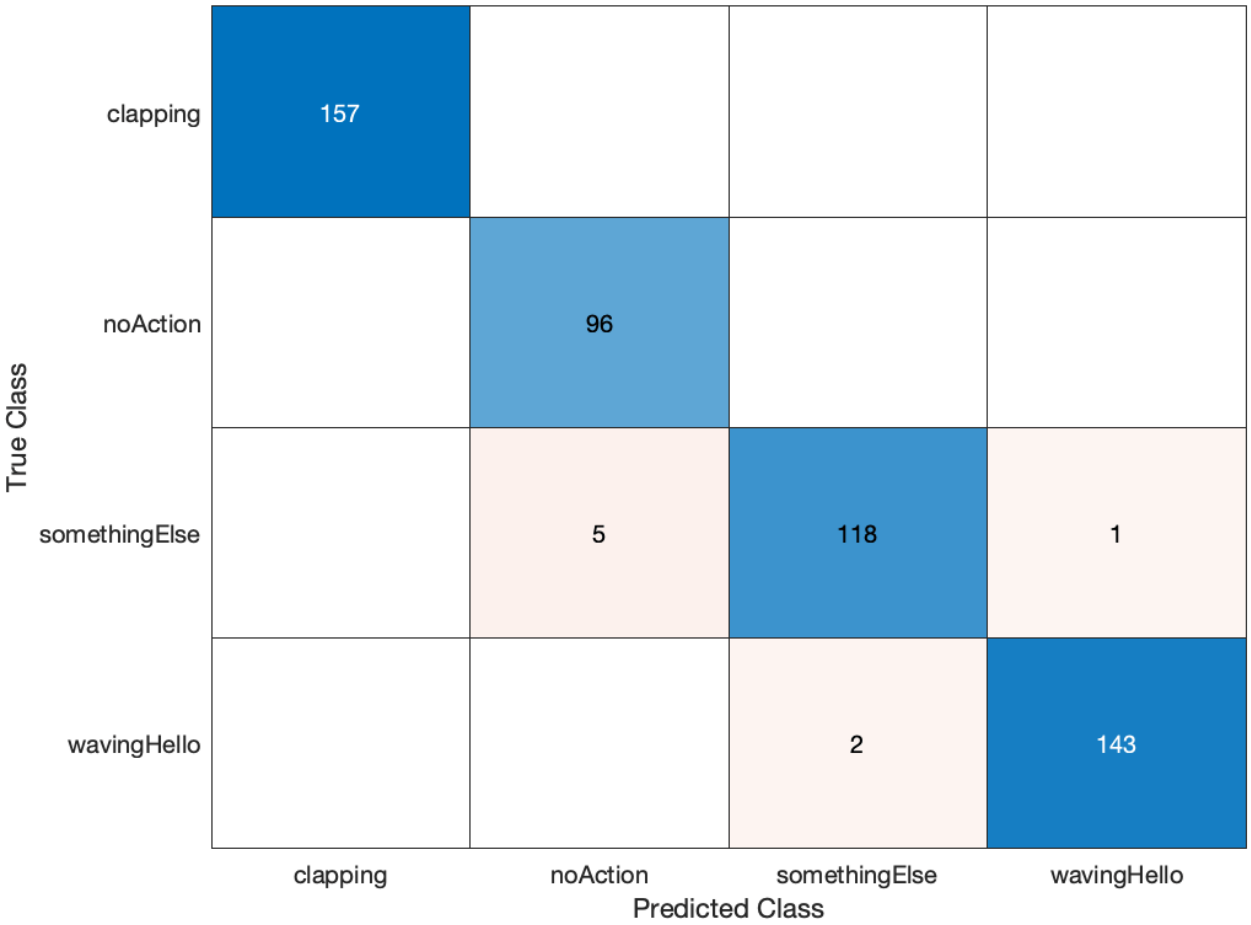
Compute the average clip classification accuracy for the trained SlowFast video classifier.

```
evalClipAccuracy = sum(diag(cmat))./sum(cmat,"all")
```

```
evalClipAccuracy = 0.9847
```

Display the confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```



The SlowFast video classifier that is pretrained on the Kinetics-400 data set [2 on page 3-0], provides strong performance for human gesture recognition on transfer learning. The above training was run on 24GB Titan-X GPU for about 60 minutes. When training from scratch on a small gesture recognition video data set, the training time and convergence takes much longer than the pretrained video classifier. Transer learning using the Kinetics-400 pretrained SlowFast video classifier also avoids overfitting the classifier when ran for larger number of epochs on such a small gesture recognition video data set. To learn more about video recognition using deep learning, see “Getting Started with Video Classification Using Deep Learning” on page 17-4.

Supporting Functions

`createFileDatastore`

The `createFileDatastore` function creates a `FileDatastore` object using the given folder name. 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(trainingFolder,numFrames,numChannels,classes,isDataForT
    readFcn = @(f,u)readVideo(f,u,numFrames,numChannels,classes,isDataForTraining);
    datastore = fileDatastore(trainingFolder,...
        'IncludeSubfolders',true,...
        'FileExtensions','.avi',...
        'ReadFcn',readFcn,...
        'ReadMode','partialfile');
end
```

`shuffleTrainDs`

The `shuffleTrainDs` function shuffles the files present in the training datastore, `dsTrain`.

```
function shuffled = shuffleTrainDs(dsTrain)
shuffled = copy(dsTrain);
transformed = isa(shuffled, 'matlab.io.datastore.TransformedDatastore');
if transformed
    files = shuffled.UnderlyingDatastores{1}.Files;
else
    files = shuffled.Files;
end
n = numel(files);
shuffledIndices = randperm(n);
if transformed
    shuffled.UnderlyingDatastores{1}.Files = files(shuffledIndices);
else
    shuffled.Files = files(shuffledIndices);
end

reset(shuffled);

end
```

`readVideo`

The `readVideo` function reads video frames, and the corresponding 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. During testing, all the frames are sequentially read. The

video frames are resized to the required classifier network input size for training, and for testing and validation.

```
function [data,userdata,done] = readVideo(filename,userdata,numFrames,numChannels,classes,isDataF
    if isempty(userdata)
        userdata.reader      = VideoReader(filename);
        userdata.batchesRead = 0;

        userdata.label = getLabel(filename,classes);

        totalFrames = floor(userdata.reader.Duration * userdata.reader.FrameRate);
        totalFrames = min(totalFrames, userdata.reader.NumFrames);
        userdata.totalFrames = totalFrames;
        userdata.datatype = class(read(userdata.reader,1));
    end
    reader      = userdata.reader;
    totalFrames = userdata.totalFrames;
    label       = userdata.label;
    batchesRead = userdata.batchesRead;

    if isDataForTraining
        video = readForTraining(reader,numFrames,totalFrames);
    else
        video = readForEvaluation(reader,userdata.datatype,numChannels,numFrames,totalFrames);
    end

    data = {video, label};

    batchesRead = batchesRead + 1;

    userdata.batchesRead = batchesRead;

    if numFrames > totalFrames
        numBatches = 1;
    else
        numBatches = floor(totalFrames/numFrames);
    end
    % Set the done flag to true, if the reader has read all the frames or
    % if it is training.
    done = batchesRead == numBatches || isDataForTraining;
end
```

`readForTraining`

The `readForTraining` function reads the video frames for training the video classifier. The function reads the specific number of frames as per the network input size, with a randomly chosen starting frame. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForTraining(reader,numFrames,totalFrames)
    if numFrames >= totalFrames
        startIdx = 1;
        endIdx = totalFrames;
    else
        startIdx = randperm(totalFrames - numFrames + 1);
        startIdx = startIdx(1);
        endIdx = startIdx + numFrames - 1;
    end
```

```
    video = read(reader,[startIdx,endIdx]);
    if numFrames > totalFrames
        % Add more frames to fill in the network input size.
        additional = ceil(numFrames/totalFrames);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

`readForEvaluation`

The `readForEvaluation` function reads the video frames for evaluating the trained video classifier. The function reads the specific number of frames sequentially as per the network input size. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForEvaluation(reader,datatype,numChannels,numFrames,totalFrames)
    H = reader.Height;
    W = reader.Width;
    toRead = min([numFrames,totalFrames]);
    video = zeros([H,W,numChannels,toRead],datatype);
    frameIndex = 0;
    while hasFrame(reader) && frameIndex < numFrames
        frame = readFrame(reader);
        frameIndex = frameIndex + 1;
        video(:,:,:,frameIndex) = frame;
    end

    if frameIndex < numFrames
        video = video(:,:,:,1:frameIndex);
        additional = ceil(numFrames/frameIndex);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

`getLabel`

The `getLabel` function obtains the label name from the full path of a filename. The label for a file is the folder in which it exists. For example, for a file path such as `"/path/to/data set/clapping/video_0001.avi"`, the label name is `"clapping"`.

```
function label = getLabel(filename,classes)
    folder = fileparts(string(filename));
    [~,label] = fileparts(folder);
    label = categorical(string(label),string(classes));
end
```

`augmentVideo`

The `augmentVideo` function augments the video frames for training the video classifier. The function augments a video sequence with the same augmentation technique provided by the `augmentTransform` function.

```
function data = augmentVideo(data)
    numClips = size(data,1);
    for ii = 1:numClips
        video = data{ii,1};
```

```
        % HxWxC
        sz = size(video,[1,2,3]);
        % One augment fcn per clip
        augmentFcn = augmentTransform(sz);
        data{ii,1} = augmentFcn(video);
    end
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
```

preprocessVideoClips

The `preprocessVideoClips` function preprocesses the training video data to resize to the SlowFast video classifier input size. It takes the `InputNormalizationStatistics` and the `InputSize` properties of the video classifier in a struct, `info`. The `InputNormalizationStatistics` property is used to rescale the video frames between 0 and 1, and then normalize the rescaled data using mean and standard deviation. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` as values for `SizingOption` to random crop or center crop the input data to the input size of the video classifier.

```
function data = preprocessVideoClips(data, info)
    inputSize = info.InputSize(1:2);
    sizingOption = info.SizingOption;
    switch sizingOption
        case "resize"
            sizingFcn = @(x)imresize(x,inputSize);
        case "randomcrop"
            sizingFcn = @(x)cropVideo(x,@randomCropWindow2d,inputSize);
        case "centercrop"
            sizingFcn = @(x)cropVideo(x,@centerCropWindow2d,inputSize);
    end
    numClips = size(data,1);

    minValue  = info.Statistics.Min;
    maxValue  = info.Statistics.Max;
    meanValue = info.Statistics.Mean;
    stdValue  = info.Statistics.StandardDeviation;

    minValue  = reshape(minValue,1,1,3);
    maxValue  = reshape(maxValue,1,1,3);
    meanValue = reshape(meanValue,1,1,3);
```

```
    stdValue  = reshape(stdValue,1,1,3);

    for ii = 1:numClips
        video = data{ii,1};
        resized = sizingFcn(video);

        % Cast the input to single.
        resized = single(resized);

        % Rescale the input between 0 and 1.
        resized = rescale(resized,0,1,InputMin=minValue,InputMax=maxValue);

        % Normalize using mean and standard deviation.
        resized = resized - meanValue;
        resized = resized./stdValue;
        data{ii,1} = resized;
    end

    function outData = cropVideo(data,cropFcn,inputSize)
        imsz = size(data,[1,2]);
        cropWindow = cropFcn(imsz,inputSize);
        numBatches = size(data,4);
        sz = [inputSize, size(data,3),numBatches];
        outData = zeros(sz,'like',data);
        for b = 1:numBatches
            outData(:,:,:,b) = imcrop(data(:,:,:,b),cropWindow);
        end
    end
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

inputFormat(1:numOutputs-1) = "SSCTB";
outputFormat = "CB";
mbq = minibatchqueue(datastore, numOutputs, ...
    "MiniBatchSize", params.MiniBatchSize, ...
    "MiniBatchFcn", @batchVideo, ...
    "MiniBatchFormat", [inputFormat,outputFormat]);
end
```

batchVideo

The `batchVideo` function batches the video, and the label data from cell arrays. It uses `onehotencode` function to encode ground truth categorical labels into one-hot arrays. The one-hot encoded array contains a `1` in the position corresponding to the class of the label, and `0` in every other position.

```
function [video,labels] = batchVideo(video,labels)
% Batch dimension: 5
video = cat(5,video{:});

% Batch dimension: 2
labels = cat(2,labels{:});

% Feature dimension: 1
labels = onehotencode(labels,1);
end
```

modelGradients

The `modelGradients` function takes as input a mini-batch of RGB data `dlRGB`, 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,loss,acc,stateRGB] = modelGradients(slowFast,dlRGB,dlY)
[dlYPredRGB,stateRGB] = forward(slowFast,dlRGB);
dlYPred = squeezeIfNeeded(dlYPredRGB,dlY);

loss = crossentropy(dlYPred,dlY);

gradientsRGB = dlgradient(loss,slowFast.Learnables);

% Calculate the accuracy of the predictions.
[~,YTest] = max(dlY,[],1);
[~,YPred] = max(dlYPred,[],1);

acc = gather(extractdata(sum(YTest == YPred)./numel(YTest)));
end
```

squeezeIfNeeded

The `squeezeIfNeeded` function takes as the predicted scores, `dlYPred` and corresponding target `Y`, and returns the predicted scores `dlYPred`, after squeezing the singleton dimensions, if there are any.

```
function dlYPred = squeezeIfNeeded(dlYPred,Y)
if ~isequal(size(Y),size(dlYPred))
    dlYPred = squeeze(dlYPred);
    dlYPred = dlarray(dlYPred,dims(Y));
end
end
```

updateLearnables

The `updateLearnables` function updates the learnable parameters of the SlowFast video classifier with gradients and other parameters using SGDM optimization function `sgdmupdate`.

```
function [slowFast,velocity,learnRate] = updateLearnables(slowFast,gradients,params,velocity,ite
    % Determine the learning rate using the cosine-annealing learning rate schedule.
```

```
    learnRate = cosineAnnealingLearnRate(iteration, params);

    % Apply L2 regularization to the weights.
    learnables = slowFast.Learnables;
    idx = learnables.Parameter == "Weights";
    gradients(idx,:) = dlupdate(@(g,w) g + params.L2Regularization*w,gradients(idx,:),learnables

    % Update the network parameters using the SGDM optimizer.
    [slowFast, velocity] = sgdmupdate(slowFast,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
```

saveData

The `saveData` function saves the given SlowFast video classifier, loss, and other training parameters to a MAT-file.

```
function bestLoss = saveData(slowFast,bestLoss,iteration,lossTrain,params)
if iteration >= params.SaveBestAfterIteration
    trainingLoss = extractdata(gather(lossTrain));
    if trainingLoss < bestLoss
        bestLoss = trainingLoss;
        slowFast = gatherFromGPUToSave(slowFast);
        data.BestLoss = bestLoss;
```

```
        data.slowFast = slowFast;
        data.Params = params;
        save(params.ModelFilename,'data');
    end
end
end
```

gatherFromGPUToSave

The `gatherFromGPUToSave` function gathers data from the GPU in order to save the model to disk.

```
function slowfast = gatherFromGPUToSave(slowfast)
if ~canUseGPU
    return;
end
slowfast.Learnables = gatherValues(slowfast.Learnables);
slowfast.State = gatherValues(slowfast.State);
    function tbl = gatherValues(tbl)
        for ii = 1:height(tbl)
            tbl.Value{ii} = gather(tbl.Value{ii});
        end
    end
end
```

extractVideoScenes

The `extractVideoScenes` function extracts training video data from a collection of videos and its corresponding collection of scene labels, by using the functions `sceneTimeRanges` and `writeVideoScenes`.

```
function extractVideoScenes(groundTruthFolder,trainingFolder,classes)
% If the video scenes are already extracted, no need to download
% the data set and extract video scenes.
if isfolder(trainingFolder)
    classFolders = fullfile(trainingFolder,string(classes));
    allClassFoldersFound = true;
    for ii = 1:numel(classFolders)
        if ~isfolder(classFolders(ii))
            allClassFoldersFound = false;
            break;
        end
    end
    if allClassFoldersFound
        return;
    end
end
if ~isfolder(groundTruthFolder)
    mkdir(groundTruthFolder);
end
downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/videoClipsAndSceneLabels.zip";

filename = fullfile(groundTruthFolder,"videoClipsAndSceneLabels.zip");
if ~exist(filename,'file')
    disp("Downloading the video clips and the corresponding scene labels to " + groundTruthFolde
    websave(filename,downloadURL);
end
% Unzip the contents to the download folder.
unzip(filename,groundTruthFolder);
```

```
labelDataFiles = dir(fullfile(groundTruthFolder,"*_labelData.mat"));
labelDataFiles = fullfile(groundTruthFolder,{labelDataFiles.name}');
numGtruth = numel(labelDataFiles);
% Load the label data information and create ground truth objects.
gTruth = groundTruth.empty(numGtruth,0);
for ii = 1:numGtruth
    ld = load(labelDataFiles{ii});
    videoFilename = fullfile(groundTruthFolder,ld.videoFilename);
    gds = groundTruthDataSource(videoFilename);
    gTruth(ii) = groundTruth(gds,ld.labelDefs,ld.labelData);
end
% Gather all the scene time ranges and the corresponding scene labels
% using the sceneTimeRanges function.
[timeRanges, sceneLabels] = sceneTimeRanges(gTruth);
% Specify the subfolder names for each duration as the scene label names.
foldernames = sceneLabels;
% Delete the folder if it already exists.
if isfolder(trainingFolder)
    rmdir(trainingFolder,'s');
end
% Video files are written to the folders specified by the folderNames input.
writeVideoScenes(gTruth,timeRanges,trainingFolder,foldernames);
end
```

initializeTrainingProgressPlot

The `initializeTrainingProgressPlot` function configures two plots for displaying the training loss, and the training 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');
    legend('Training Accuracy','Location','northwest');
    xlabel("Iteration")
    ylabel("Accuracy")
else
    plotters = [];
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)
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);
    drawnow
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 | Mini-Batch |  Base Learning  | Train
    disp("|       |           |  (hh:mm:ss)  |  Accuracy  |    Loss    |      Rate       | (hh:m
    disp("|=========================================================================================
end
end
```

displayVerboseOutputEveryEpoch

The `displayVerboseOutputEveryEpoch` function displays the verbose output of the training values, such as the epoch, mini-batch accuracy, and mini-batch loss.

```
function displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
                accTrain,lossTrain,trainTime)
    if params.Verbose
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        trainTime = duration(0,0,trainTime,'Format','hh:mm:ss');

        lossTrain = gather(extractdata(lossTrain));
        lossTrain = compose('%.4f',lossTrain);

        accTrain = composePadAccuracy(accTrain);

        learnRate = compose('%.13f',learnRate);

        disp("| " + ...
            pad(string(epoch),5,'both') + " | " + ...
            pad(string(iteration),9,'both') + " | " + ...
```

```
            pad(string(D),12,'both') + " | " + ...
            pad(string(accTrain),10,'both') + " | " + ...
            pad(string(lossTrain),10,'both') + " | " + ...
            pad(string(learnRate),13,'both') + " | " + ...
            pad(string(trainTime),10,'both') + " |")
    end

    function acc = composePadAccuracy(acc)
        acc = compose('%.2f',acc*100) + "%";
        acc = pad(string(acc),6,'left');
    end

end
```

`endVerboseOutput`

The `endVerboseOutput` function displays the end of verbose output during training.

```
function endVerboseOutput(params)
if params.Verbose
    disp("|=======================================================================================
end
end
```

References

[1] Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. "SlowFast Networks for Video Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), 2019.

[2] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, Andrew Zisserman. "The Kinetics Human Action Video data set." *arXiv preprint arXiv:1705.06950*, 2017.

[3] Loshchilov, Ilya, and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts." *International Conferencee on Learning Representations 2017*. Toulon, France: ICLR, 2017.

Explore Semantic Segmentation Network Using Grad-CAM

This example shows how to explore the predictions of a semantic segmentation network using Grad-CAM.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. You can use Grad-CAM, a deep learning visualization technique, to see which regions of the image are important for the pixel classification decision.

Load Data Set

This example uses the CamVid data set [1] from the University of Cambridge for training. This data set is a collection of images containing street-level views obtained while driving. The data set provides pixel-level labels for 32 semantic classes, including car, pedestrian, and road.

Download CamVid Data Set

Download the CamVid data set.

```
rng("default")

imageURL = "http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.zi
labelURL = "http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.z

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 data set labels...");
    websave(labelsZip, labelURL);
    unzip(labelsZip, fullfile(outputFolder,"labels"));

    disp("Downloading 557 MB CamVid data set images...");
    websave(imagesZip, imageURL);
    unzip(imagesZip, fullfile(outputFolder,"images"));
end
```

```
Downloading 16 MB CamVid data set labels...

Downloading 557 MB CamVid data set images...
```

Load CamVid Images

Use an `imageDatastore` to load the 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);
```

The data set contains 32 classes. To make training easier, reduce the number of classes to 11 by grouping multiple classes from the original data set together. For example, create a "`Car`" class that combines the "`Car`", "`SUVPickupTruck`", "`Truck_Bus`", "`Train`", and "`OtherMoving`" classes from the original data set. Return the grouped label IDs by using the supporting function `camvidPixelLabelIDs`, which is listed at the end of this example.

```
classes = [
    "Sky"
    "Building"
    "Pole"
    "Road"
    "Pavement"
    "Tree"
    "SignSymbol"
    "Fence"
    "Car"
    "Pedestrian"
    "Bicyclist"
    ];

labelIDs = camvidPixelLabelIDs;
```

Use the classes and label IDs to create a `pixelLabelDatastore`.

```
labelDir = fullfile(outputFolder,"labels");
pxds = pixelLabelDatastore(labelDir,classes,labelIDs);
```

Load Pretrained Semantic Segmentation Network

Load a pretrained semantic segmentation network. The pretrained model allows you to run the entire example without having to wait for training to complete. This example loads a trained Deeplab v3+ network with weights initialized from a pretrained ResNet-18 network. To get a pretrained ResNet-18, install `resnet18` (Deep Learning Toolbox). For more information on building and training a semantic segmentation network, see “Semantic Segmentation Using Deep Learning” (Deep Learning Toolbox).

```
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
pretrainedNet = load(pretrainedNetwork);
net = pretrainedNet.net;
```

Test Network

The trained semantic segmentation network predicts the label of each pixel within an image. You can test the network by predicting the pixel labels of an image.

Load a test image.

```
figure
img = readimage(imds,615);
imshow(img,InitialMagnification=35)
```

Use the `semanticseg` function to predict the pixel labels of the image by using the trained semantic segmentation network.

```
predLabels = semanticseg(img,net);
```

Display the results.

```
cmap = camvidColorMap;
segImg = labeloverlay(img,predLabels,Colormap=cmap,Transparency=0.4);
figure
imshow(segImg,InitialMagnification=40)

pixelLabelColorbar(cmap,classes)
```

You can see that the network labels the parts of the image fairly accurately. The network does misclassify some areas, for example, the road to the left of the intersection, which is partially misclassified as pavement.

Explore Network Predictions

Deep networks are complex, so understanding how a network determines a particular prediction is difficult. You can use Grad-CAM to see which areas of the test image the semantic segmentation network is using to make its pixel classifications.

Grad-CAM computes the gradient of a differentiable output, such as class score, with respect to the convolutional features in a chosen layer. Grad-CAM is typically used for image classification tasks [2]; however, it can also be extended to semantic segmentation problems [3].

In semantic segmentation tasks, the softmax layer of the network outputs a score for each class for every pixel in the original image. This contrasts with standard image classification problems, where the softmax layer outputs a score for each class for the entire image. The Grad-CAM map for class c is

$$M^c = ReLU(\sum_k \alpha_c^k A^k) \text{ where } \alpha_c^k = 1/N \sum_{i,j} \frac{dy^c}{dA_{i,j}^k}$$

N is the number of pixels, A^k is the feature map of interest, and y^c corresponds to a scalar class score. For a simple image classification problem, y^c is the softmax score for the class of interest. For semantic segmentation, you can obtainy^c by reducing the pixel-wise class scores for the class of interest to a scalar. For example, sum over the spatial dimensions of the softmax layer:

$y^c = \sum_{(i,j) \in P} y^c_{i,j}$, where P is the pixels in the output layer of a semantic segmentation network [3]. In this example, the output layer is the softmax layer before the pixel classification layer. The map M^c highlights areas that influence the decision for class c. Higher values indicate regions of the image that are important for the pixel classification decision.

To use Grad-CAM, you must select a feature layer to extract the feature map from and a reduction layer to extract the output activations from. Use `analyzeNetwork` to find the layers to use with Grad-CAM.

```
analyzeNetwork(net)
```

Specify a feature layer. Typically this is a ReLU layer which takes the output of a convolutional layer at the end of the network.

```
featureLayer = "dec_relu4";
```

Specify a reduction layer. The `gradCAM` function sums the spatial dimensions of the reduction layer, for the specified classes, to produce a scalar value. This scalar value is then differentiated with respect to each feature in the feature layer. For semantic segmentation problems, the reduction layer is usually the softmax layer.

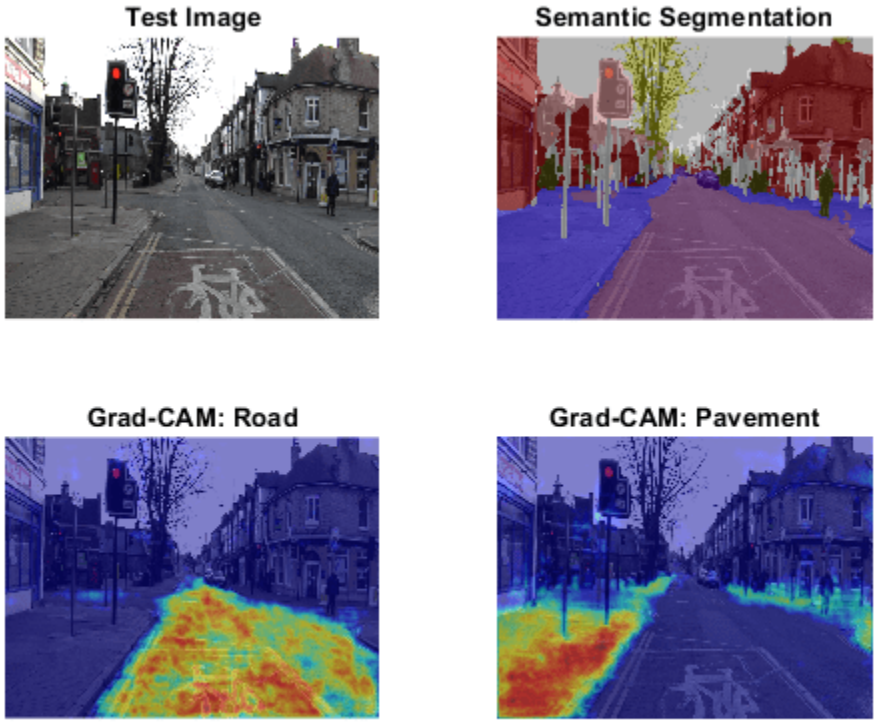
```
reductionLayer = "softmax-out";
```

Compute the Grad-CAM map for the road and pavement classes.

```
classes = ["Road" "Pavement"];

gradCAMMap = gradCAM(net,img,classes, ...
    ReductionLayer=reductionLayer, ...
    FeatureLayer=featureLayer);
```

Compare the Grad-CAM map for the two classes to the semantic segmentation map.

```
predLabels = semanticseg(img,net);
segMap = labeloverlay(img,predLabels,Colormap=cmap,Transparency=0.4);

figure;
subplot(2,2,1)
imshow(img)
title("Test Image")
subplot(2,2,2)
imshow(segMap)
title("Semantic Segmentation")
subplot(2,2,3)
imshow(img)
hold on
imagesc(gradCAMMap(:,:,1),AlphaData=0.5)
title("Grad-CAM: " + classes(1))
colormap jet
subplot(2,2,4)
imshow(img)
hold on
imagesc(gradCAMMap(:,:,2),AlphaData=0.5)
title("Grad-CAM: " + classes(2))
colormap jet
```



The Grad-CAM maps and semantic segmentation map show similar highlighting. None of the maps distinguish the road to the left of the intersection, which the semantic segmentation map labels as pavement. The Grad-CAM map for the pavement class shows that the edge of the pavement is more important than the center for the classification decision of the network. The network possibly misclassifies the road to the left of the intersection due to the poor visibility of the pavement edge.

Explore Intermediate Layers

The Grad-CAM map resembles the semantic segmentation map when you use a layer near the end of the network for the computation. You can also use Grad-CAM to investigate intermediate layers in the trained network. Earlier layers have a small receptive field size and learn small, low-level features compared to the layers at the end of the network.

Compute the Grad-CAM map for layers that are successively deeper in the network.

```
layers = ["res5b_relu","catAspp","dec_relu1"];
numLayers = length(layers);
```

The `res5b_relu` layer is near the middle of the network, whereas `dec_relu1` is near the end of the network.

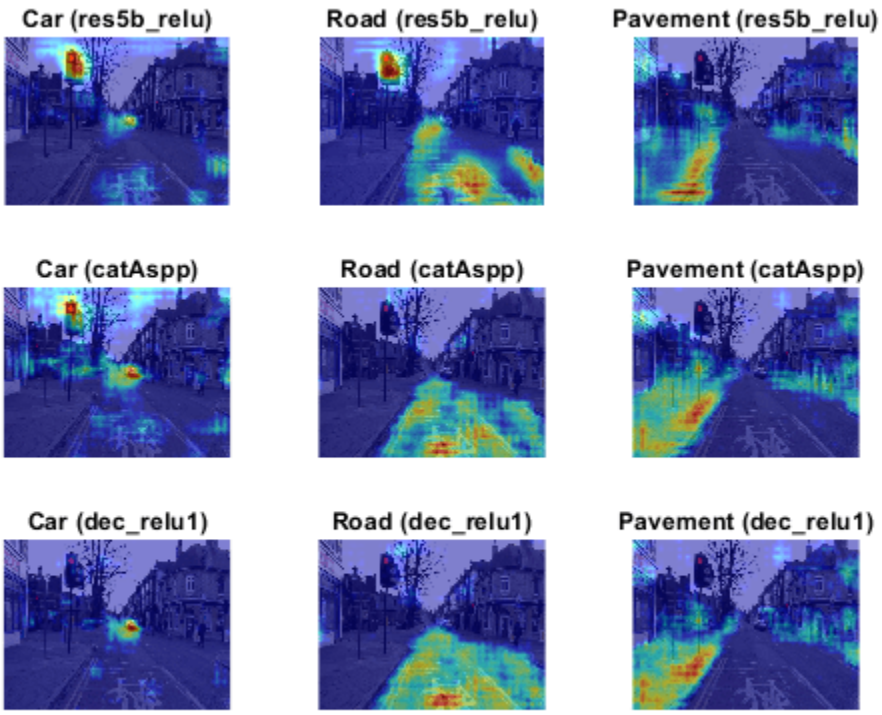
Investigate the network classification decisions for the car, road, and pavement classes. For each layer and class, compute the Grad-CAM map.

```
classes = ["Car" "Road" "Pavement"];
numClasses = length(classes);
```

```
gradCAMMaps = [];
for i = 1:numLayers
    gradCAMMaps(:,:,:,i) = gradCAM(net,img,classes, ...
        ReductionLayer=reductionLayer, ...
        FeatureLayer=layers(i));
end
```

Display the Grad-CAM maps for each layer and each class. The rows represent the map for each layer, with the layers ordered from those early in the network to those at the end of the network.

```
figure;
idx = 1;
for i=1:numLayers
    for j=1:numClasses
        subplot(numLayers,numClasses,idx)
        imshow(img)
        hold on
        imagesc(gradCAMMaps(:,:,j,i),AlphaData=0.5)
        title(sprintf("%s (%s)",classes(j),layers(i)), ...
            Interpreter="none")
        colormap jet
        idx = idx + 1;
    end
end
```



The later layers produce maps very similar to the segmentation map. However, the layers earlier in the network produce more abstract results and are typically more concerned with lower level

features like edges, with less awareness of semantic classes. For example, in the maps for earlier layers, you can see that for both car and road classes, the traffic light is highlighted. This suggests that the earlier layers focus on areas of the image that are related to the class but do not necessarily belong to it. For example, a traffic light is likely to appear near to a road, so the network might be using this information to predict which pixels are roads. You can also see that for the pavement class, the earlier layers are highly focused on the edge, suggesting this feature is important to the network when detecting which pixels are in the pavement class.

References

[1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. “Semantic Object Classes in Video: A High-Definition Ground Truth Database.” *Pattern Recognition Letters* 30, no. 2 (January 2009): 88–97. https://doi.org/10.1016/j.patrec.2008.04.005.

[2] Selvaraju, R. R., M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization." In IEEE International Conference on Computer Vision (ICCV), 2017, pp. 618–626. Available at `Grad-CAM` on the Computer Vision Foundation Open Access website.

[3] Vinogradova, Kira, Alexandr Dibrov, and Gene Myers. “Towards Interpretable Semantic Segmentation via Gradient-Weighted Class Activation Mapping (Student Abstract).” *Proceedings of the AAAI Conference on Artificial Intelligence* 34, no. 10 (April 3, 2020): 13943–44. https://doi.org/10.1609/aaai.v34i10.7244.

Supporting Functions

```
function labelIDs = camvidPixelLabelIDs()
% Return the label IDs corresponding to each class.
%
% The CamVid data set 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 pixelLabelColorbar(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 a colorbar to the 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 marks.
c.TickLength = 0;
end

function cmap = camvidColorMap
% Define the colormap used by the CamVid data set.

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
```

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.

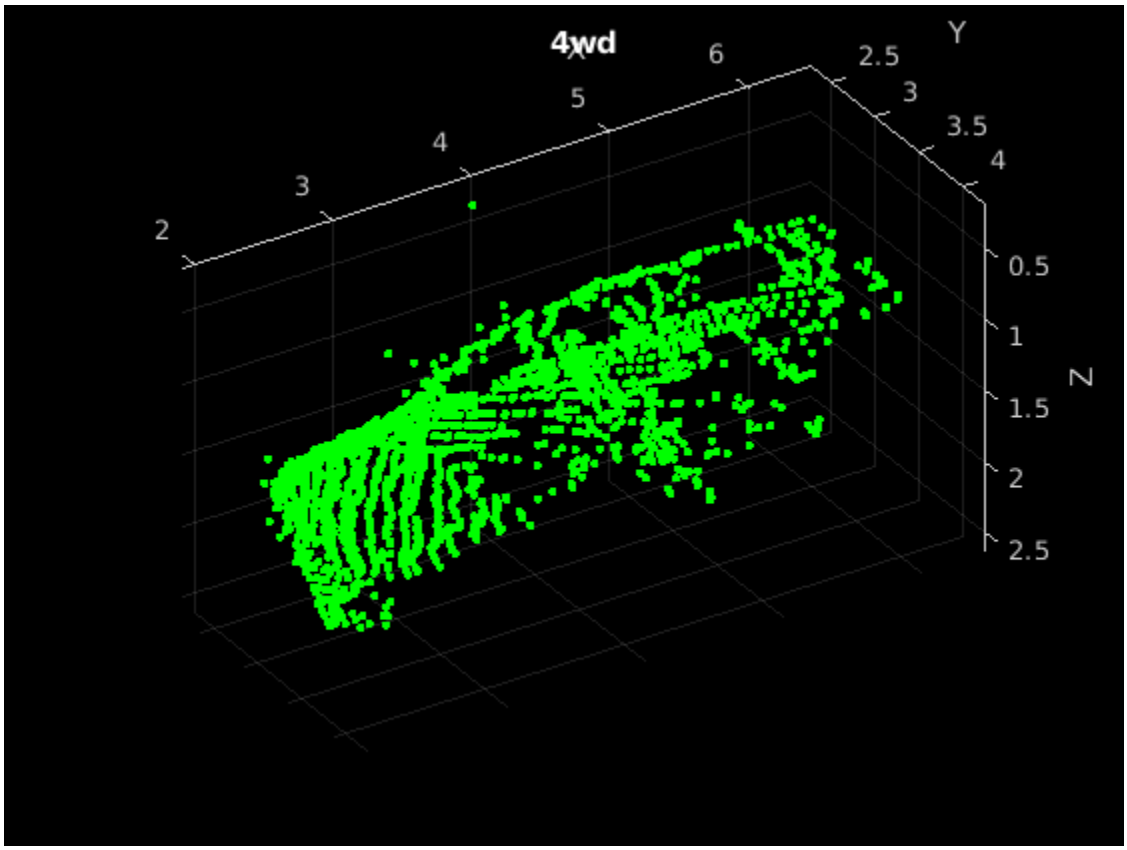
```
downloadDirectory = tempdir;
datapath = downloadSydneyUrbanObjects(downloadDirectory);
```

Load the downloaded training and validation data set using the `loadSydneyUrbanObjectsData` 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 = loadSydneyUrbanObjectsData(datapath,foldsTrain);
dsVal = loadSydneyUrbanObjectsData(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 1 0],"MarkerSize",40,"VerticalAxisDir","down")
xlabel("X")
ylabel("Y")
zlabel("Z")
title(label)
```

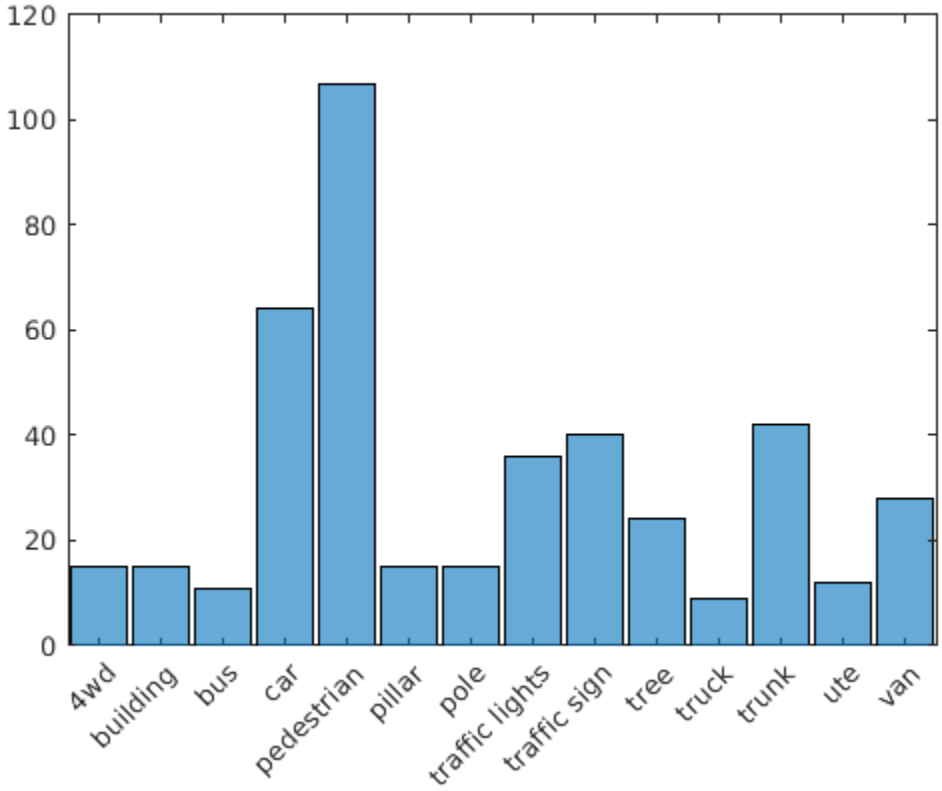


Read the labels and count the number of points assigned to each label to better understand the distribution of labels within the data set.

```
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;
```

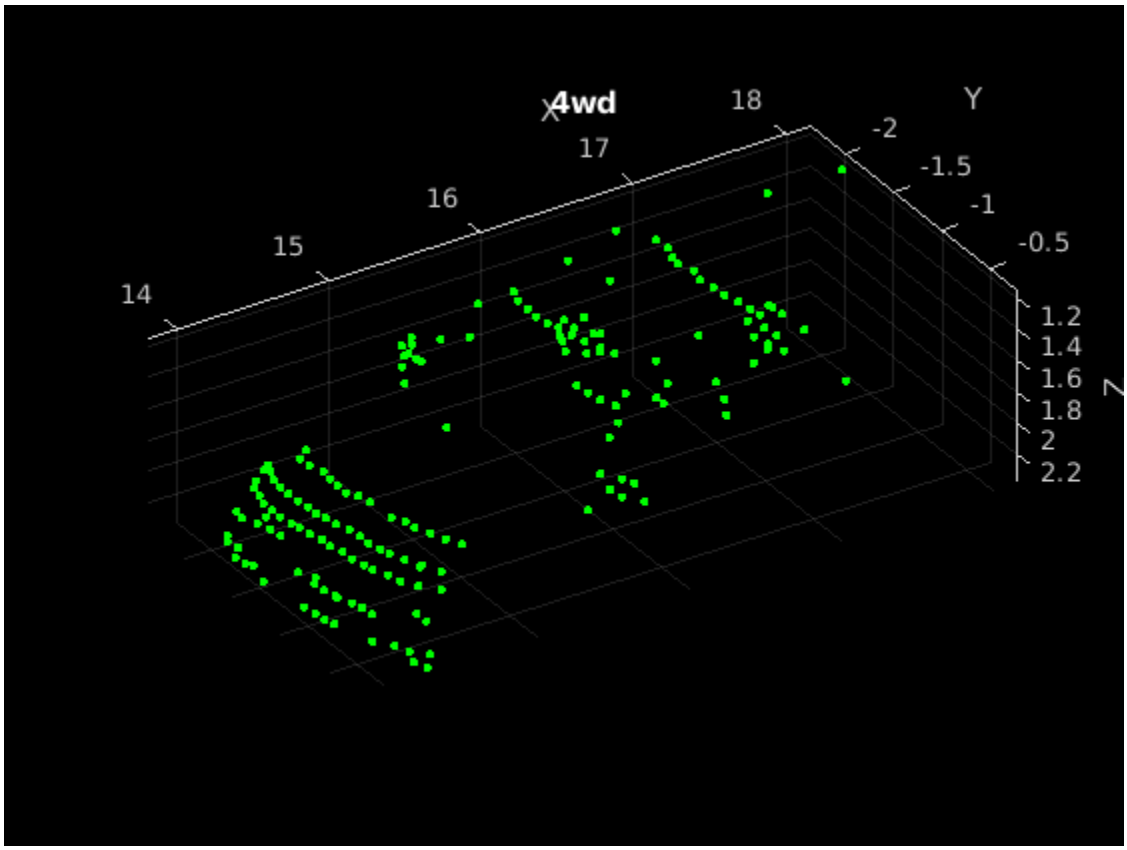
Data Augmentation

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 1 0],"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=14×5 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.

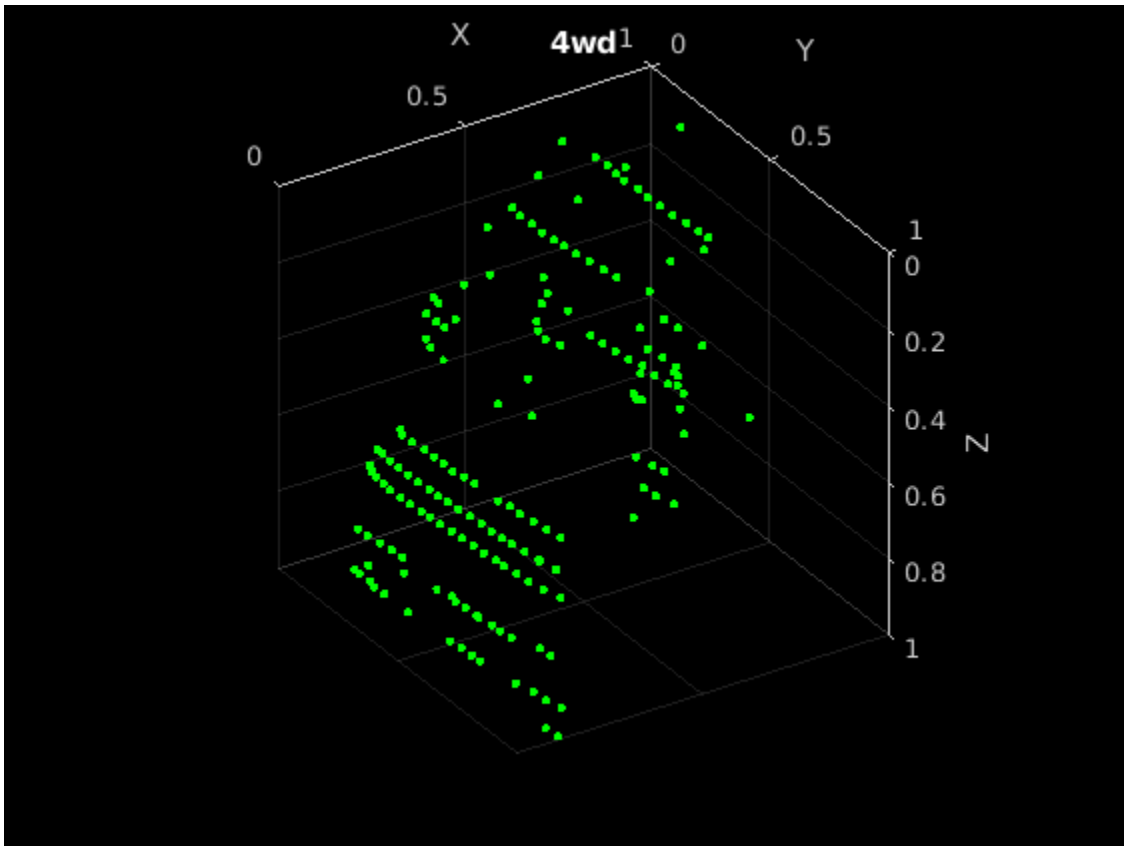
```
numPoints = 1024;
dsTrain = transform(dsTrain,@(data)selectPoints(data,numPoints));
dsVal = transform(dsVal,@(data)selectPoints(data,numPoints));
```

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 1 0],"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,hiddenCh
```

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,hidd
```

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 `initalizeClassifier` 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(inputChanne
```

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 10 epochs and load data in batches of 128. Set the initial learning rate to 0.002 and the L2 regularization factor to 0.01.

```
numEpochs = 10;
learnRate = 0.002;
miniBatchSize = 128;
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.

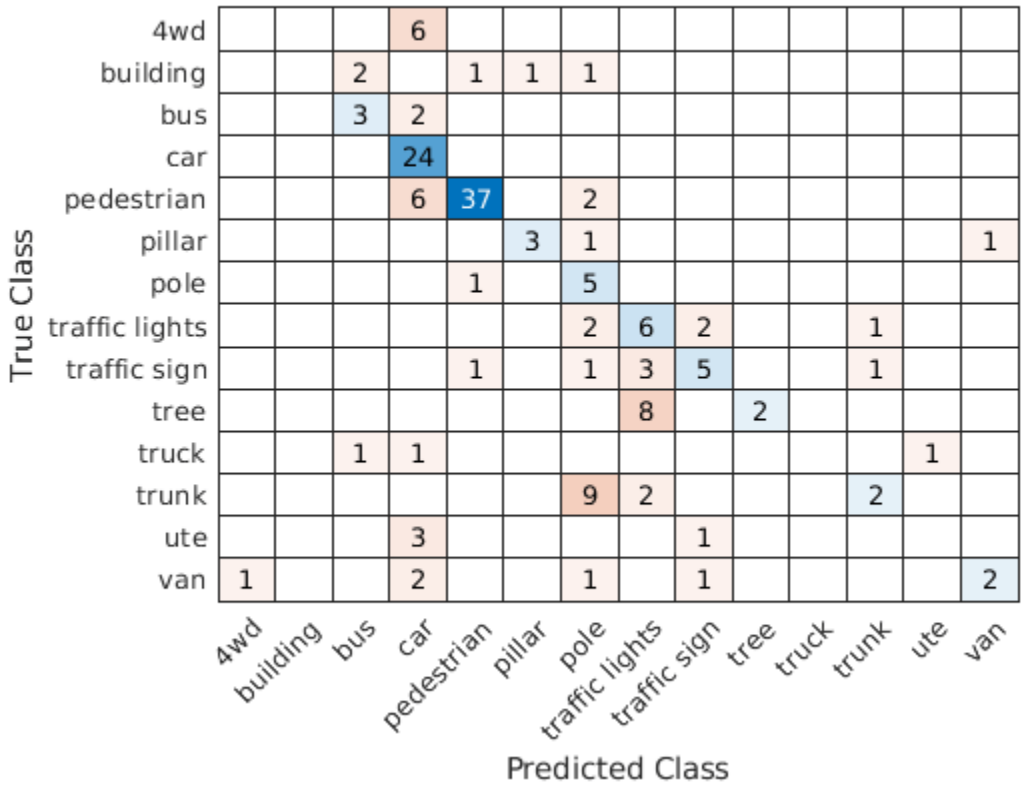
```
doTraining = false;

if doTraining
```

```
% Create a minibatchqueue to batch data from training and validation
% datastores. Use the batchData function, listed at the end of the
% example, to batch the point cloud data and one-hot encode the label
% data.
numOutputsFromDSRead = 2;
mbqTrain = minibatchqueue(dsTrain,numOutputsFromDSRead,...
    "MiniBatchSize", miniBatchSize,...
    "MiniBatchFcn",@batchData,...
    "MiniBatchFormat",["SCSB" "BC"]);

mbqVal = minibatchqueue(dsVal,numOutputsFromDSRead,...
    "MiniBatchSize", miniBatchSize,...
    "MiniBatchFcn",@batchData,...
    "MiniBatchFormat",["SCSB" "BC"]);

% 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

    % Shuffle data every epoch.
    shuffle(mbqTrain);

    % Iterate through data set.
    while hasdata(mbqTrain)
        iteration = iteration + 1;

        % Read next batch of training data.
        [XTrain, YTrain] = next(mbqTrain);

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients, loss, state, acc] = dlfeval(@modelGradients,XTrain,YTrain,parameters,sta

        % 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(mbqVal)
```

```
            % Read next batch of validation data.
            [XVal, YVal] = next(mbqVal);

            % 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 = aggreateConfusionMetric(cmat,YValLabel,YPredLabel);
        end

        % Update training progress plot with average classification accuracy.
        acc = sum(diag(cmat))./sum(cmat,"all");
        addpoints(valAccPlotter,iteration,acc);

        % Upate the learning rate.
        if mod(epoch,learnRateDropPeriod) == 0
            learnRate = learnRate * learnRateDropFactor;
        end

        % Reset training and validation data queues.
        reset(mbqTrain);
        reset(mbqVal);
    end

else
    % Download pretrained model parameters, model state, and validation
    % results.
    pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/pointnetSydneyUrbanObjec
    pretrainedResults = downloadPretrainedPointNet(pretrainedURL);

    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
```

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 = 0.5742
```

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 `dlarray`.

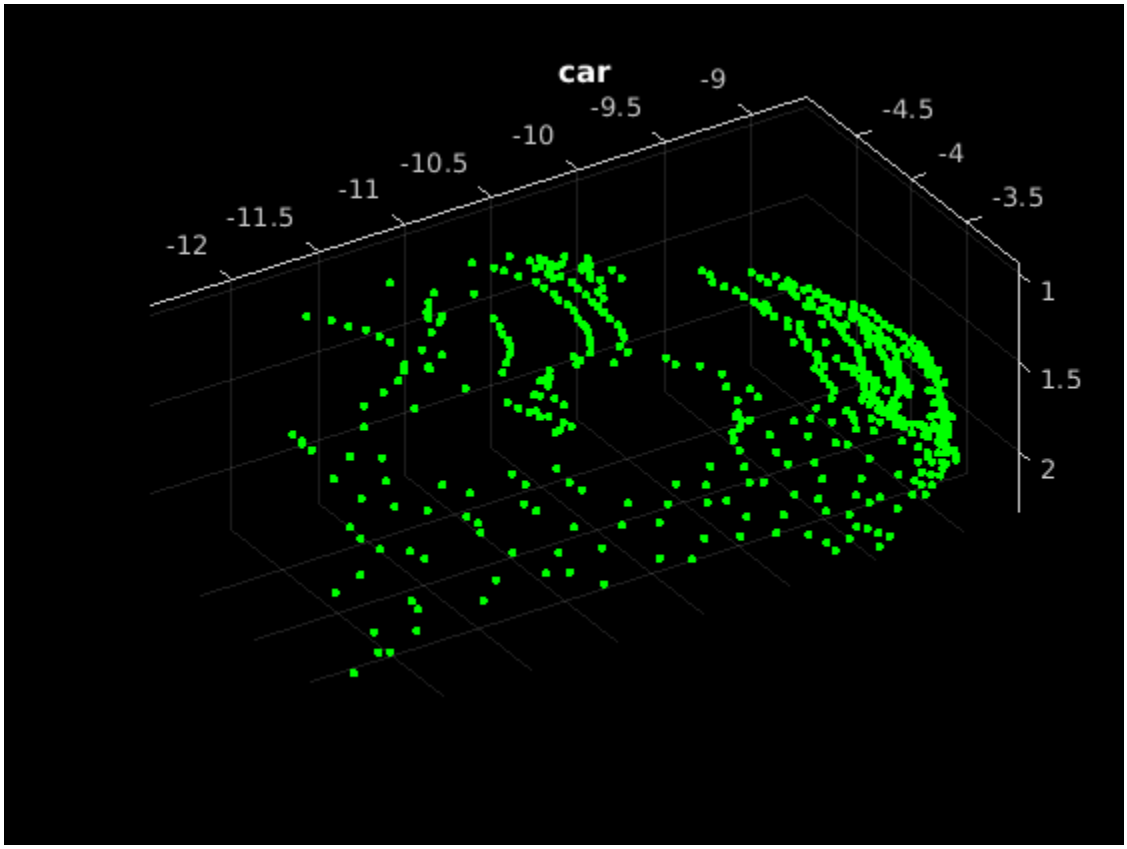
```
ptCloud = pcread("car.pcd");
X = preprocessPointCloud(ptCloud);
dlX = dlarray(X{1},"SCSB");
```

Predict point cloud labels with the `pointnetClassifier` model function.

```
YPred = pointnetClassifier(dlX,parameters,state,false);
[~,classIdx] = max(YPred,[],1);
```

Display the point cloud and the predicted label with the highest score.

```
figure
pcshow(ptCloud.Location,[0 1 0],"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,is

% Shared MLP.
[dlY,state.SharedMLP1.Perceptron] = sharedMLP(dlY,parameters.SharedMLP1.Perceptron,state.SharedMl

% Feature transform.
[dlY,state.FeatureTransform,T] = dataTransform(dlY,parameters.FeatureTransform,state.FeatureTrans

% Shared MLP.
[dlY,state.SharedMLP2.Perceptron] = sharedMLP(dlY,parameters.SharedMLP2.Perceptron,state.SharedMl

% 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.Perceptr

% Max operation.
dlY = max(dlY,[],1);

% Shared MLP.
[dlY,state.Block2.Perceptron] = sharedMLP(dlY,parameters.Block2.Perceptron,state.Block2.Perceptr

% Transform net (T-Net). Apply last fully connected operation as W*X to
% predict tranformation matrix T.
dlY = squeeze(dlY); % N-by-B
T = parameters.Transform * stripdims(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.
[C,B] = size(dlX,[3 4]);  % [M 1 K B]
dlX = reshape(dlX,[],C,1,B); % [M K 1 B]
Y = pagemtimes(dlX,T);
dlY = dlarray(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 = dlarray(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 = dlarray(weights);
p.Conv.Bias = dlarray(bias);

p.BatchNorm.Offset = dlarray(zeros(hiddenChannelSize(1),1,"single"));
p.BatchNorm.Scale = dlarray(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 = dlarray(weights);
    p.Conv.Bias = dlarray(bias);

    p.BatchNorm.Offset = dlarray(zeros(hiddenChannelSize(k),1,"single"));
    p.BatchNorm.Scale = dlarray(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,numClass
[params,state] = initializeSharedMLP(inputChannelSize,hiddenChannelSize);

weights = initializeWeightsGaussian([numClasses hiddenChannelSize(end)]);
bias = zeros(numClasses,1,"single");
params.FC.Weights = dlarray(weights);
params.FC.Bias = dlarray(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 `pcdownsample` 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} = pcdownsample(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 = pcdownsample(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 = aggreateConfusionMetric(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
```

`downloadSydneyUrbanObjects` Function

The `downloadSydneyUrbanObjects` function downloads the data set and saves it to a temporary directory.

```
function datapath = 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";

datapath = fullfile(dataLoc,'sydney-urban-objects-dataset');
if ~exist(datapath,'dir')
    disp('Downloading Sydney Urban Objects data set...');
```

```
    untar(url+name,dataLoc);
end

end
```

`loadSydneyUrbanObjectsData` Function

The `loadSydneyUrbanObjectsData` function creates a datastore for loading point cloud and label data from the Sydney Urban Objects data set.

```
function ds = loadSydneyUrbanObjectsData(datapath,folds)

if nargin == 0
    return;
end

if nargin < 2
    folds = 1:4;
end

datapath = string(datapath);
path = fullfile(datapath,'objects',filesep);

% Add folds to 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
```

`batchData` Function

The `batchData` function collates data into batches and moves data to the GPU for processing.

```
function [X,Y] = batchData(ptCloud,labels)
X = cat(4,ptCloud{:});
labels = cat(1,labels{:});
Y = onehotencode(labels,2);
end
```

`extractTrainingData` Function

The extractTrainingData function extracts point cloud and label data from the Sydney Urban Objects data set.

```
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
```

`readbin` Function

The `readbin` function reads the point cloud data from Sydney Urban Object binary files.

```
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
```

`downloadPretrainedPointNet` Function

The `downloadPretrainedPointNet` function downloads a pretrained pointnet model.

```
function data = downloadPretrainedPointNet(pretrainedURL)
% Download and load a pretrained pointnet model.
if ~exist('pointnetSydneyUrbanObjects.mat', 'file')
    if ~exist('pointnetSydneyUrbanObjects.zip', 'file')
        disp('Downloading pretrained detector (5 MB)...');
        websave('pointnetSydneyUrbanObjects.zip', pretrainedURL);
    end
    unzip('pointnetSydneyUrbanObjects.zip');
end
data = load("pointnetSydneyUrbanObjects.mat");
end
```

`prepareForPrediction` Function

The `prepareForPrediction` function is used to apply a user-defined function to nested structure data. It is a 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 12-2
- “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”.

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
    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. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. 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=4×2 table
              imageFilename                    vehicle
    ____________________________________    _______________

    {'vehicleImages/image_00001.jpg'}    {[220 136 35 28]}
    {'vehicleImages/image_00002.jpg'}    {[175 126 61 45]}
    {'vehicleImages/image_00003.jpg'}    {[108 120 45 33]}
    {'vehicleImages/image_00004.jpg'}    {[124 112 38 36]}
```

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.

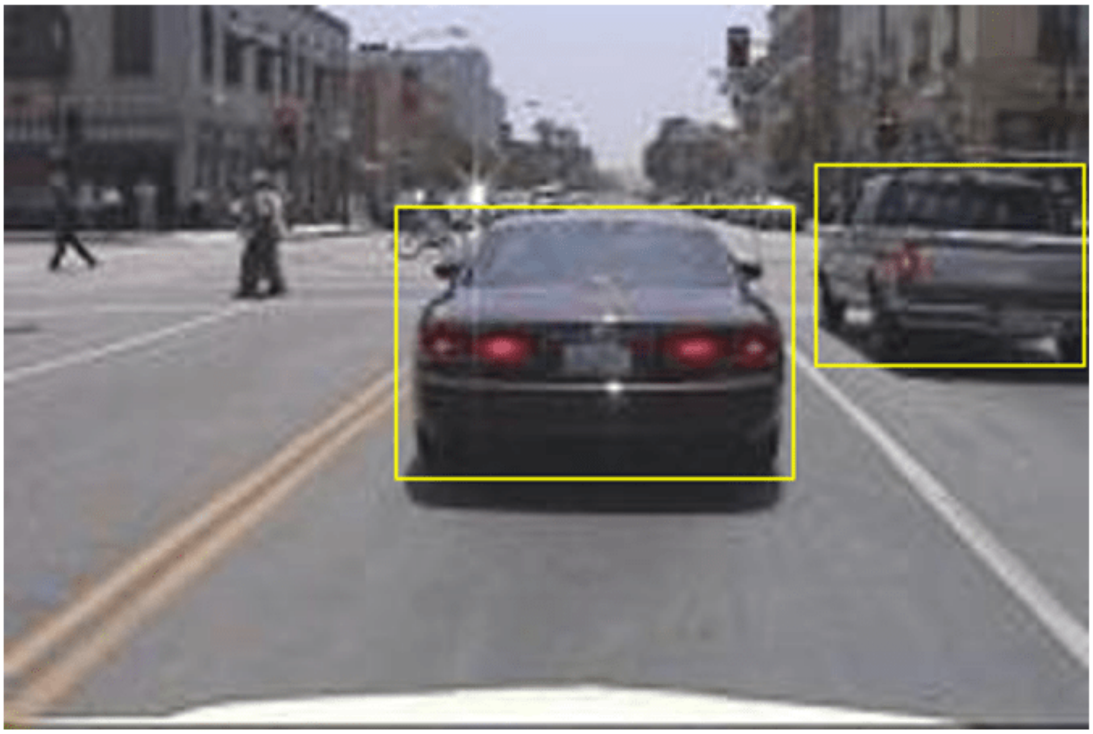
```
imdsTrain = imageDatastore(trainingData{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingData(:,'vehicle'));

imdsTest = imageDatastore(testData{:,'imageFilename'});
bldsTest = boxLabelDatastore(testData(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
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 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 3-298.

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.

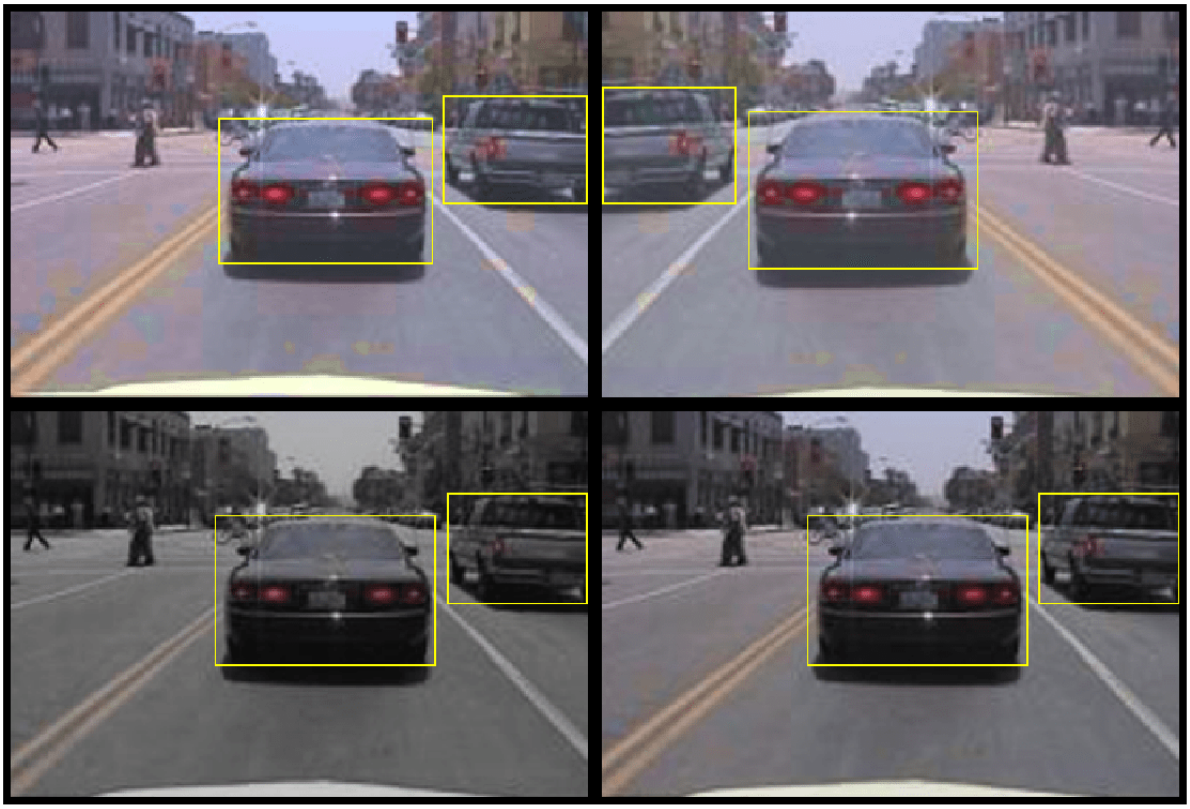
Note that data augmentation is not applied to the test data. Ideally, test data should be representative of the original data and is left unmodified for unbiased evaluation.

```
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.

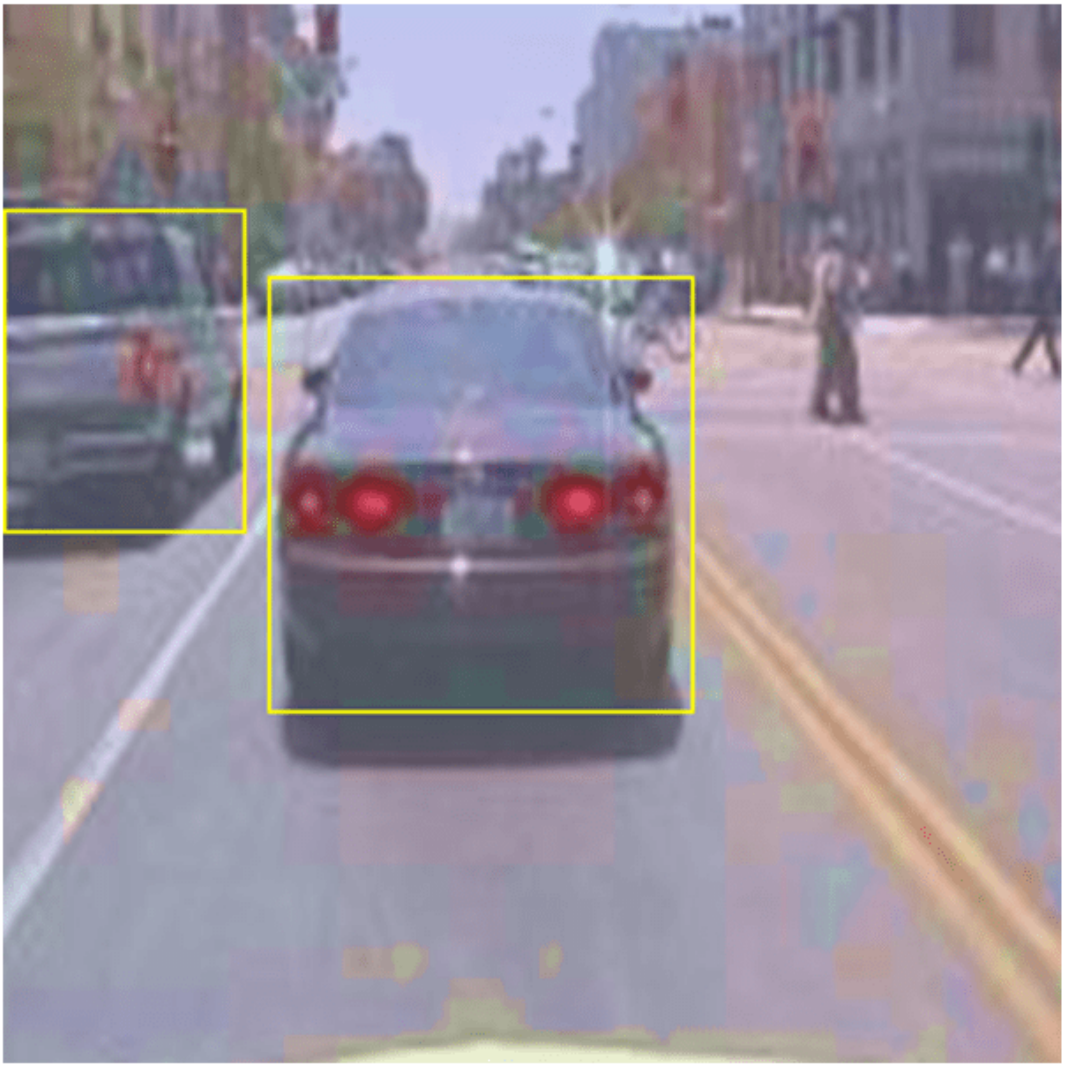
```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))
```

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);
```

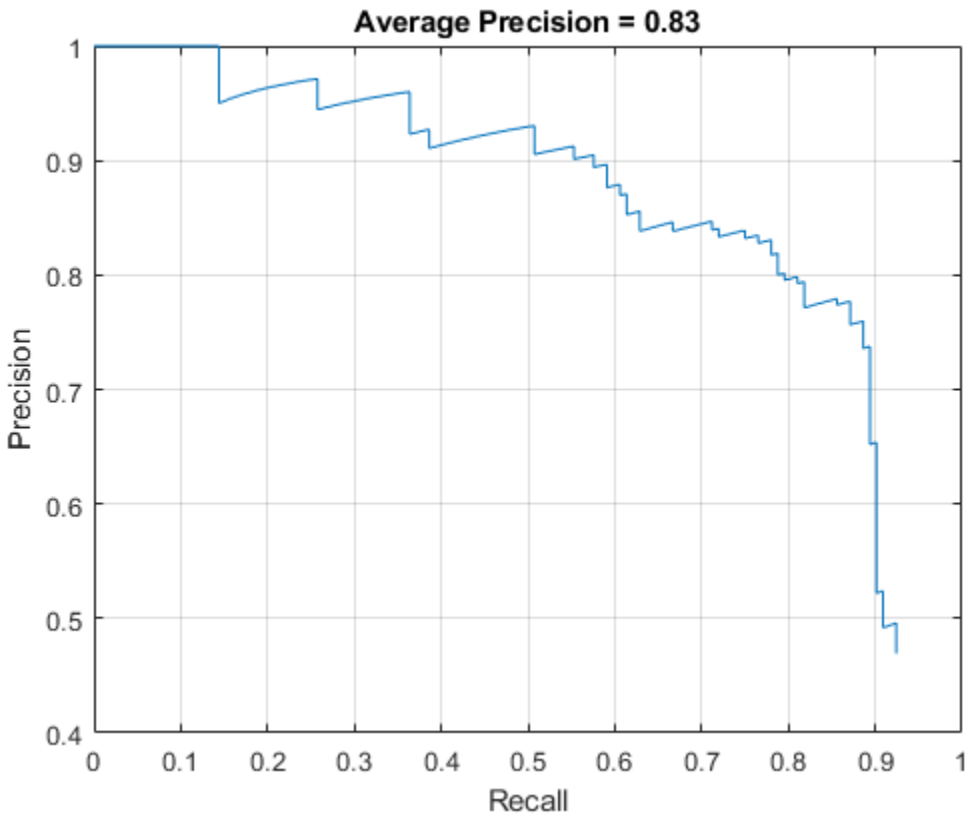
Evaluate the object detector using average precision metric.

```
[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);

% Sanitize boxes, if needed.
A{2} = helperSanitizeBoxes(A{2}, sz);

% 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.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize boxes, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Resize boxes.
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
`estimateAnchorBoxes` | `analyzeNetwork` | `combine` | `transform` | `read` | `evaluateDetectionPrecision`

Objects
`boxLabelDatastore` | `imageDatastore`

More About

- “Anchor Boxes for Object Detection” on page 17-32
- “Estimate Anchor Boxes From Training Data” on page 3-319
- “Transfer Learning with Deep Network Designer” (Deep Learning Toolbox)

- “Getting Started with Object Detection Using Deep Learning” on page 17-24

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');
```

Read the target image containing a cluttered scene.

```
sceneImage = imread('clutteredDesk.jpg');
figure;
imshow(sceneImage);
title('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));
```

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);
```

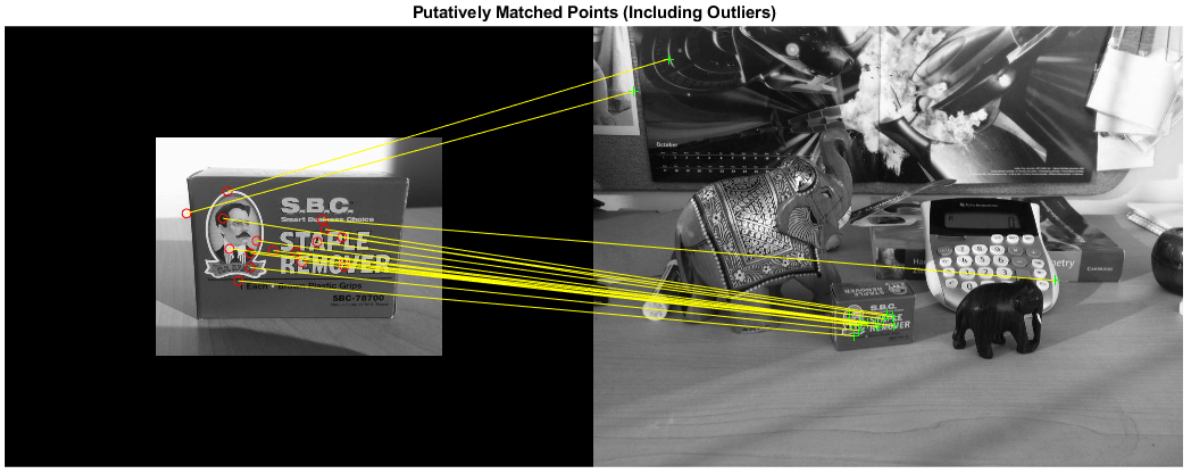
Step 4: Find Putative Point Matches

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)');
```



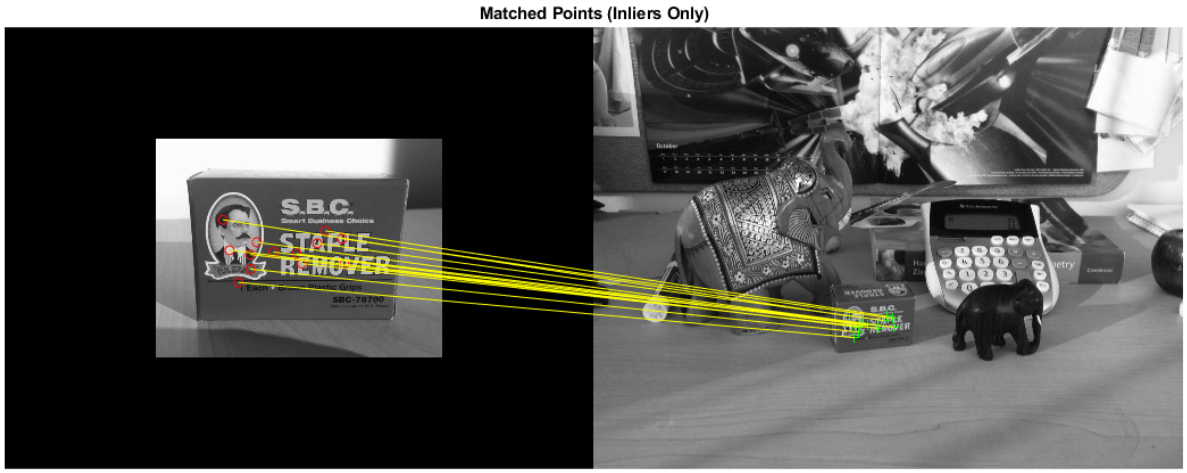
Step 5: Locate the Object in the Scene Using Putative Matches

`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.

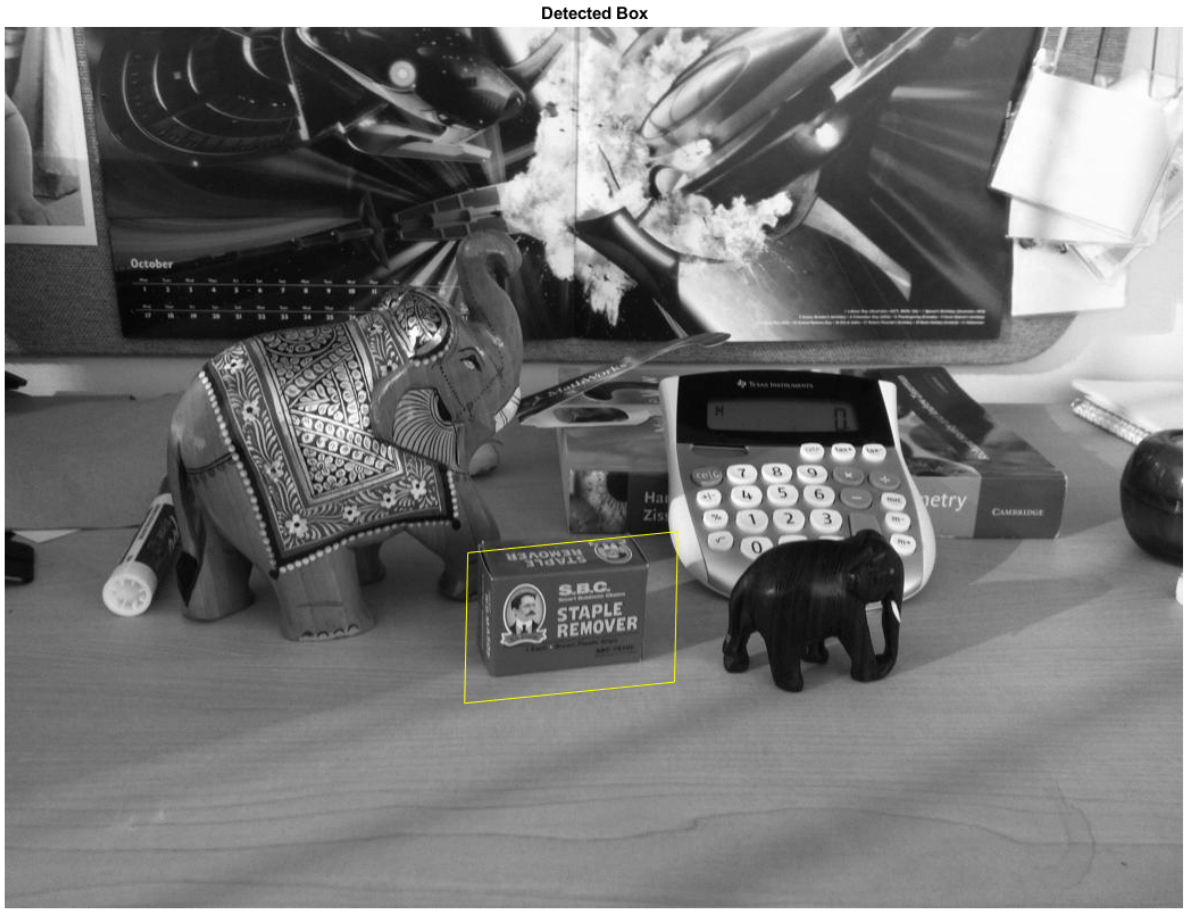
```
boxPolygon = [1, 1;...                           % top-left
        size(boxImage, 2), 1;...                 % top-right
        size(boxImage, 2), size(boxImage, 1);... % bottom-right
        1, size(boxImage, 1);...                 % bottom-left
        1, 1];                   % top-left again 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

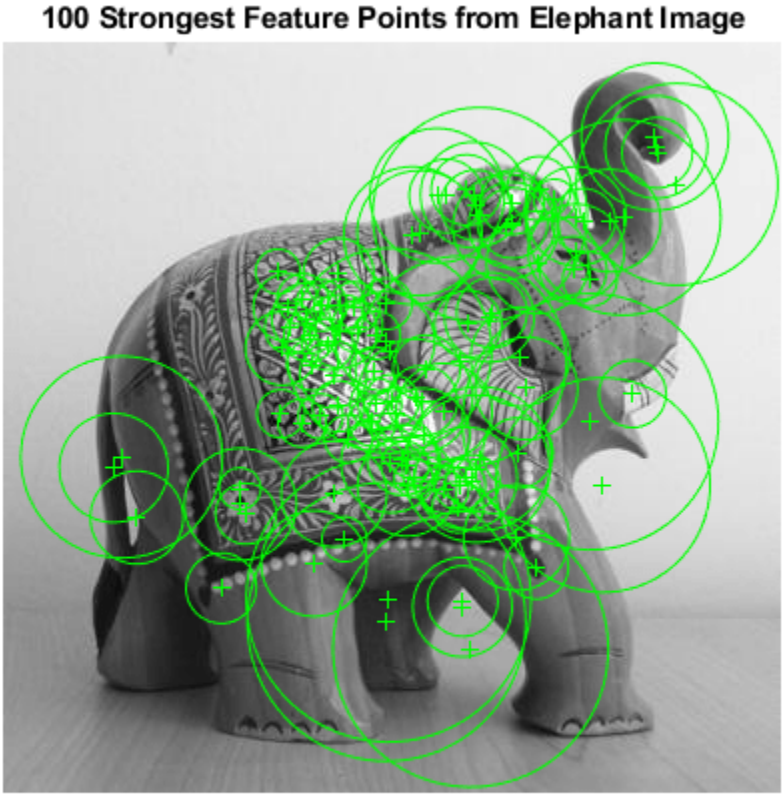
Detect a second object by using the same steps as before.

Read an image containing the second object of interest.

```
elephantImage = imread('elephant.jpg');
figure;
imshow(elephantImage);
title('Image of an Elephant');
```

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');
```



Extract feature descriptors.

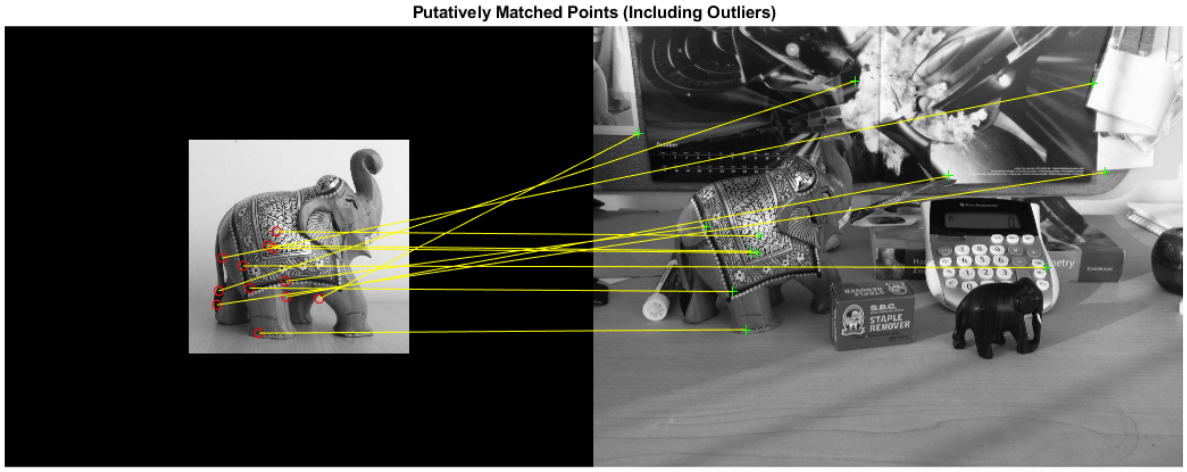
```
[elephantFeatures, elephantPoints] = extractFeatures(elephantImage, elephantPoints);
```

Match Features

```
elephantPairs = matchFeatures(elephantFeatures, sceneFeatures, 'MaxRatio', 0.9);
```

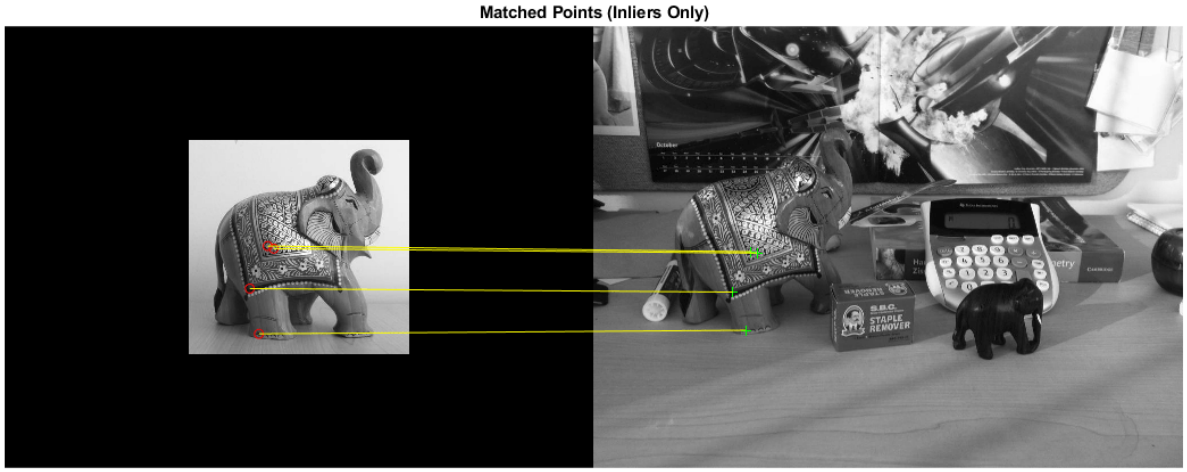
Display putatively matched features.

```
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

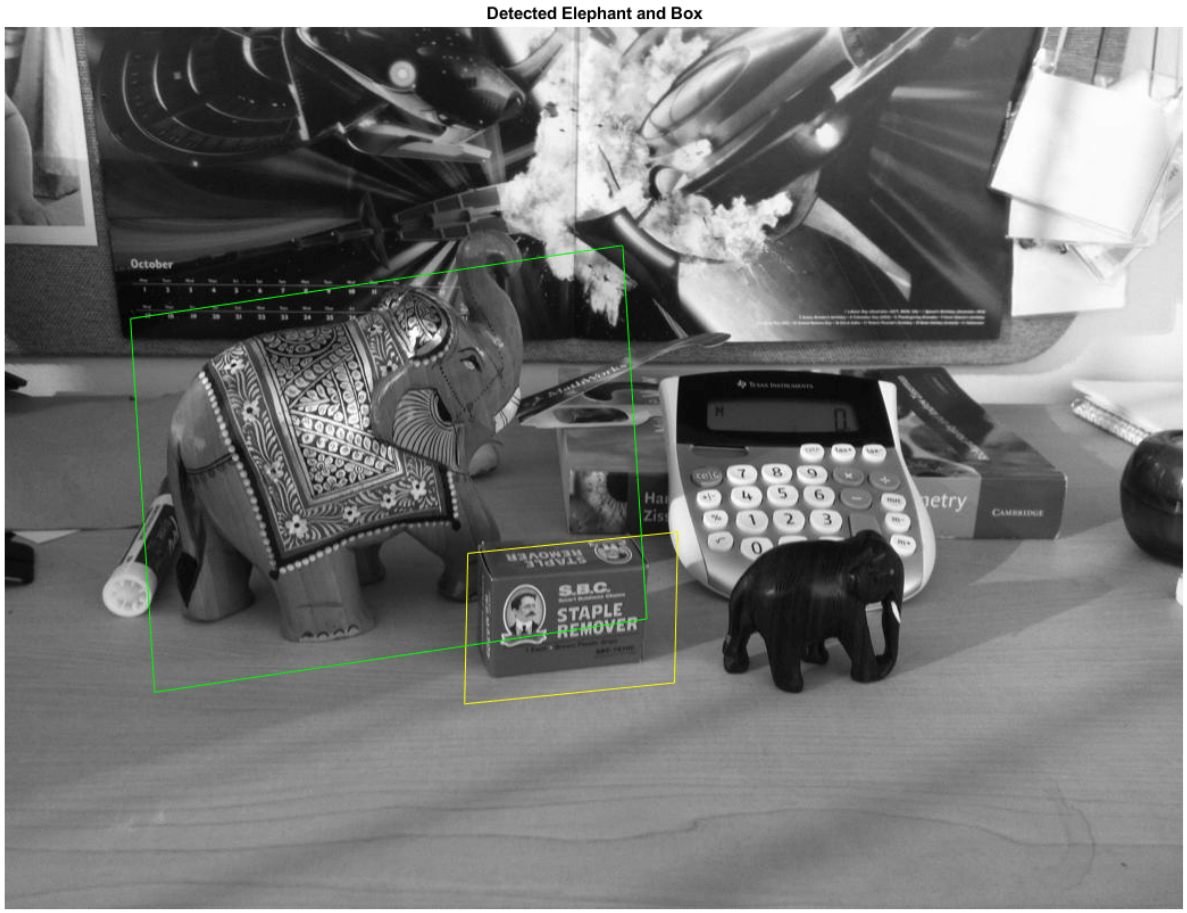
```
[tform, inlierElephantPoints, inlierScenePoints] = ...
    estimateGeometricTransform(matchedElephantPoints, matchedScenePoints, 'affine');
figure;
showMatchedFeatures(elephantImage, sceneImage, inlierElephantPoints, ...
    inlierScenePoints, 'montage');
title('Matched Points (Inliers Only)');
```



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');
```



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 17-62.

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 `resnet18` (Deep Learning Toolbox). 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://ssd.mathworks.com/supportfiles/vision/data/deeplabv3plusResnet18CamVid.z
pretrainedFolder = fullfile(tempdir,'pretrainedNetwork');
pretrainedNetworkZip = fullfile(pretrainedFolder,'deeplabv3plusResnet18CamVid.zip');
if ~exist(pretrainedNetworkZip,'file')
    mkdir(pretrainedFolder);
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetworkZip,pretrainedURL);
end
unzip(pretrainedNetworkZip, pretrainedFolder)
```

A CUDA-capable NVIDIA™ GPU is highly recommended for running this example. Use of a GPU requires Parallel Computing Toolbox™. For information about the supported compute capabilities, see “GPU Support by Release” (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.zi
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.z

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

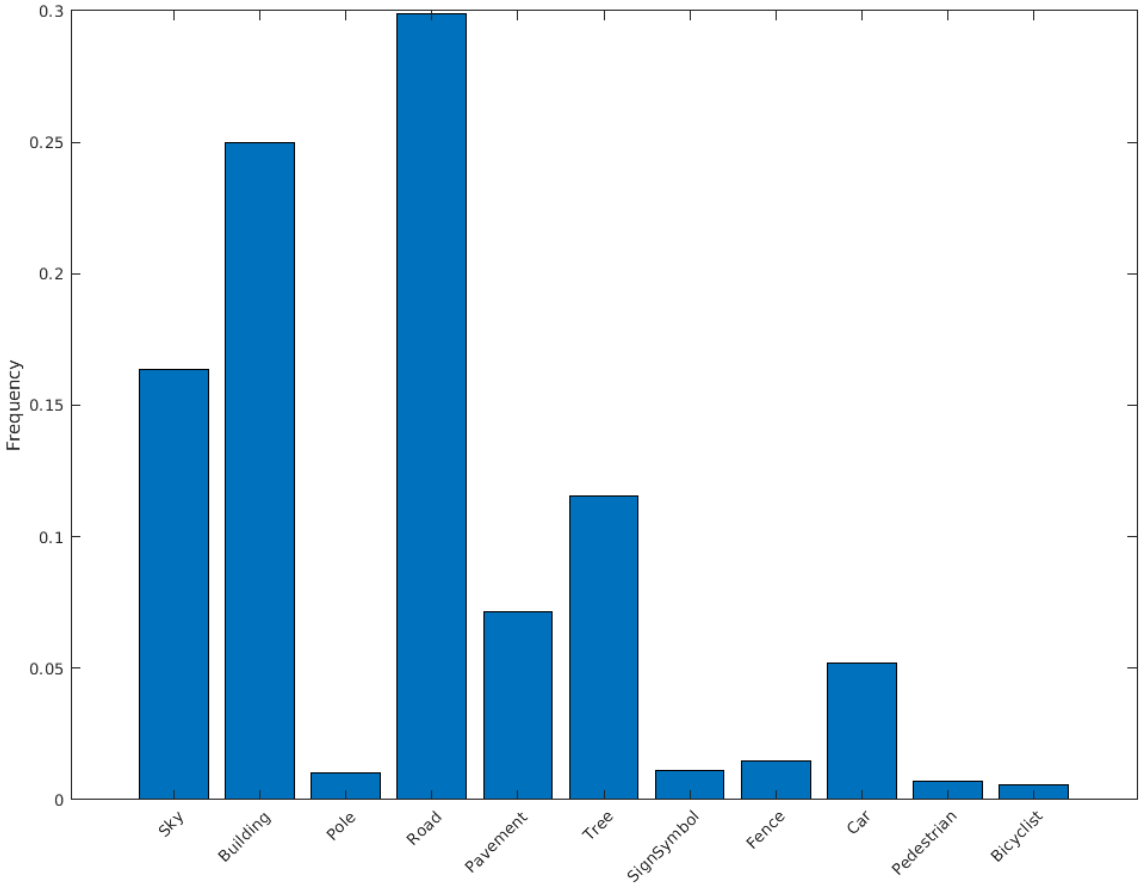
To see the distribution of class labels in the CamVid dataset, use `countEachLabel`. This function counts the number of pixels by class label.

```
tbl = countEachLabel(pxds)
```

```
tbl=11×3 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 = 11×1

    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.
dsVal = combine(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',dsVal,...
    '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 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 increasing the number of labeled training samples. To apply the same random transformation to both image and pixel label data use datastore `combine` and `transform`. First, combine `imdsTrain` and `pxdsTrain`.

```
dsTrain = combine(imdsTrain, pxdsTrain);
```

Next, use datastore `transform` to apply the desired data augmentation defined in the supporting function `augmentImageAndLabel`. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation.

```
xTrans = [-10 10];
yTrans = [-10 10];
dsTrain = transform(dsTrain, @(data)augmentImageAndLabel(data,xTrans,yTrans));
```

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.

Start Training

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. Training this network takes about 70 minutes. Depending on your GPU hardware, it may take longer.

```
doTraining = false;
if doTraining
    [net, info] = trainNetwork(dsTrain,lgraph,options);
else
    pretrainedNetwork = fullfile(pretrainedFolder,'deeplabv3plusResnet18CamVid.mat');
    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.

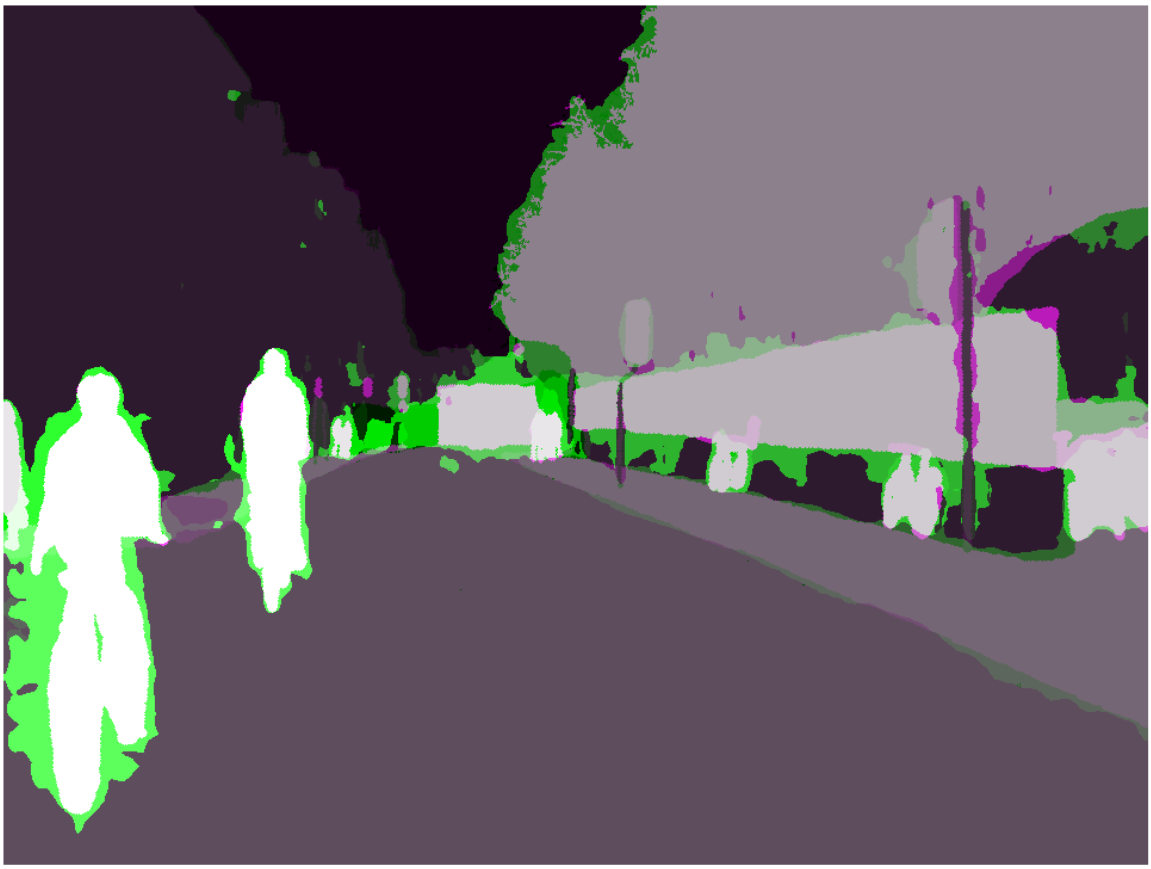
```
I = readimage(imdsTest,35);
C = semanticseg(I, net);
```

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=11×2 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=1×5 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=11×3 table
                  Accuracy      IoU      MeanBFScore
                  ________    _______    ___________

    Sky           0.93112     0.90209       0.8952
    Building      0.78453     0.76098      0.58511
    Pole          0.71586     0.21477       0.5144
    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         0.81506     0.46484      0.48565
    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 pixelLabelColorbar(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

function data = augmentImageAndLabel(data, xTrans, yTrans)
% Augment images and pixel label images using random reflection and
% translation.

for i = 1:size(data,1)

    tform = randomAffine2d(...
        'XReflection',true,...
        'XTranslation', xTrans, ...
        'YTranslation', yTrans);

    % Center the view at the center of image in the output space while
    % allowing translation to move the output image out of view.
    rout = affineOutputView(size(data{i,1}), tform, 'BoundsStyle', 'centerOutput');

    % Warp the image and pixel labels using the same transform.
    data{i,1} = imwarp(data{i,1}, tform, 'OutputView', rout);
    data{i,2} = imwarp(data{i,2}, tform, 'OutputView', rout);

end
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

`pixelLabelDatastore` | `pixelLabelImageDatastore` | `semanticseg` | `labeloverlay` | `countEachLabel` | `segnetLayers` | `pixelClassificationLayer` | `trainingOptions` | `imageDataAugmenter` | `trainNetwork` | `evaluateSemanticSegmentation`

More About

- “Getting Started with Semantic Segmentation Using Deep Learning” on page 17-62
- “Label Pixels for Semantic Segmentation” on page 9-14
- “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 `blockedImage` 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.

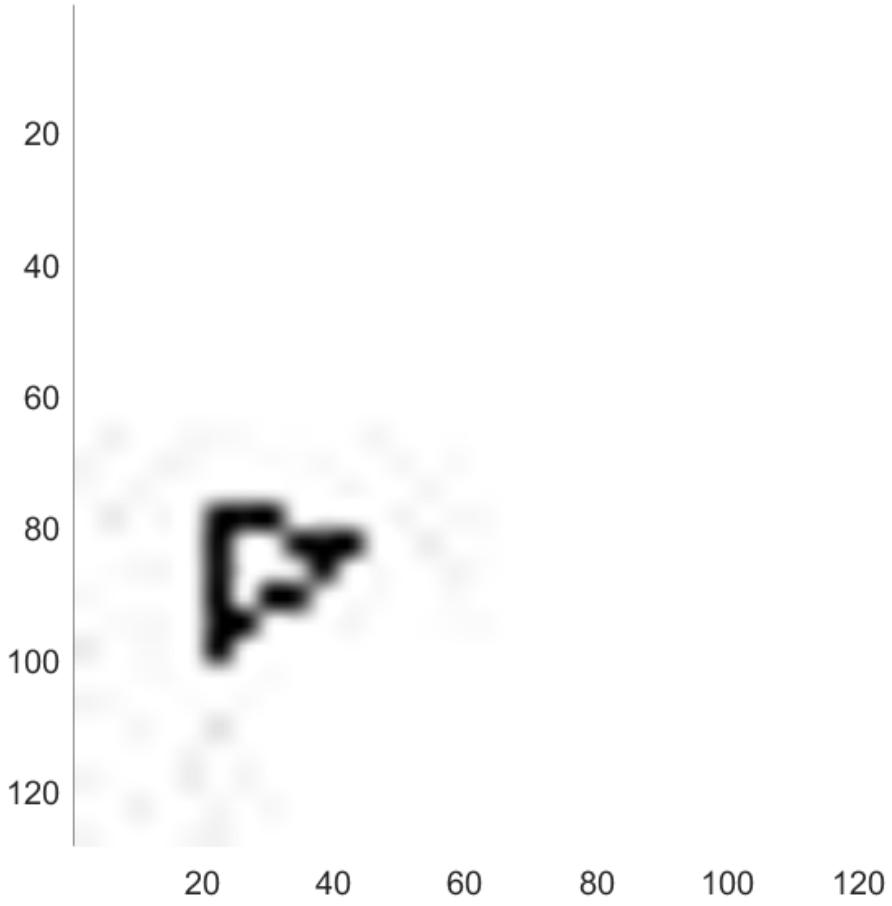
```
testImagesDir = fullfile(dataSetDir,'testImages');
```

Read three test images. Resize each image by a factor of four, convert it to data type `double`, then create a `blockedImage` object. A `blockedImage` 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) = blockedImage(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.

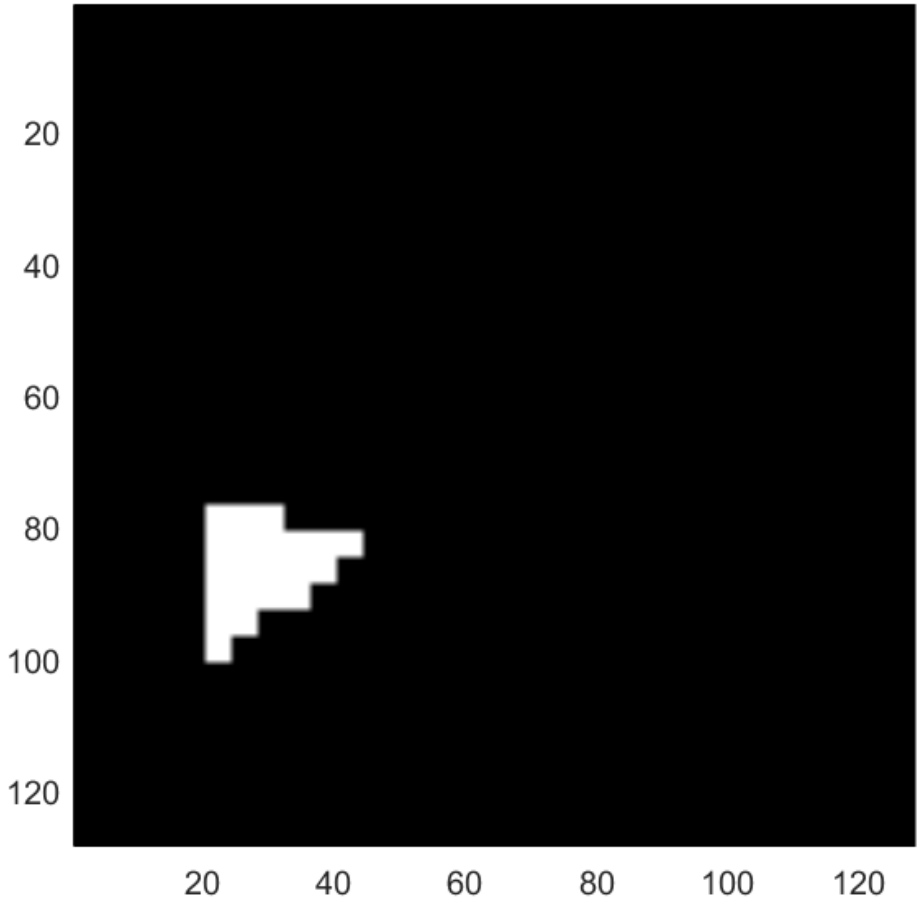
```
classNames = ["triangle","background"];
labelIDs   = [255 0];
```

Read in the ground truth labels for each test image. Create a `blockedImage` object from each ground truth label.

```
for idx = 1:numImages
    gtLabel = imread(fullfile(testLabelsDir,['labeled_image_' '00' num2str(idx) '.png']));
    gtLabel = imresize(gtLabel,4,'nearest');
    groundTruthImages(idx) = blockedImage(gtLabel);
end
```

Display the first ground truth image.

```
bigimageshow(groundTruthImages(1))
```

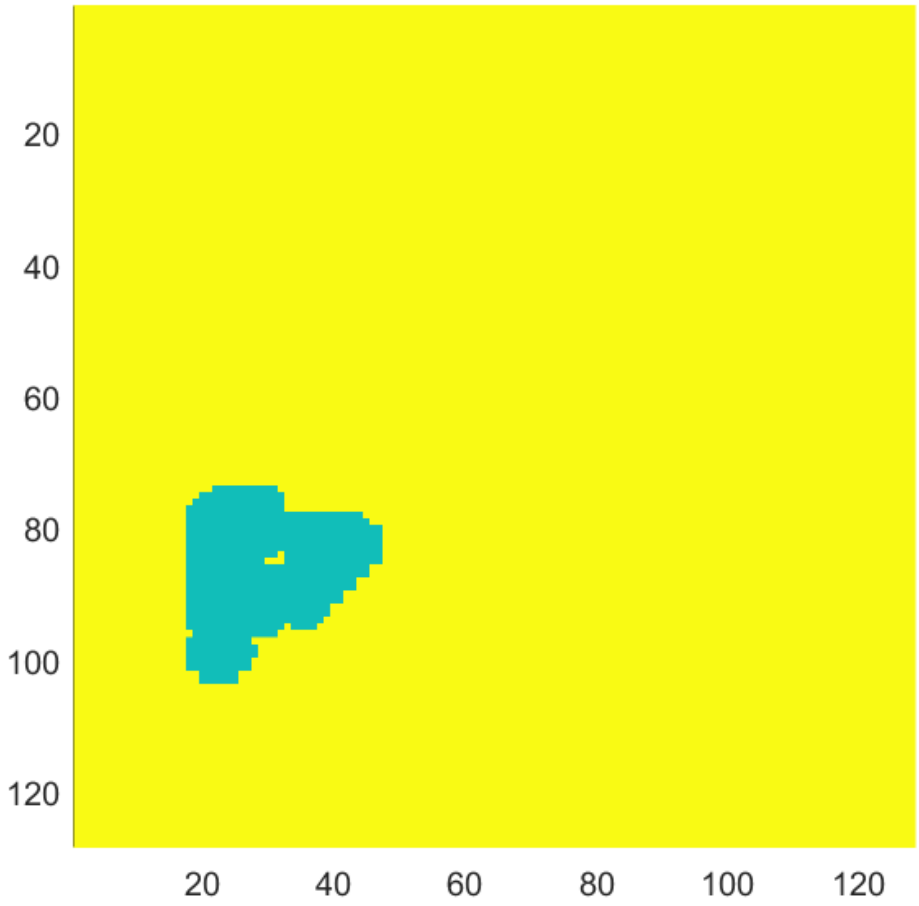


For each test image, use the `apply` function to process each block. The `apply` function performs the operations specified by the helper function `segmentAndCalculateBlockMetrics`, which is defined at the end of this example. The function performs semantic segmentation of each block and calculates the confusion matrix between the predicted and ground truth labels.

```
blockSize = [32 32];
datasetConfMat = table;
for idx = 1:numImages
    [segmentedImages(idx),blockConfMatOneImage] = apply(testImages(idx), ...
        @(block,labeledImageBlock) segmentAndCalculateBlockMetrics(block,labeledImageBlock,net,c
        'ExtraImages',groundTruthImages(idx),'PadPartialBlocks',true,'BlockSize',blockSize,'UsePa

    % Read all the block results of an image and update the image number
    blockConfMatOneImageDS = blockedImageDatastore(blockConfMatOneImage);
    blockConfMat = readall(blockConfMatOneImageDS);
    blockConfMat = struct2table([blockConfMat{:}]);
    blockConfMat.ImageNumber = idx.*ones(height(blockConfMat),1);
    datasetConfMat = [datasetConfMat;blockConfMat];
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 = 3×1

    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,

    outputLabeledImageBlock = semanticseg(block.Data,net);

    % Convert the ground truth labels to categorical
    labeledImageBlock = categorical(labeledImageBlock,labelIDs,classNames);
    confusionMatrix = segmentationConfusionMatrix(outputLabeledImageBlock,labeledImageBlock);

    % blockConfMatPerBlock is a struct with confusion matrices, image number
    % and blockInfo. Use the struct with evaluateSemanticSegmentation to
    % calculate metrics and aggregate block-based results.
    blockConfMatPerBlock.ConfusionMatrix = confusionMatrix;
    blockConfMatPerBlock.ImageNumber = block.ImageNumber;

    blockInfo.Start = block.Start;
    blockInfo.End = block.End;
    blockConfMatPerBlock.BlockInfo = blockInfo;
end
```

See Also

`apply` | `bigimage` | `segmentationConfusionMatrix` | `semanticSegmentationMetrics` | `evaluateSemanticSegmentation`

Related Examples

- “Semantic Segmentation Using Deep Learning” on page 3-222

More About

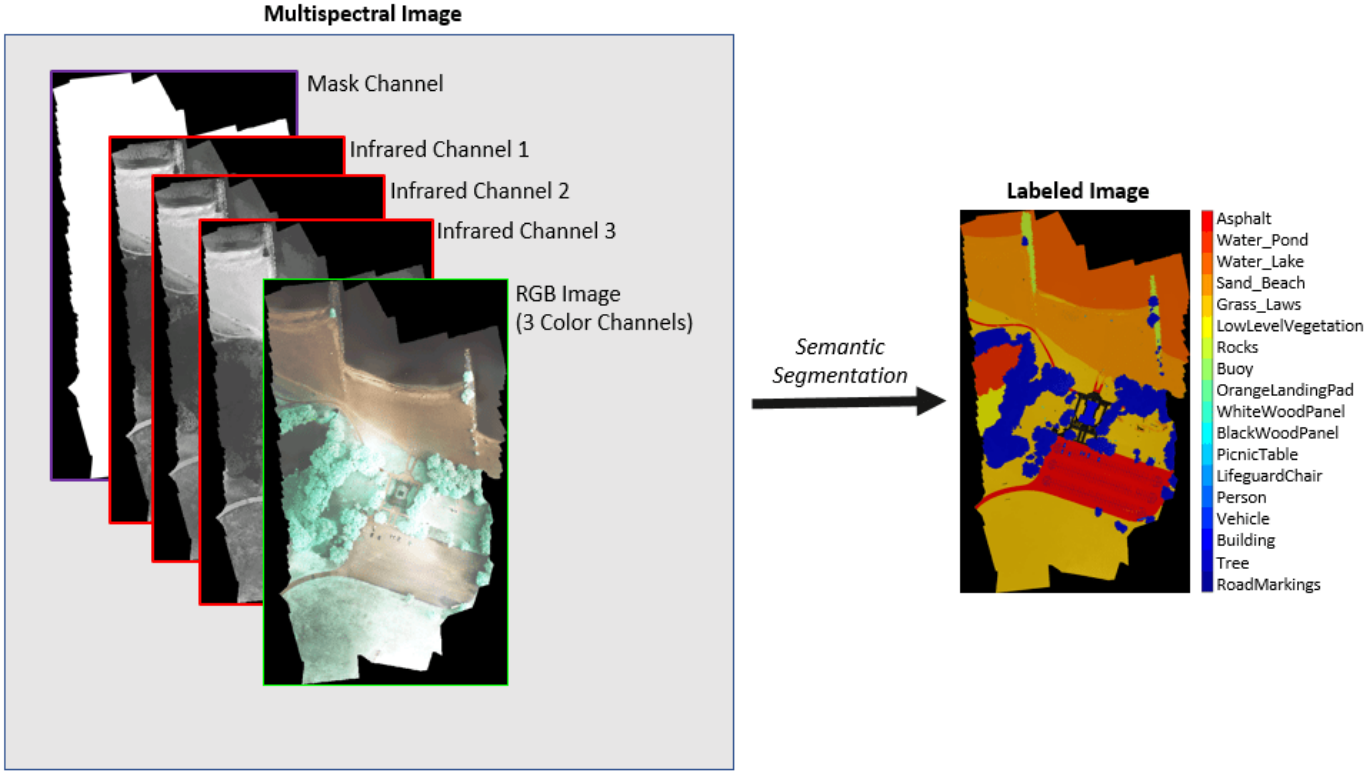
- “Getting Started with Semantic Segmentation Using Deep Learning” on page 17-62

Semantic Segmentation of Multispectral Images Using Deep Learning

This example shows how to perform semantic segmentation of a multispectral image with seven channels using a U-Net.

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 three near-infrared channels that provide a clearer separation of the classes.



This example shows how to use deep learning 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. Specify `dataDir` as the desired location of the data.

```
dataDir = fullfile(tempdir,"rit18_data");
downloadHamlinBeachMSIData(dataDir);
```

Inspect Training Data

Load the data set into the workspace.

```
load(fullfile(dataDir,"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 `switchChannelsToThirdPlane` helper function. 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);
```

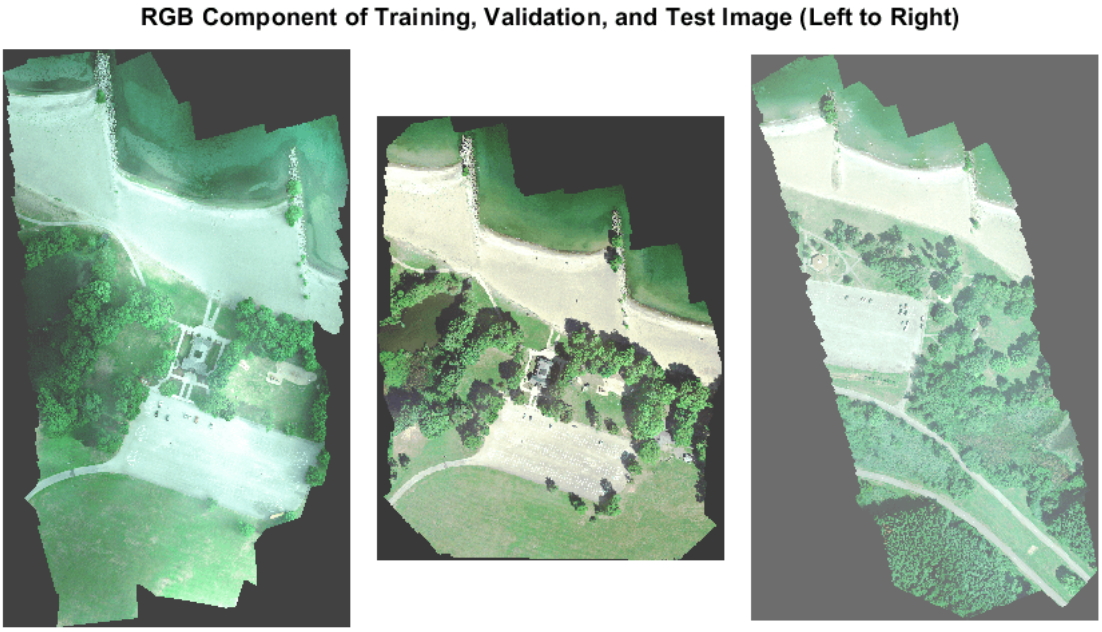
Confirm that the data has the correct structure.

```
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
```

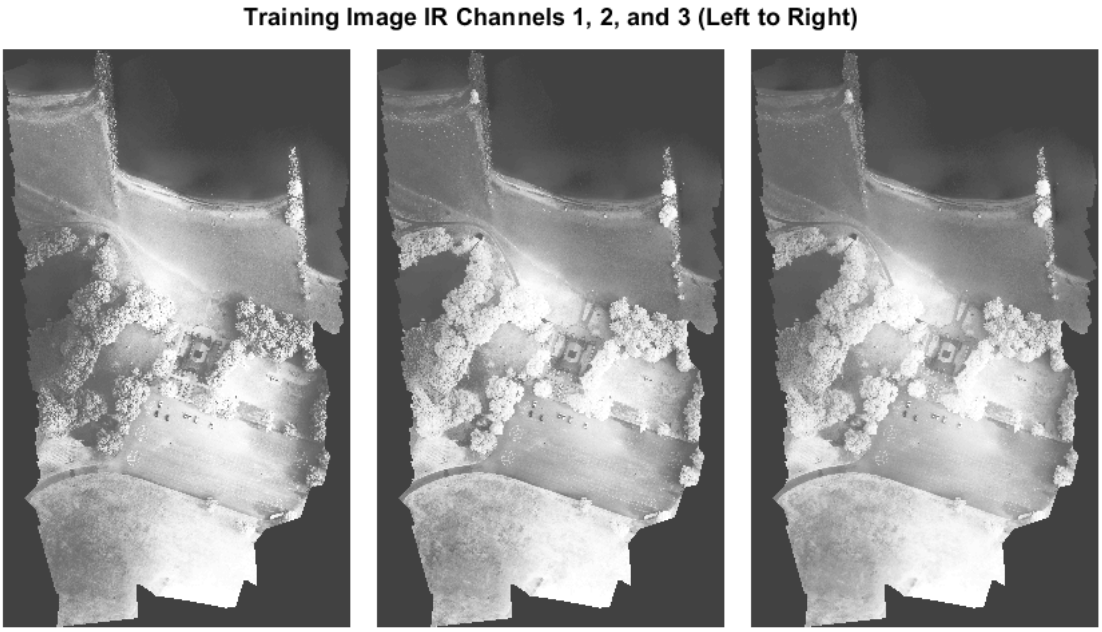
The RGB color channels are the 3rd, 2nd, and 1st image channels. Display the color component of the training, validation, and test images as a montage. To make the images appear brighter on the screen, equalize their histograms by using the `histeq` function.

```
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, Validation, and Test Image (Left to Right)")
```



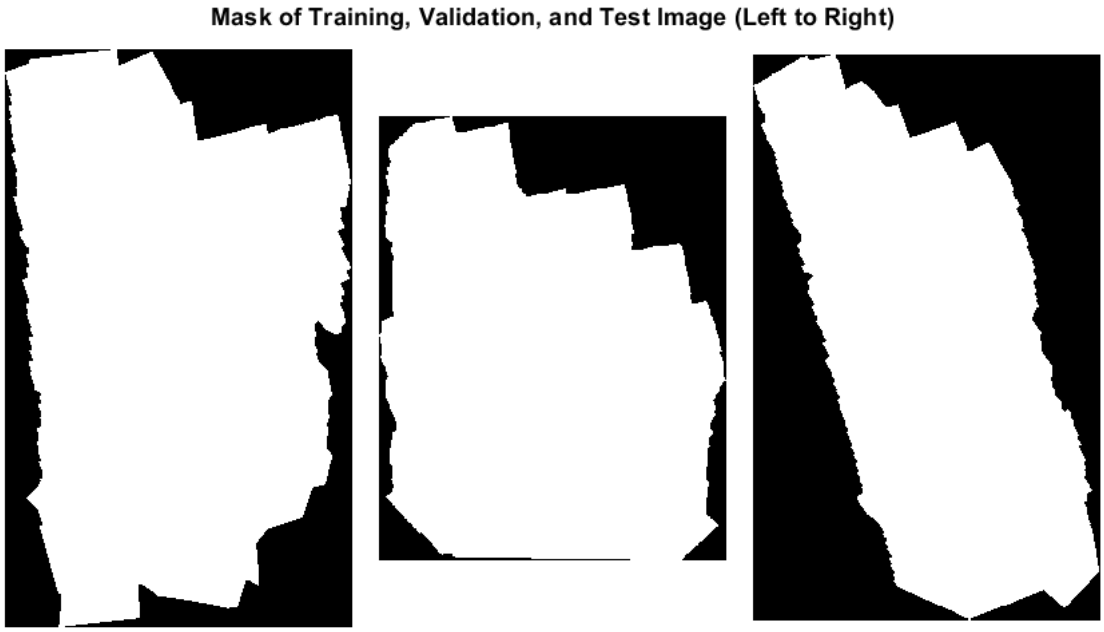
Display the last three histogram-equalized channels of the training data as a montage. These channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. For example, the trees near the center of the second channel image show more detail 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("Training Image IR Channels 1, 2, and 3 (Left to Right)")
```



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, Validation, and Test Image (Left to 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)
```

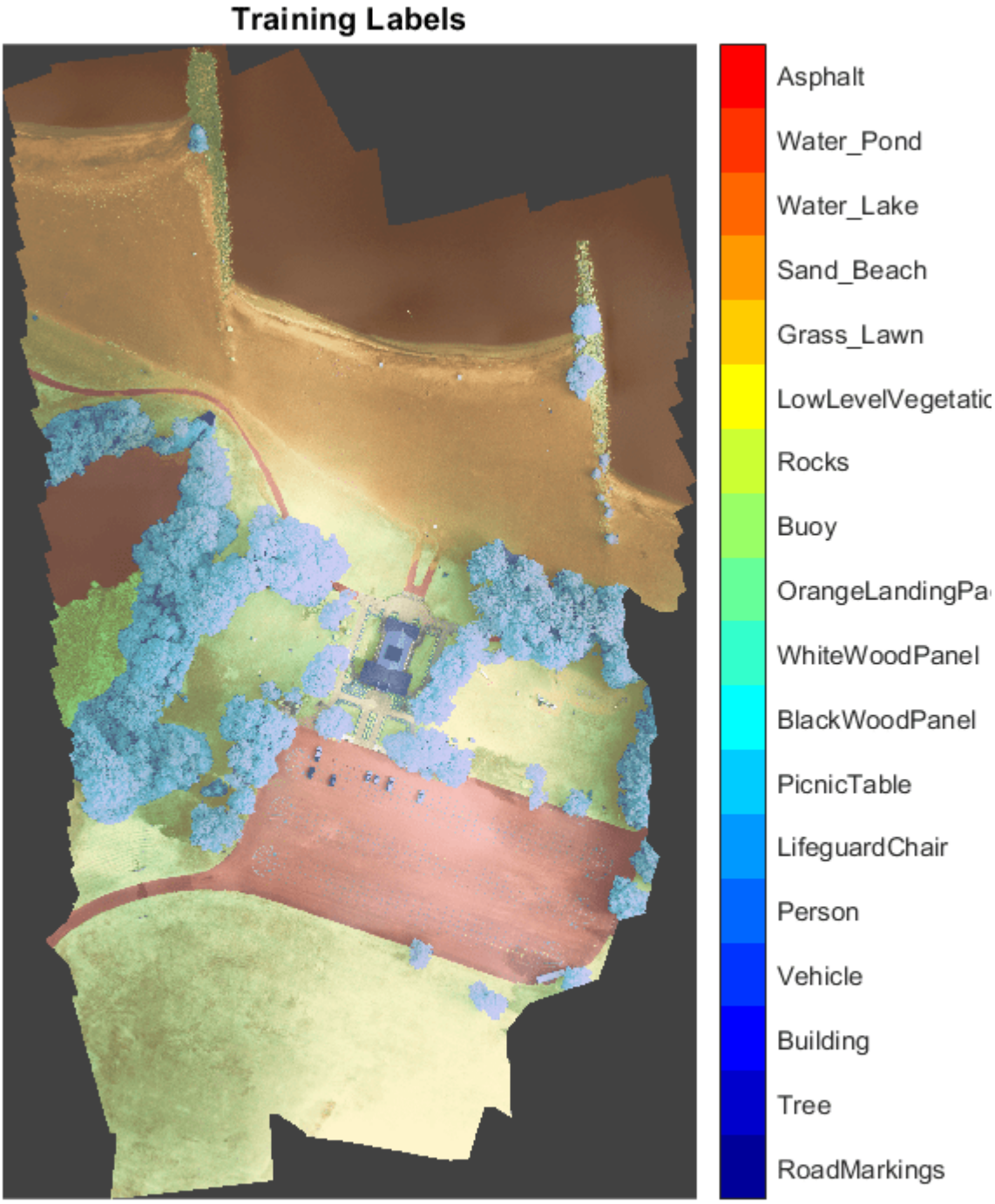
Create a vector of class names.

```
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 color bar to the image.

```
cmap = jet(numel(classNames));
B = labeloverlay(histeq(train_data(:,:,4:6)),train_labels,Transparency=0.8,Colormap=cmap);

figure
imshow(B)
title("Training Labels")
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar(TickLabels=cellstr(classNames),Ticks=ticks,TickLength=0,TickLabelInterpreter="none");
colormap(cmap)
```


Training Labels
Asphalt
Water_Pond
Water_Lake
Sand_Beach
Grass_Lawn
LowLevelVegetatic
Rocks
Buoy
OrangeLandingPa
WhiteWoodPanel
BlackWoodPanel
PicnicTable
LifeguardChair
Person
Vehicle
Building
Tree
RoadMarkings

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, `matRead6Channels`, 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=@matRead6Channels);
```

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)
```

```
  58×1 Layer array with layers:

     1   'ImageInputLayer'                           Image Input                 256×256×6 images w
     2   'Encoder-Section-1-Conv-1'                  Convolution                 64 3×3×6 convoluti
     3   'Encoder-Section-1-ReLU-1'                  ReLU                        ReLU
     4   'Encoder-Section-1-Conv-2'                  Convolution                 64 3×3×64 convolut
     5   'Encoder-Section-1-ReLU-2'                  ReLU                        ReLU
     6   'Encoder-Section-1-MaxPool'                 Max Pooling                 2×2 max pooling wi
     7   'Encoder-Section-2-Conv-1'                  Convolution                 128 3×3×64 convolu
     8   'Encoder-Section-2-ReLU-1'                  ReLU                        ReLU
     9   'Encoder-Section-2-Conv-2'                  Convolution                 128 3×3×128 convol
    10   'Encoder-Section-2-ReLU-2'                  ReLU                        ReLU
    11   'Encoder-Section-2-MaxPool'                 Max Pooling                 2×2 max pooling wi
    12   'Encoder-Section-3-Conv-1'                  Convolution                 256 3×3×128 convol
    13   'Encoder-Section-3-ReLU-1'                  ReLU                        ReLU
    14   'Encoder-Section-3-Conv-2'                  Convolution                 256 3×3×256 convol
    15   'Encoder-Section-3-ReLU-2'                  ReLU                        ReLU
    16   'Encoder-Section-3-MaxPool'                 Max Pooling                 2×2 max pooling wi
    17   'Encoder-Section-4-Conv-1'                  Convolution                 512 3×3×256 convol
    18   'Encoder-Section-4-ReLU-1'                  ReLU                        ReLU
    19   'Encoder-Section-4-Conv-2'                  Convolution                 512 3×3×512 convol
    20   'Encoder-Section-4-ReLU-2'                  ReLU                        ReLU
    21   'Encoder-Section-4-DropOut'                 Dropout                     50% dropout
    22   'Encoder-Section-4-MaxPool'                 Max Pooling                 2×2 max pooling wi
    23   'Mid-Conv-1'                                Convolution                 1024 3×3×512 convo
    24   'Mid-ReLU-1'                                ReLU                        ReLU
    25   'Mid-Conv-2'                                Convolution                 1024 3×3×1024 conv
    26   'Mid-ReLU-2'                                ReLU                        ReLU
    27   'Mid-DropOut'                               Dropout                     50% dropout
    28   'Decoder-Section-1-UpConv'                  Transposed Convolution      512 2×2×1024 trans
    29   'Decoder-Section-1-UpReLU'                  ReLU                        ReLU
    30   'Decoder-Section-1-DepthConcatenation'      Depth concatenation         Depth concatenatio
    31   'Decoder-Section-1-Conv-1'                  Convolution                 512 3×3×1024 convo
    32   'Decoder-Section-1-ReLU-1'                  ReLU                        ReLU
    33   'Decoder-Section-1-Conv-2'                  Convolution                 512 3×3×512 convol
    34   'Decoder-Section-1-ReLU-2'                  ReLU                        ReLU
    35   'Decoder-Section-2-UpConv'                  Transposed Convolution      256 2×2×512 transp
    36   'Decoder-Section-2-UpReLU'                  ReLU                        ReLU
    37   'Decoder-Section-2-DepthConcatenation'      Depth concatenation         Depth concatenatio
    38   'Decoder-Section-2-Conv-1'                  Convolution                 256 3×3×512 convol
    39   'Decoder-Section-2-ReLU-1'                  ReLU                        ReLU
    40   'Decoder-Section-2-Conv-2'                  Convolution                 256 3×3×256 convol
    41   'Decoder-Section-2-ReLU-2'                  ReLU                        ReLU
    42   'Decoder-Section-3-UpConv'                  Transposed Convolution      128 2×2×256 transp
    43   'Decoder-Section-3-UpReLU'                  ReLU                        ReLU
    44   'Decoder-Section-3-DepthConcatenation'      Depth concatenation         Depth concatenatio
    45   'Decoder-Section-3-Conv-1'                  Convolution                 128 3×3×256 convol
    46   'Decoder-Section-3-ReLU-1'                  ReLU                        ReLU
    47   'Decoder-Section-3-Conv-2'                  Convolution                 128 3×3×128 convol
```

```
    48   'Decoder-Section-3-ReLU-2'              ReLU                          ReLU
    49   'Decoder-Section-4-UpConv'              Transposed Convolution        64 2×2×128 transpo
    50   'Decoder-Section-4-UpReLU'              ReLU                          ReLU
    51   'Decoder-Section-4-DepthConcatenation'  Depth concatenation           Depth concatenatio
    52   'Decoder-Section-4-Conv-1'              Convolution                   64 3×3×128 convolu
    53   'Decoder-Section-4-ReLU-1'              ReLU                          ReLU
    54   'Decoder-Section-4-Conv-2'              Convolution                   64 3×3×64 convolut
    55   'Decoder-Section-4-ReLU-2'              ReLU                          ReLU
    56   'Final-ConvolutionLayer'                Convolution                   18 1×1×64 convolut
    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 or Download Pretrained Network

By default, the example downloads 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 network enables you to run the entire example without having to wait for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model by using the `trainNetwork` (Deep Learning Toolbox) function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 20 hours on an NVIDIA Titan X.

```
doTraining = false;
if doTraining
```

```
    net = trainNetwork(dsTrain,lgraph,options);
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save(fullfile(dataDir,"multispectralUnet-"+modelDateTime+".mat"),"net");

else
    trainedUnet_url = "https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat";
    downloadTrainedNetwork(trainedUnet_url,dataDir);
    load(fullfile(dataDir,"multispectralUnet.mat"));
end
```

You can now use the U-Net to semantically segment the multispectral image.

Perform Semantic Segmentation

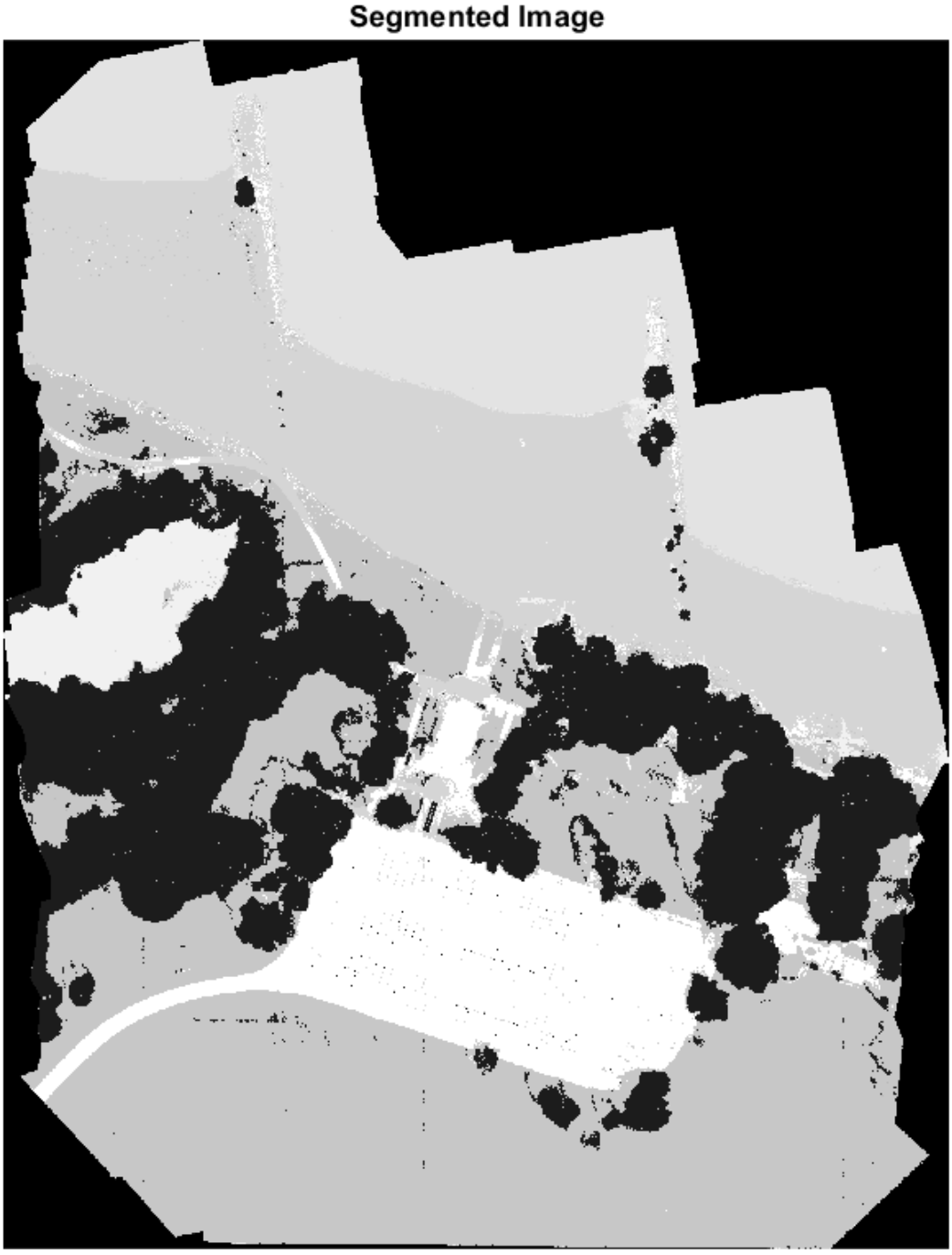
The example evaluates the network using the validation image because the data set includes ground truth labels for the validation image but not the test image.

To perform the forward pass on the trained network, use the `segmentMultispectralImage` helper function with the test data set. This function is attached to the example as a supporting file. The `segmentMultispectralImage` function performs segmentation on image patches using the `semanticseg` function.

```
predictPatchSize = [1024 1024];
segmentedImage = segmentMultispectralImage(val_data,net,predictPatchSize);
```

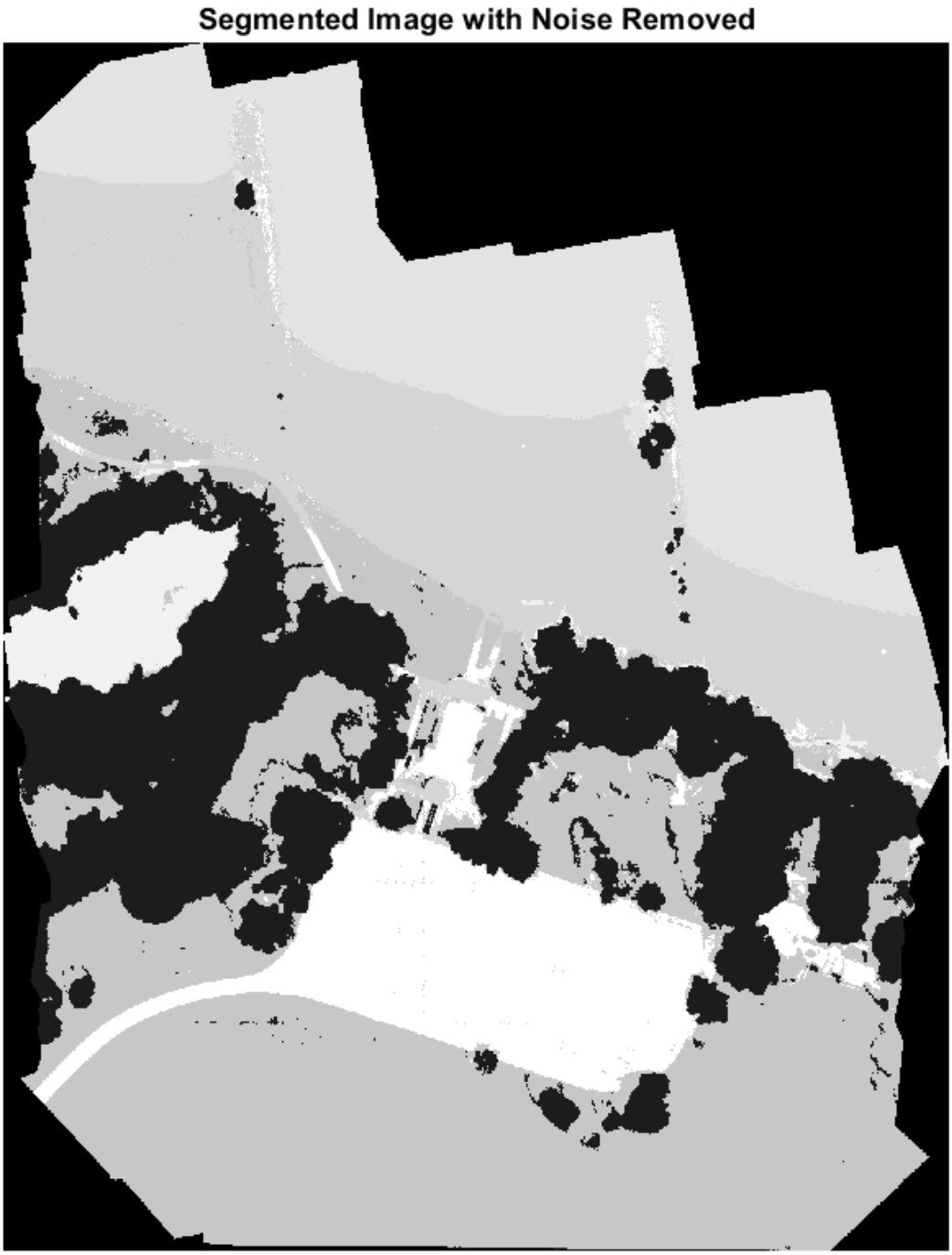
To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the validation data.

```
segmentedImage = uint8(val_data(:,:,7)~=0) .* segmentedImage;

figure
imshow(segmentedImage,[])
title("Segmented Image")
```



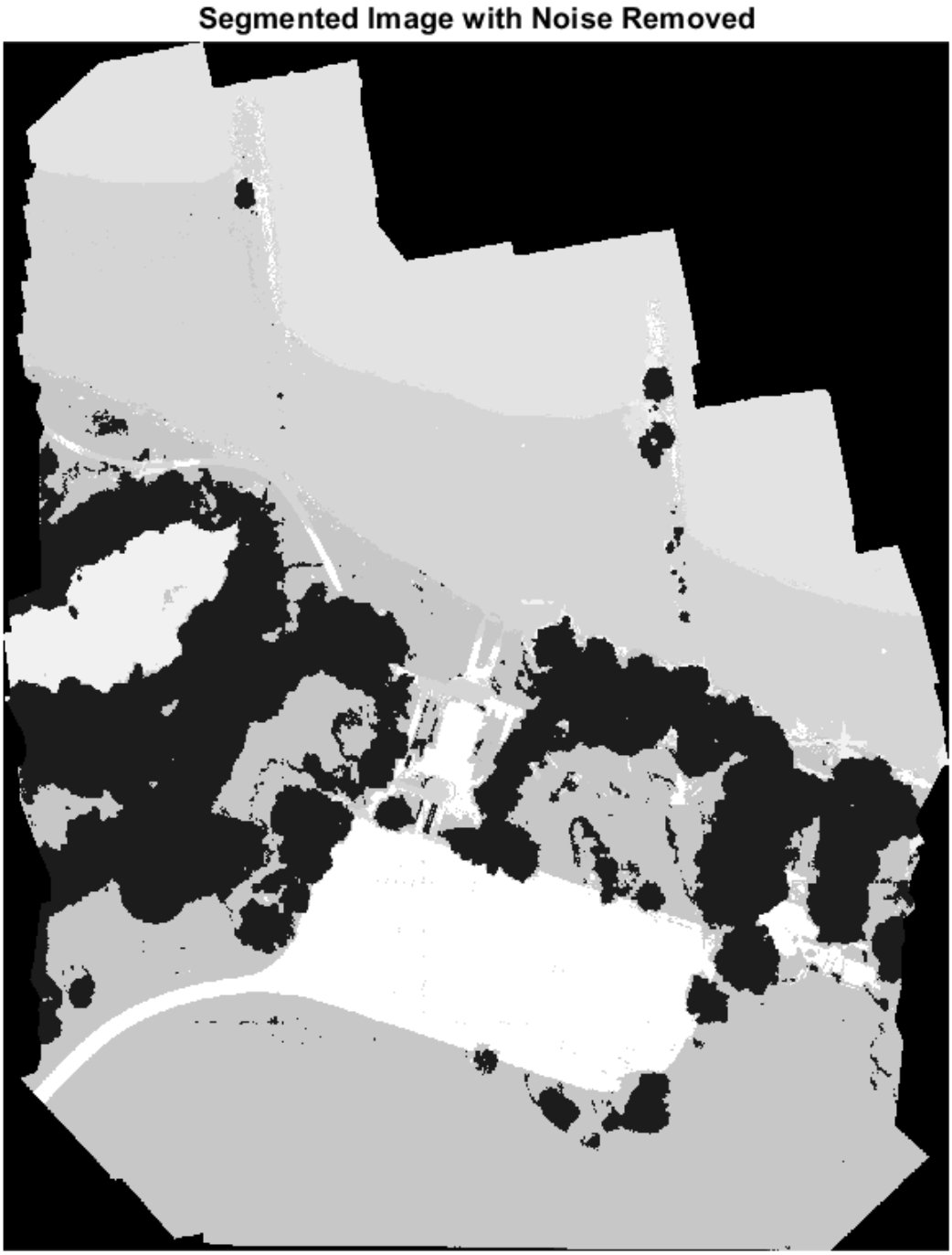
The output of semantic segmentation is noisy. Perform post image processing to remove noise and stray pixels. Use the `medfilt2` function to remove salt-and-pepper noise from the segmentation. Visualize the segmented image with the noise removed.

```
segmentedImage = medfilt2(segmentedImage,[7,7]);
imshow(segmentedImage,[]);
title("Segmented Image with Noise Removed")
```

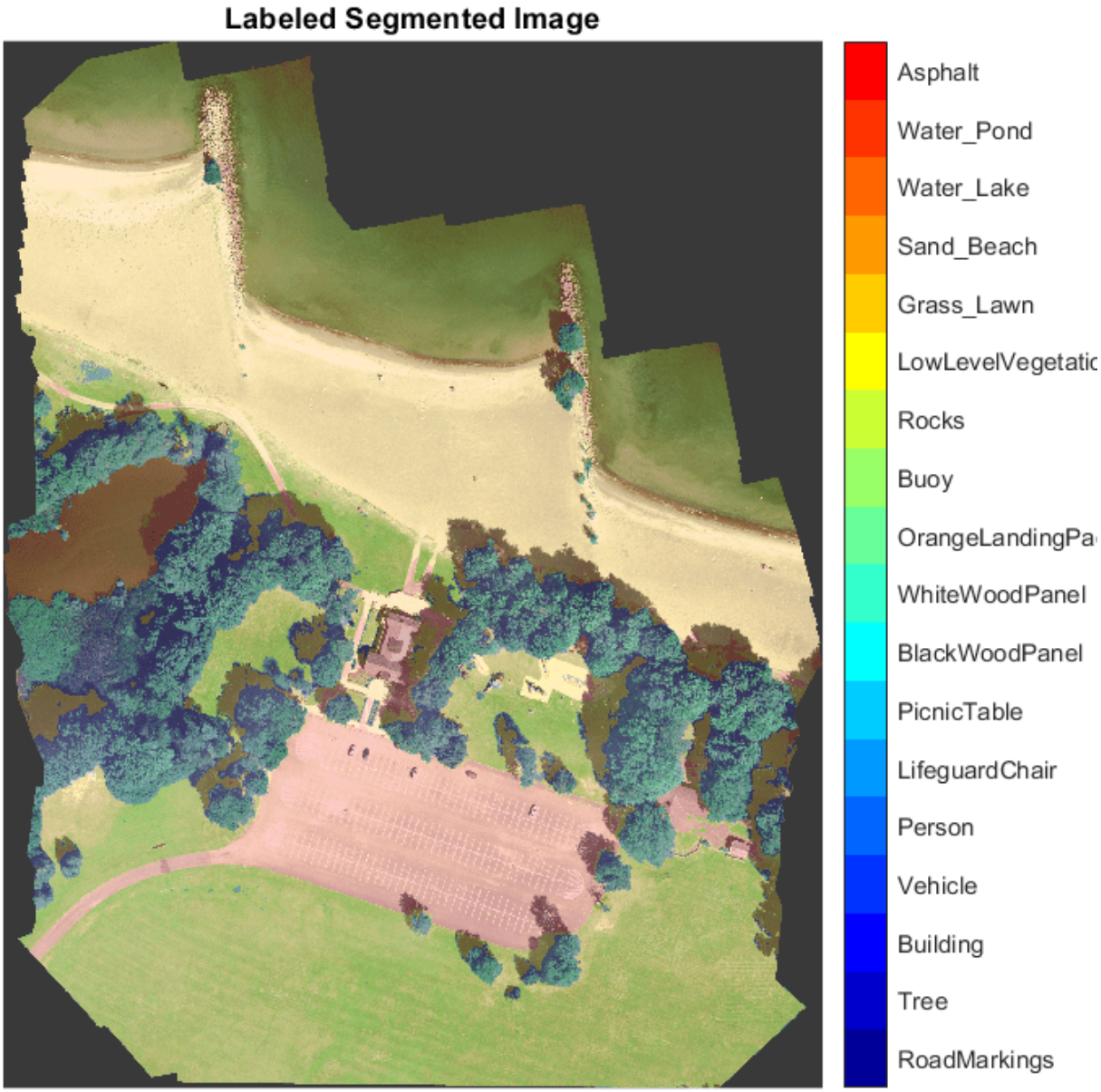


Overlay the segmented image on the histogram-equalized RGB validation image.

```
B = labeloverlay(histeq(val_data(:,:,[3 2 1])),segmentedImage,Transparency=0.8,Colormap=cmap);

figure
imshow(B)
```




```
title("Labeled Segmented Image")
colorbar(TickLabels=cellstr(classNames),Ticks=ticks,TickLength=0,TickLabelInterpreter="none");
colormap(cmap)
```



Save the segmented image and ground truth labels as PNG files. The example uses these files to calculate 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

`trainingOptions` | `trainNetwork` | `randomPatchExtractionDatastore` | `pixelLabelDatastore` | `semanticseg` | `evaluateSemanticSegmentation` | `imageDatastore` | `histeq` | `unetLayers`

More About

- “Getting Started with Semantic Segmentation Using Deep Learning” on page 17-62
- “Semantic Segmentation Using Deep Learning” on page 3-222
- “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.

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 Sample Test Set on page 3-0 section in this example.

Create a directory to store the BraTS data set.

```
imageDir = fullfile(tempdir,"BraTS");
```

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 = imageDir+filesep+"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.

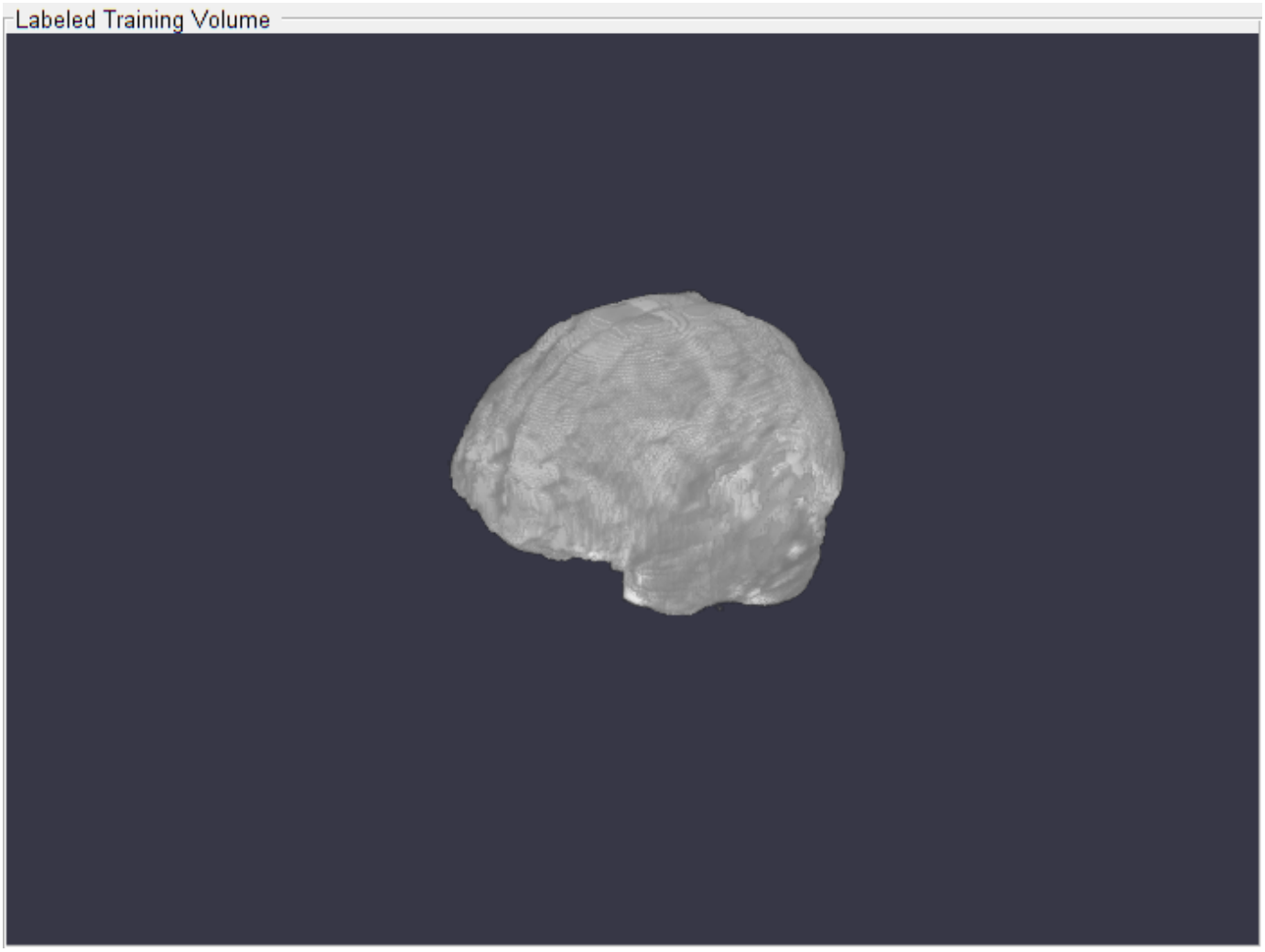
```
volLoc = fullfile(preprocessDataLoc,"imagesTr");
volds = imageDatastore(volLoc,FileExtensions=".mat",ReadFcn=@matRead);
```

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=@matRead);
```

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=@matRead);

lblLocVal = fullfile(preprocessDataLoc,"labelsVal");
```

```
pxdsVal = pixelLabelDatastore(lblLocVal,classNames,pixelLabelID, ...
    FileExtensions=".mat",ReadFcn=@matRead);

dsVal = randomPatchExtractionDatastore(voldsVal,pxdsVal,patchSize, ...
    PatchesPerImage=patchPerImage);
dsVal.MiniBatchSize = miniBatchSize;
```

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.

```
numChannels = 4;
inputPatchSize = [patchSize numChannels];
numClasses = 2;
[lgraph,outPatchSize] = unet3dLayers(inputPatchSize,numClasses,ConvolutionPadding="valid");
```

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.

```
dsTrain = transform(patchds,@(patchIn)augmentAndCrop3dPatch(patchIn,outPatchSize,"Training"));
dsVal = transform(dsVal,@(patchIn)augmentAndCrop3dPatch(patchIn,outPatchSize,"Validation"));
```

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 Sample Test Set

Download five sample test volumes and their corresponding labels from the BraTS data set using the `downloadBraTSSampleTestData` helper function [3 on page 3-0]. The helper function is attached to the example as a supporting file. The sample data enables you to perform segmentation on test data without downloading the full data set.

```
imageDir = fullfile(tempdir,"BraTS");
downloadBraTSSampleTestData(imageDir);
```

Specify the patch sizes and class information for the test data set.

```
inputPatchSize = [132 132 132 4];
outPatchSize = [44 44 44 2];
classNames = ["background","tumor"];
pixelLabelID = [0 1];
```

Train Network

By default, the example loads a pretrained 3-D U-Net network. 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 using the `trainNetwork` (Deep Learning Toolbox) function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). 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
    [net,info] = trainNetwork(dsTrain,lgraph,options);
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save("trained3DUNet-"+modelDateTime+".mat","net");
```

```
else
    trained3DUnet_url = "https://www.mathworks.com/supportfiles/vision/data/brainTumor3DUNetValid
    downloadTrainedNetwork(trained3DUnet_url,imageDir);
    load(imageDir+filesep+"brainTumor3DUNetValid.mat");
end
```

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 = preprocessDataLoc+filesep+"imagesTest";
    lblLocTest = preprocessDataLoc+filesep+"labelsTest";
else
    volLocTest = imageDir+filesep+"sampleBraTSTestSetValid"+filesep+"imagesTest";
    lblLocTest = imageDir+filesep+"sampleBraTSTestSetValid"+filesep+"labelsTest";
end
```

The `voldsTest` variable stores the ground truth test images. The `pxdsTest` variable stores the ground truth labels.

```
voldsTest = imageDatastore(volLocTest,FileExtensions=".mat",ReadFcn=@matRead);
pxdsTest = pixelLabelDatastore(lblLocTest,classNames,pixelLabelID, ...
    FileExtensions=".mat",ReadFcn=@matRead);
```

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};
    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
```

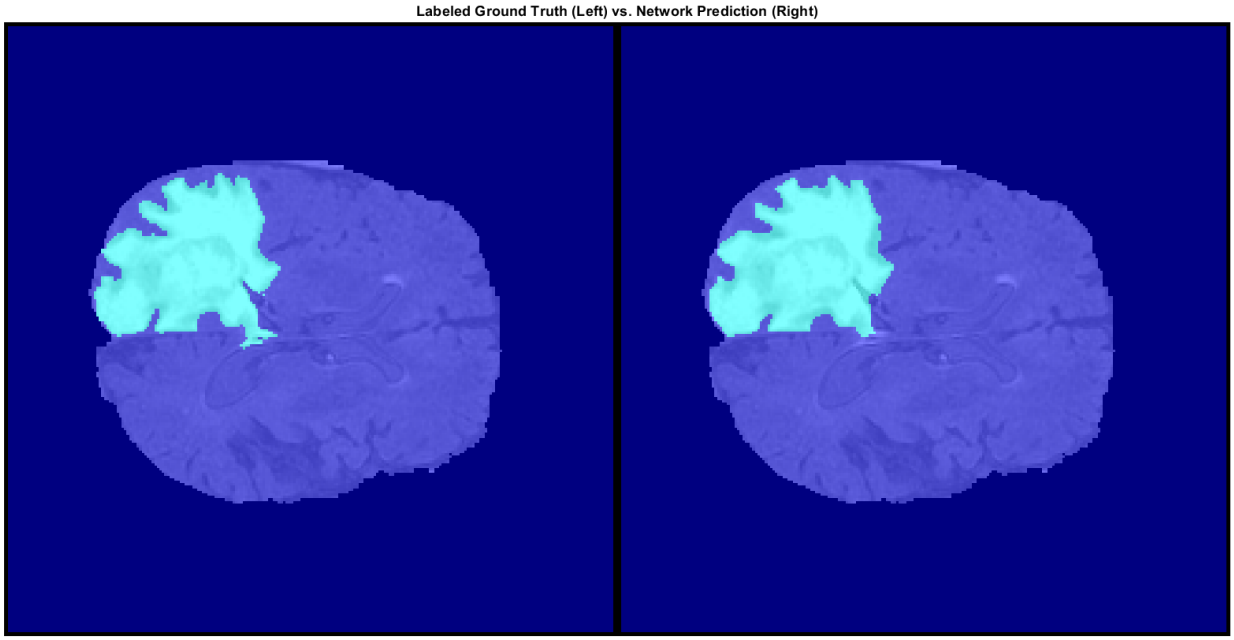
Compare Ground Truth Against Network Prediction

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);
```

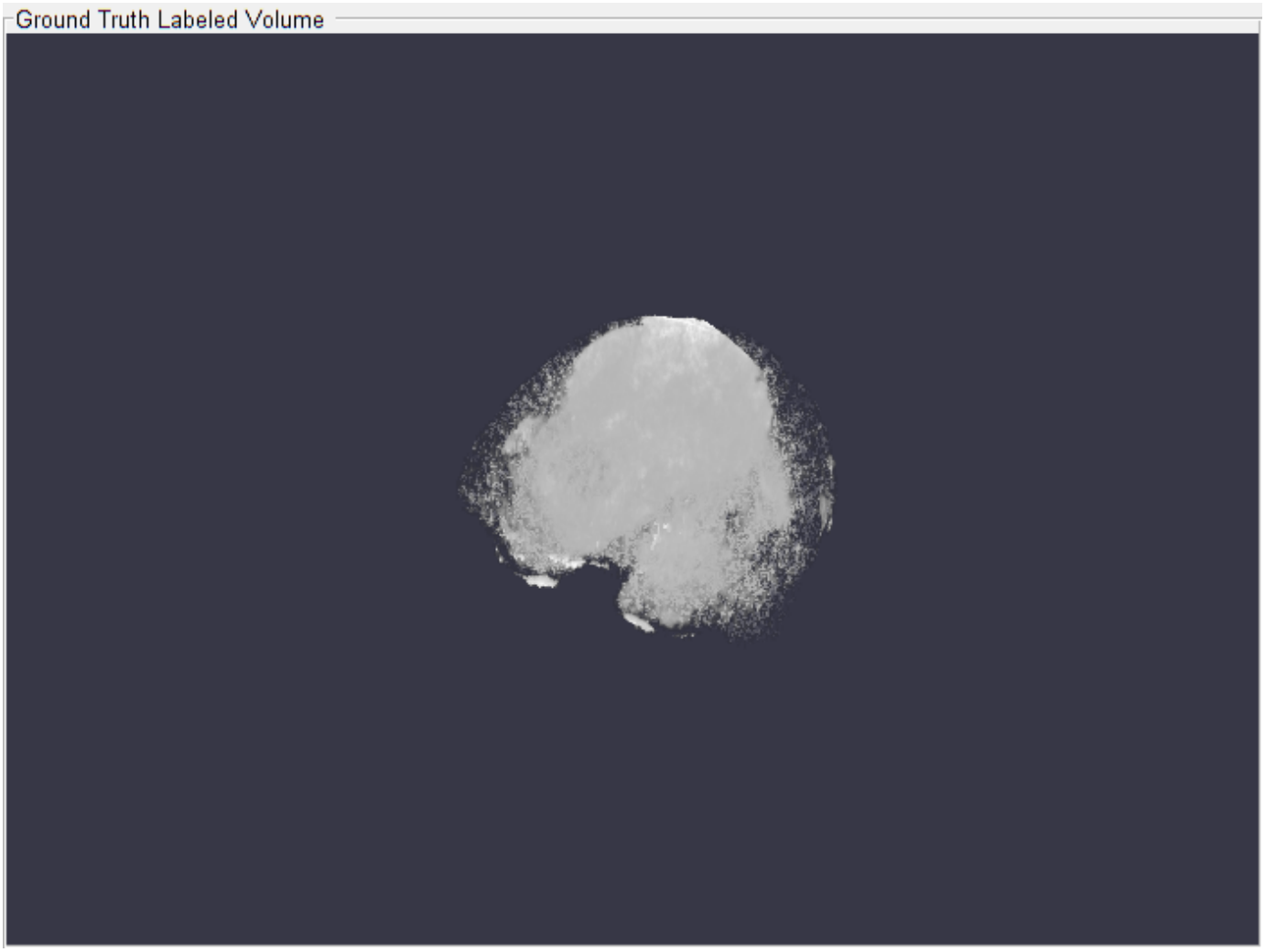
Display in a montage the center slice of the ground truth and predicted labels along the depth direction.

```
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)")
```



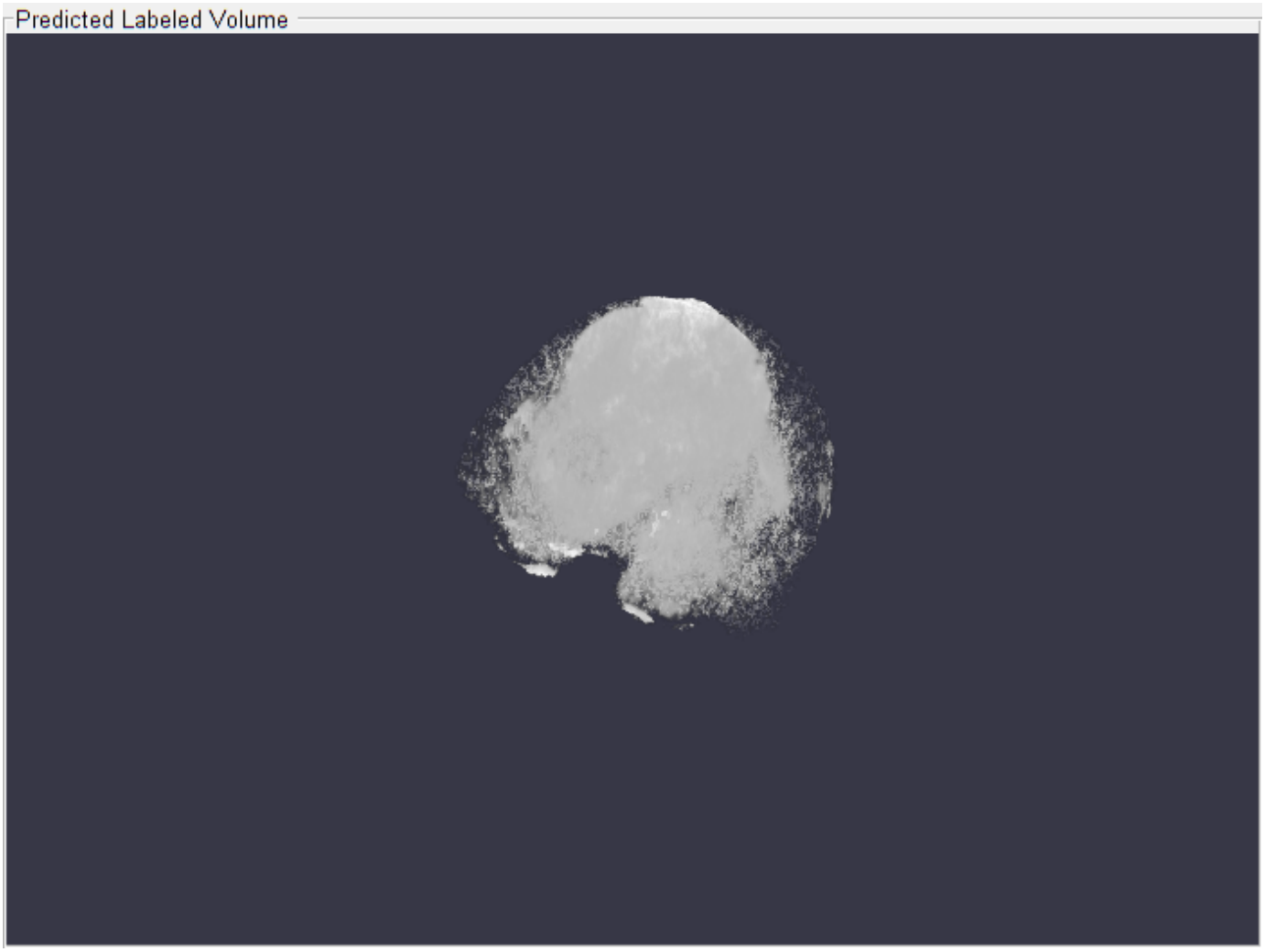
Display the ground-truth labeled volume using the `labelvolshow` function. Make the background fully transparent by setting the visibility of the background label (`1`) to `0`. Because the tumor is inside the brain tissue, make some of the brain voxels transparent, so that the tumor is visible. To make 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;
```

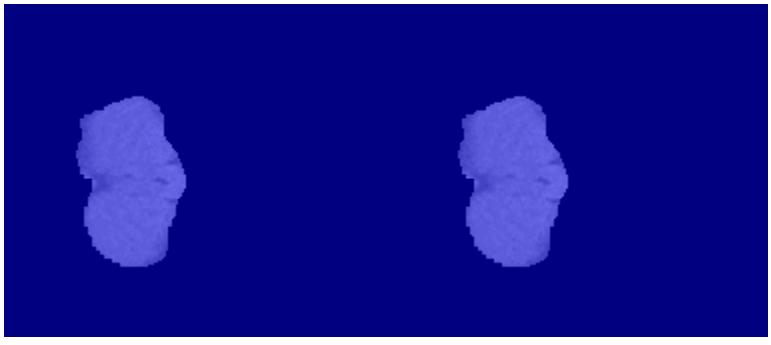


For the same volume, display the predicted labels.

```
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.

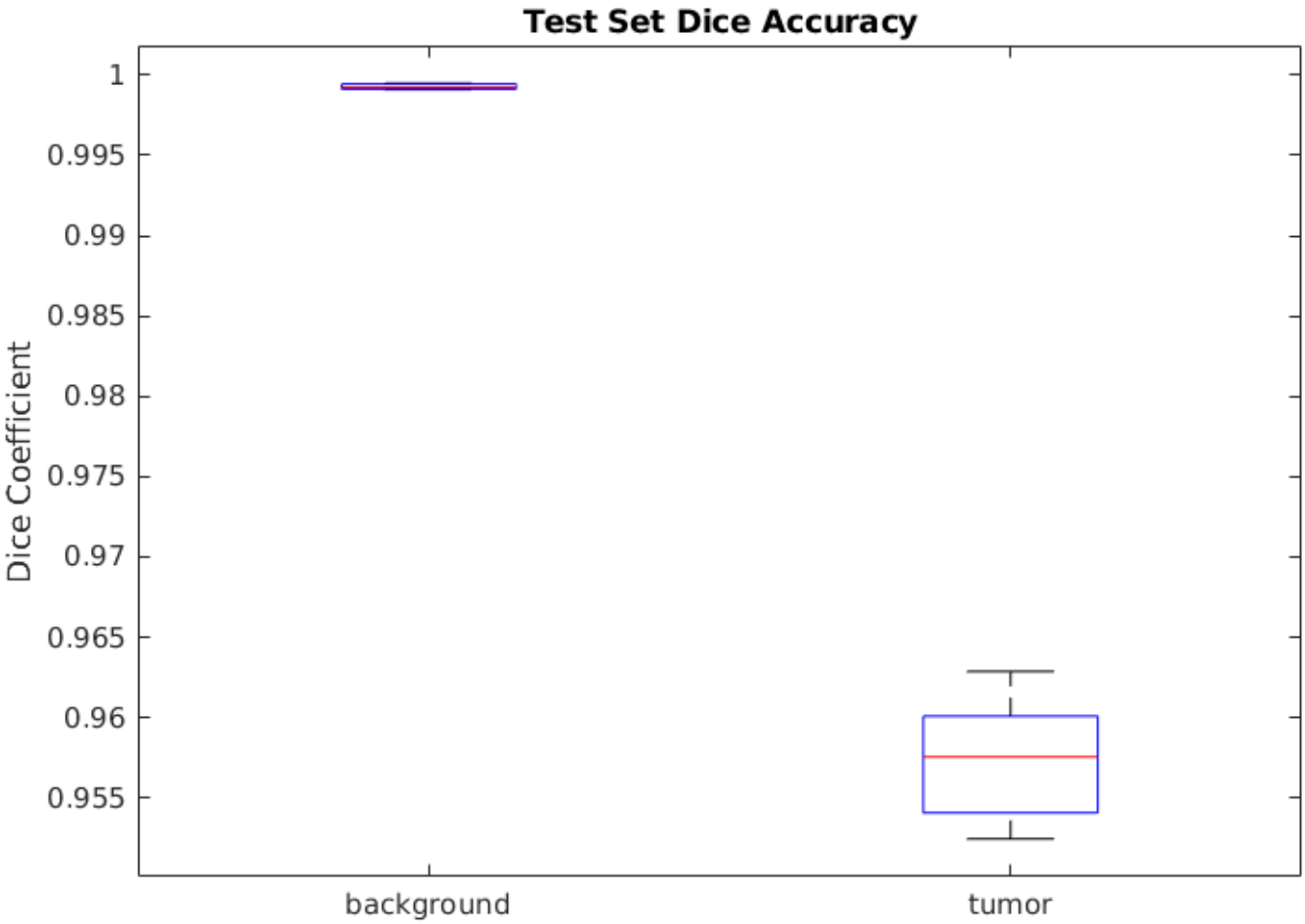
```
meanDiceBackground = mean(diceResult(:,1));
disp("Average Dice score of background across "+num2str(j)+ ...
    " test volumes = "+num2str(meanDiceBackground))
```

```
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 Sample Test Set on page 3-0 section of this example. The modified sample data set 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

`randomPatchExtractionDatastore` | `trainNetwork` | `trainingOptions` | `transform` | `pixelLabelDatastore` | `imageDatastore` | `semanticseg` | `dicePixelClassificationLayer`

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 collections.

```
imds = imageDatastore('MerchData','IncludeSubfolders',true,'LabelSource','foldernames');
```

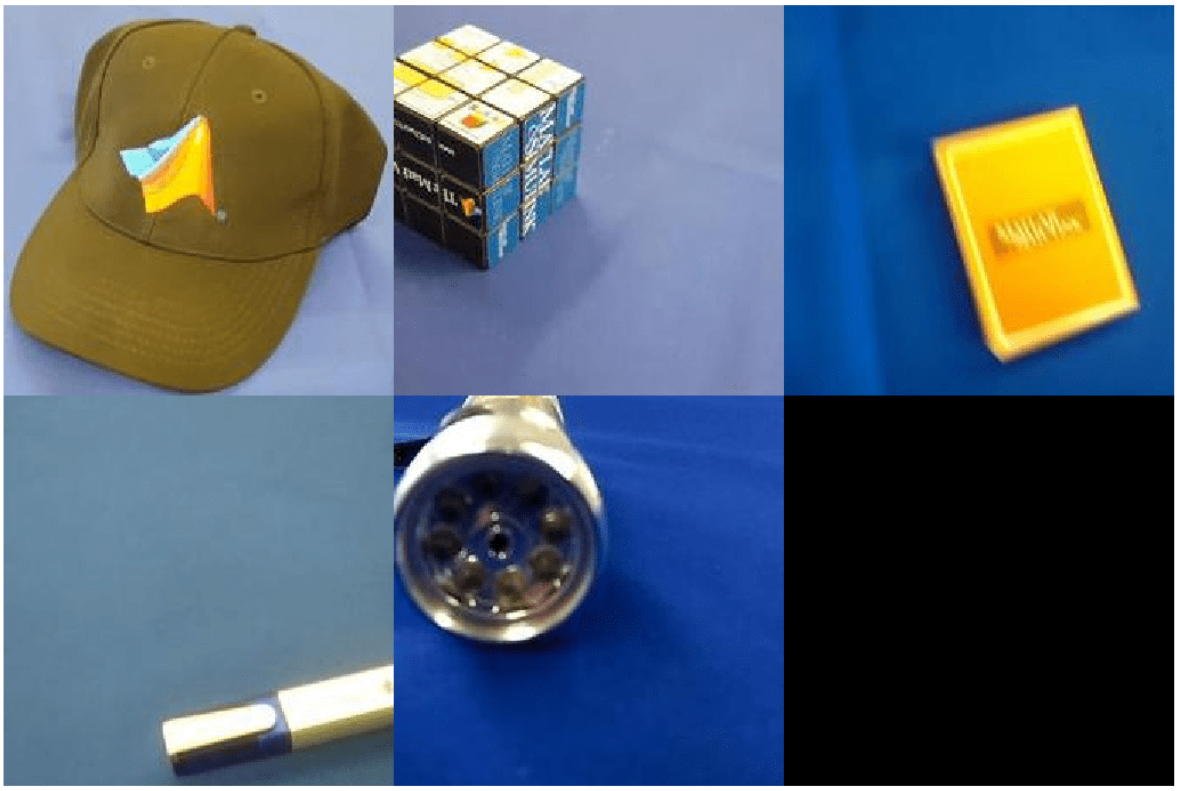
You can easily inspect the number of images per category as well as category labels as shown below:

```
tbl = countEachLabel(imds)
```

```
tbl=5×2 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);
```

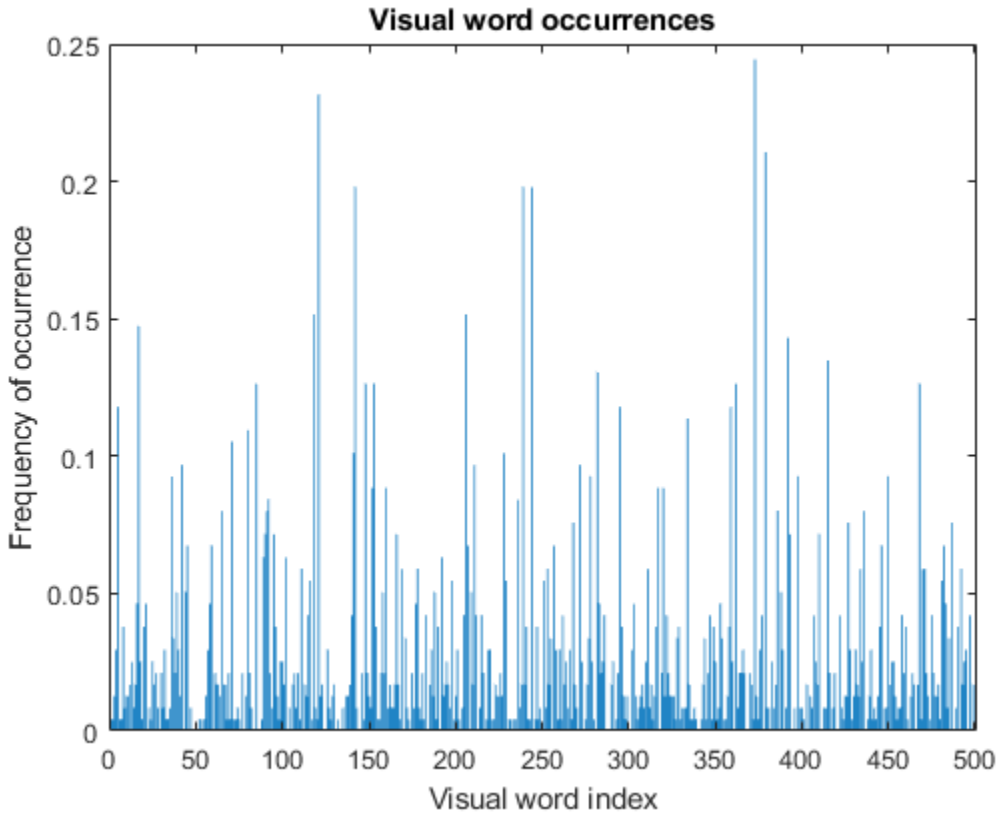
```
Creating Bag-Of-Features.
-------------------------
* Image category 1: MathWorks Cap
* Image category 2: MathWorks Cube
* Image category 3: MathWorks Playing Cards
* Image category 4: MathWorks Screwdriver
* Image category 5: MathWorks Torch
* Selecting feature point locations using the Grid method.
* 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.

* Creating a 500 word visual vocabulary.
* Number of levels: 1
* Branching factor: 500
* Number of clustering steps: 1

* [Step 1/1] Clustering vocabulary level 1.
* Number of features          : 112895
* Number of clusters          : 500
* Initializing cluster centers...100.00%.
* Clustering...completed 33/100 iterations (~0.29 seconds/iteration)...converged in 33 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);
```

```
Encoding images using Bag-Of-Features.
--------------------------------------
* Encoding an image...done.
```

```
% 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:


                                                                  PREDICTED
KNOWN                      | MathWorks Cap   MathWorks Cube   MathWorks Playing Cards   MathWorks
---------------------------------------------------------------------------------------------------
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
--------------------------------------------------------------------------------------------------
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);

Encoding images using Bag-Of-Features.
----------------------------------------
* Encoding an image...done.
```

```
% 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 3-275 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 is highly recommended for running this example. Use of a GPU requires the Parallel Computing Toolbox™. For information about the supported compute capabilities, see “GPU Support by Release” (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=5×2 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=5×2 table
      Label       Count
    __________    _____

    daisy          100
    dandelion      100
    roses          100
    sunflowers     100
    tulips         100
```

Load pretrained Network

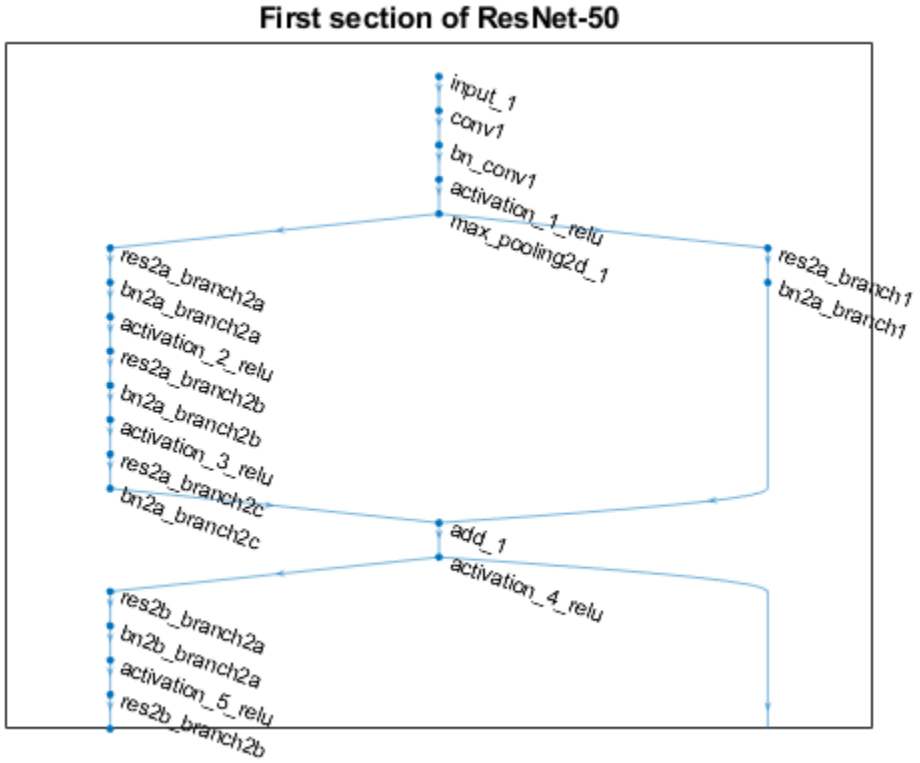
There are several pretrained networks that have gained popularity. Most of these have been trained on the ImageNet dataset, which has 1000 object categories and 1.2 million training images[1]. "ResNet-50" is one such model and can be loaded using the `resnet50` function from Neural Network Toolbox™. 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 the 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: [1000×1 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.

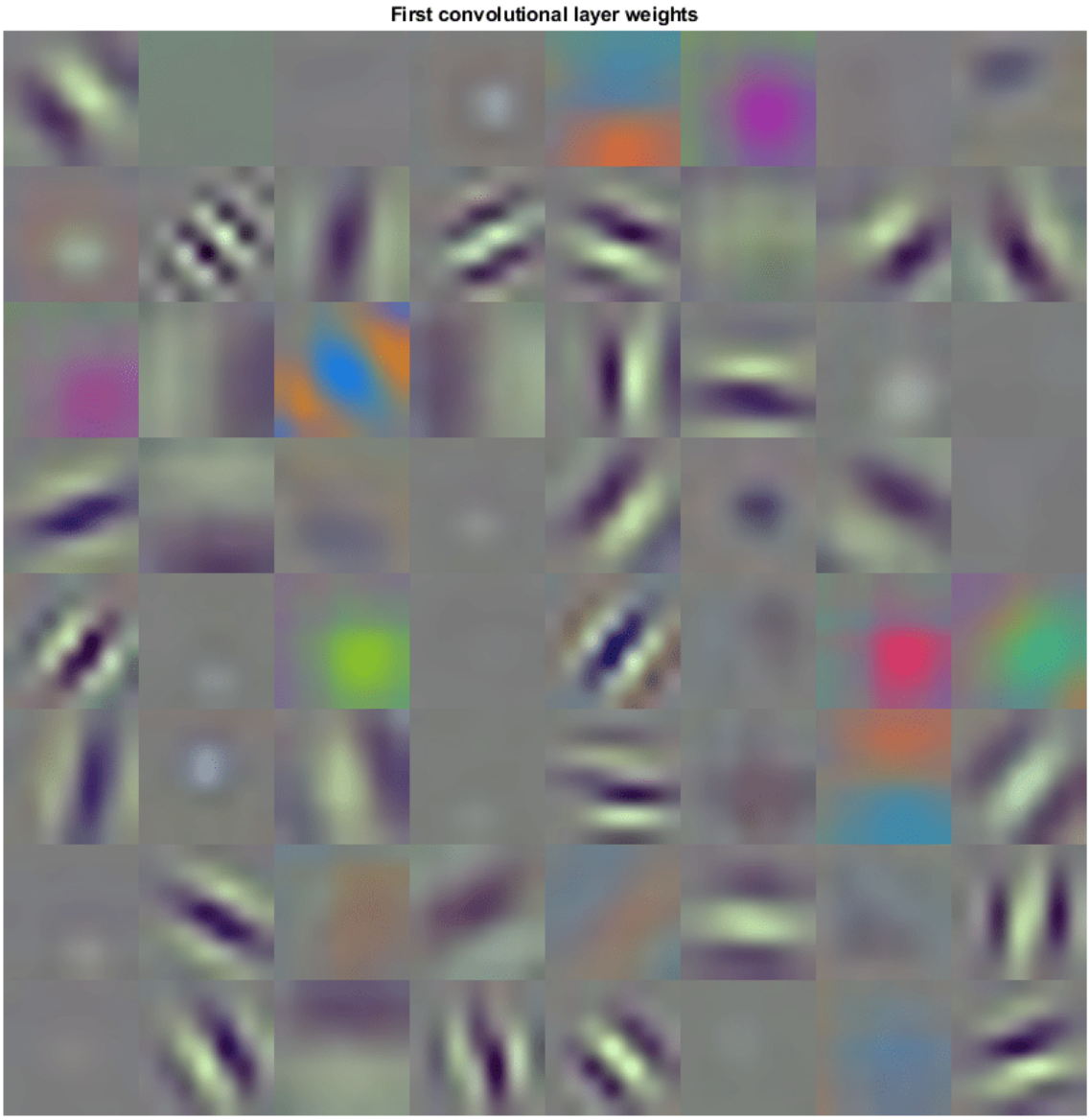
```
% Create augmentedImageDatastore from training and test sets to resize
% images in imds to the size required by the network.
imageSize = net.Layers(1).InputSize;
augmentedTrainingSet = augmentedImageDatastore(imageSize, trainingSet, 'ColorPreprocessing', 'gra
augmentedTestSet = augmentedImageDatastore(imageSize, 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 = 5×5
```

```
    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

`alexnet` | `countEachLabel` | `imageInputLayer` | `convolution2dLayer` | `reluLayer` | `maxPooling2dLayer` | `classificationLayer` | `fullyConnectedLayer` | `activations` | `predict` | `deepDreamImage` | `fitcecoc` | `confusionmat`

More About

- “Image Category Classification Using Bag of Features” on page 3-275
- “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.

%   Copyright 2014-2020 The MathWorks, Inc.

[~,~,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');

    % Specify the number of levels and branching factor of the vocabulary
    % tree used within bagOfFeatures. Empirical analysis is required to
    % choose optimal values.
    numLevels = 1;
    numBranches = 5000;

    % Create a custom bag of features using the 'CustomExtractor' option.
    colorBag = bagOfFeatures(trainingSet, ...
        'CustomExtractor', @exampleBagOfFeaturesColorExtractor, ...
        'TreeProperties', [numLevels numBranches]);
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 = 5×1

    0.4776
    0.2138
    0.1386
    0.1382
```

```
    0.1317
```

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 Classifcation 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: [25×1 nnet.cnn.layer.Layer]
    Connections: [24×2 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 3-348.

```
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  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |     RMSE     |     Loss     |      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.
*********************************************************************
```

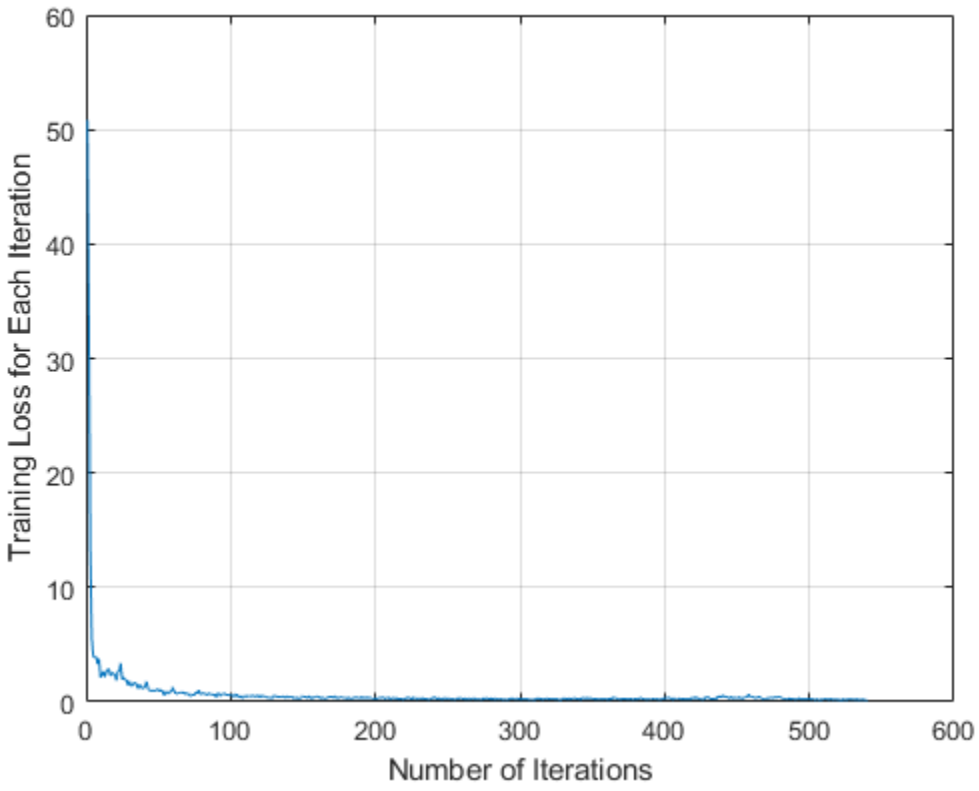
Inspect the properties of the detector.

```
detector
```

```
detector =
  yolov2ObjectDetector with properties:

            ModelName: 'vehicle'
              Network: [1×1 DAGNetwork]
    TrainingImageSize: [128 128]
         AnchorBoxes: [4×2 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.

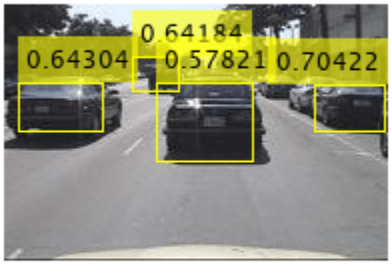
```
img = imread('detectcars.png');
```

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)
```


0.64184
0.64304
0.57821
0.70422

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.

```
pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/deeplearning/models/yolov2/tiny_yo
pretrainedNetTar = 'yolov2Tiny.tar';
if ~exist(pretrainedNetTar,'file')
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetTar,pretrainedURL);
end
```

Extract YOLO v2 Network

Untar the downloaded file to extract the Tiny YOLO v2 network. Load the `'Model.onnx'` model from `tiny_yolov2` folder, which is an ONNX YOLO v2 network pretrained on the PASCAL VOC data set [2] on page 3-0 . The network can detect objects from 20 different classes [3] on page 3-0 .

```
onnxfiles = untar(pretrainedNetTar);
pretrainedNet = fullfile('tiny_yolov2','Model.onnx');
```

Import ONNX YOLO v2 Layers

Use the `importONNXLayers` function to import the downloaded network.

```
lgraph = importONNXLayers(pretrainedNet,'ImportWeights',true);
```

importONNXLayers adds regression layer at the end by default. Remove the last regression layer added by importONNXLayers as `yolov2ObjectDetector` expects YOLO v2 detection network to end with `yolov2OutputLayer`. For more information on YOLO v2 detection network, see “Getting Started with YOLO v2” on page 17-37.

```
lgraph = removeLayers(lgraph,'RegressionLayer_grid');
```

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

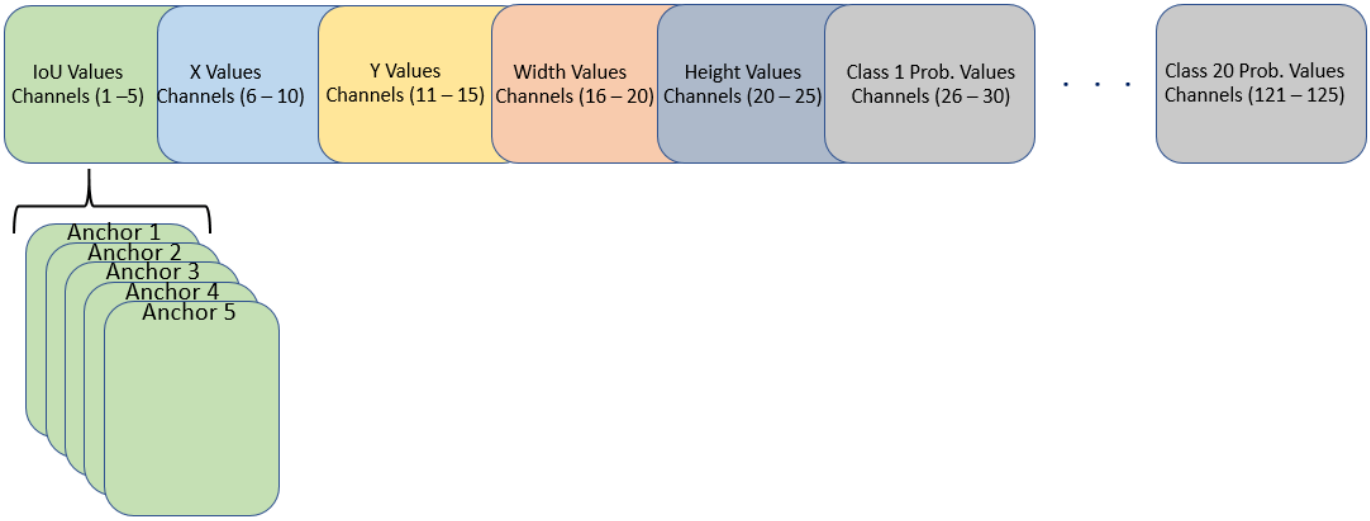
YOLO v2 uses predefined anchor boxes to predict object location. The anchor boxes used in the imported network are defined in the Tiny YOLO v2 network configuration file [4] on page 3-0 . The ONNX anchors are defined with respect to the output size of the final convolution layer, which is 13-by-13. To 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 = 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:

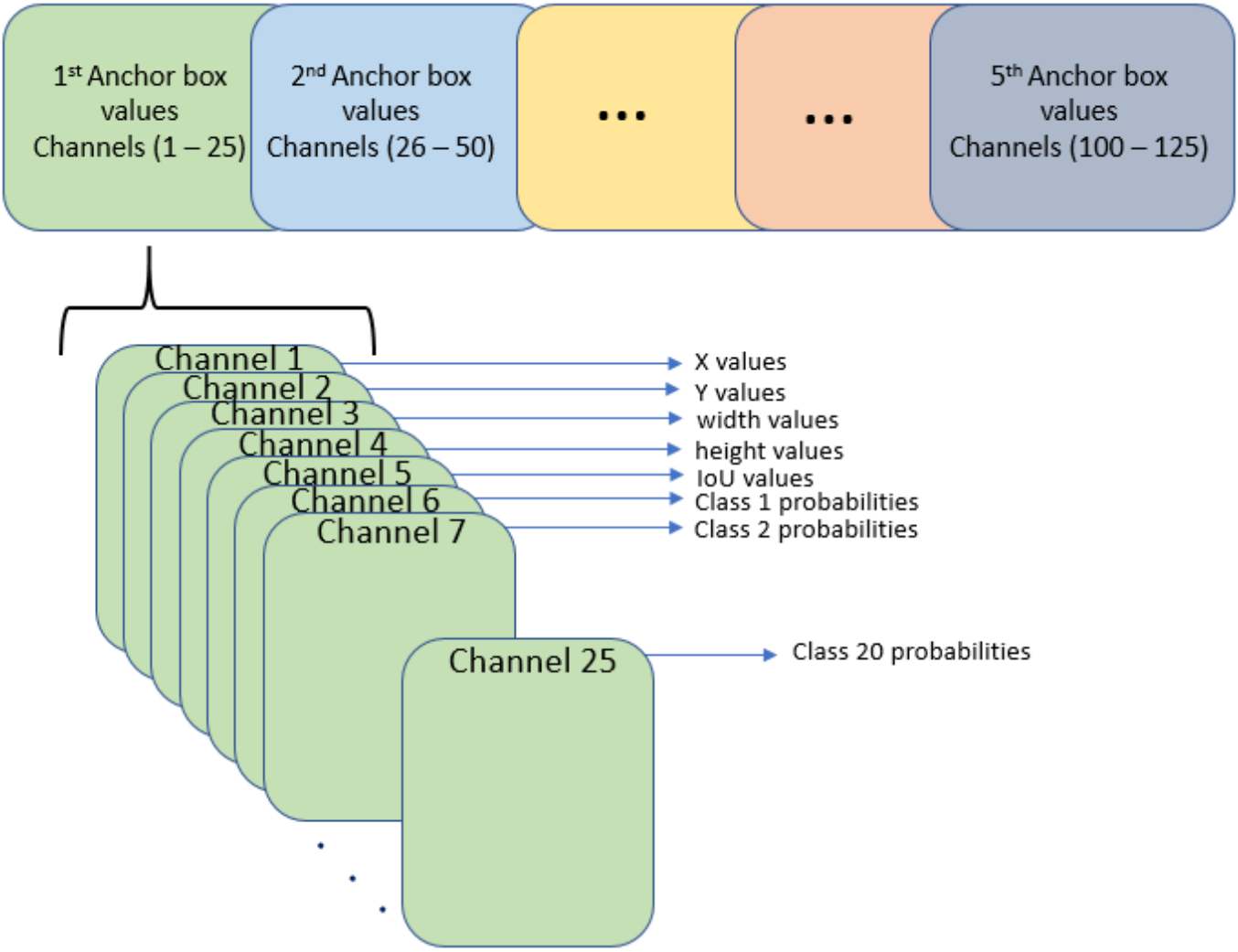
- Channels 1 to 5 - IoU values for five anchors
- Channels 6 to 10 - X values for five anchors
- Channels 11 to 15 - Y values for five anchors
- Channels 16 to 20 - Width values for five anchors
- Channels 21 to 25 - Height values for five anchors
- Channels 26 to 30 - Class 1 probability values for 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,'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: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'image'}
    OutputNames: {'yolov2Output'}
```

```
yolov2Detector = yolov2ObjectDetector(net)
```

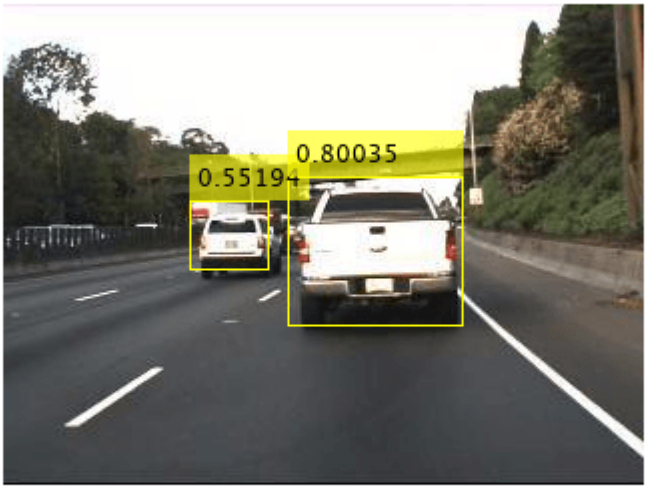
```
yolov2Detector =
  yolov2ObjectDetector with properties:
```

```
          ModelName: 'importedNetwork'
            Network: [1×1 DAGNetwork]
    TrainingImageSize: [416 416]
        AnchorBoxes: [5×2 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('highway.png');
% Convert image to BGR format.
Ibgr = cat(3,I(:,:,3),I(:,:,2),I(:,:,1));
[bboxes, scores, labels] = detect(yolov2Detector, Ibgr);
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 regared 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.

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 License." https://github.com/onnx/onnx/blob/master/LICENSE.

[3] 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.

[4] "yolov2-tiny-voc.cfg" https://github.com/pjreddie/darknet/blob/master/cfg/yolov2-tiny-voc.cfg.

References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." *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

[3] Tiny YOLO v2 Model 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*. Number 2 (June 2010): 303-38. https://doi.org/10.1007/s11263-009-0275-4.

[5] 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.

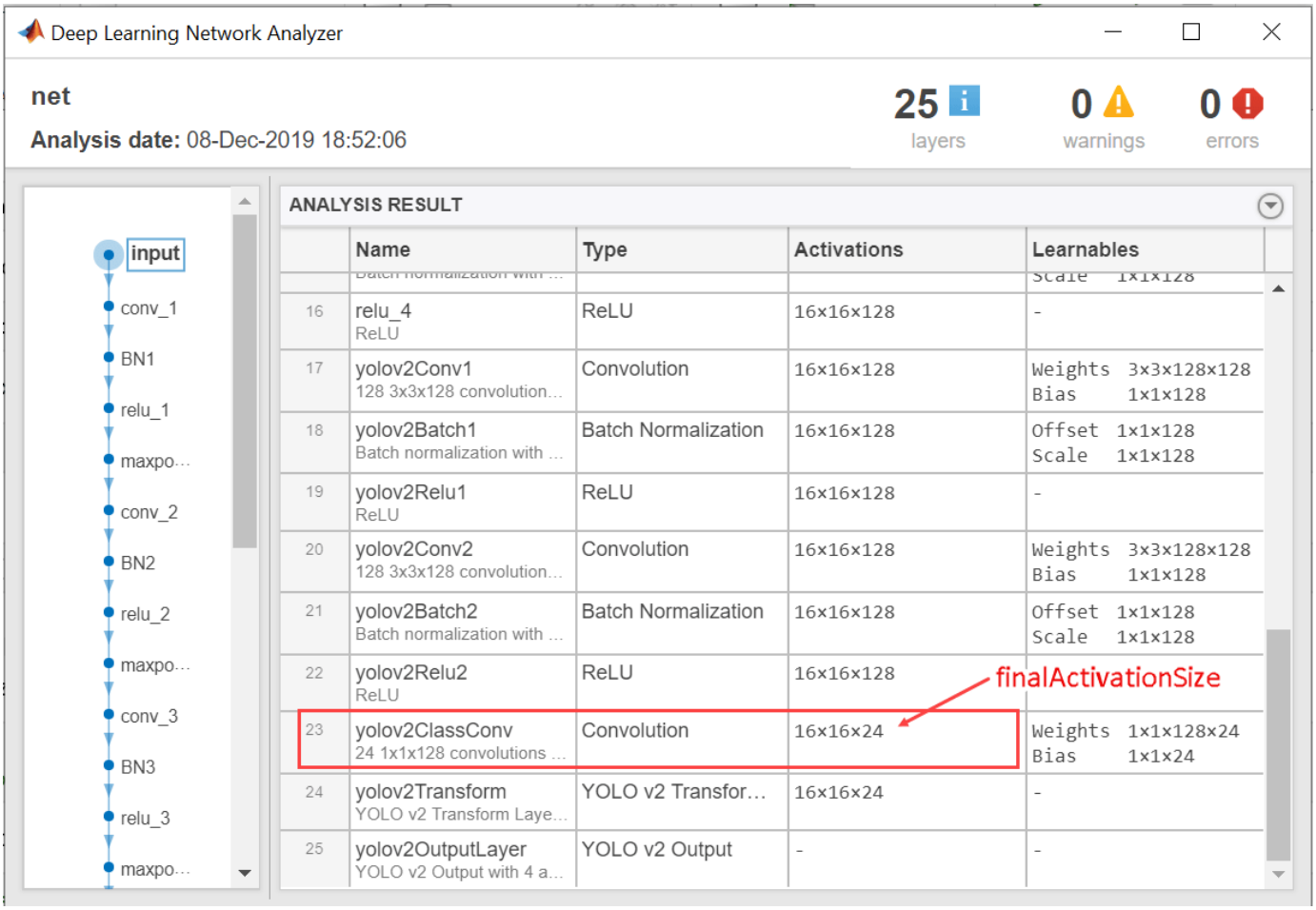
```
inputImageSize = net.Layers(1,1).InputSize;
```

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);
```




```
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.

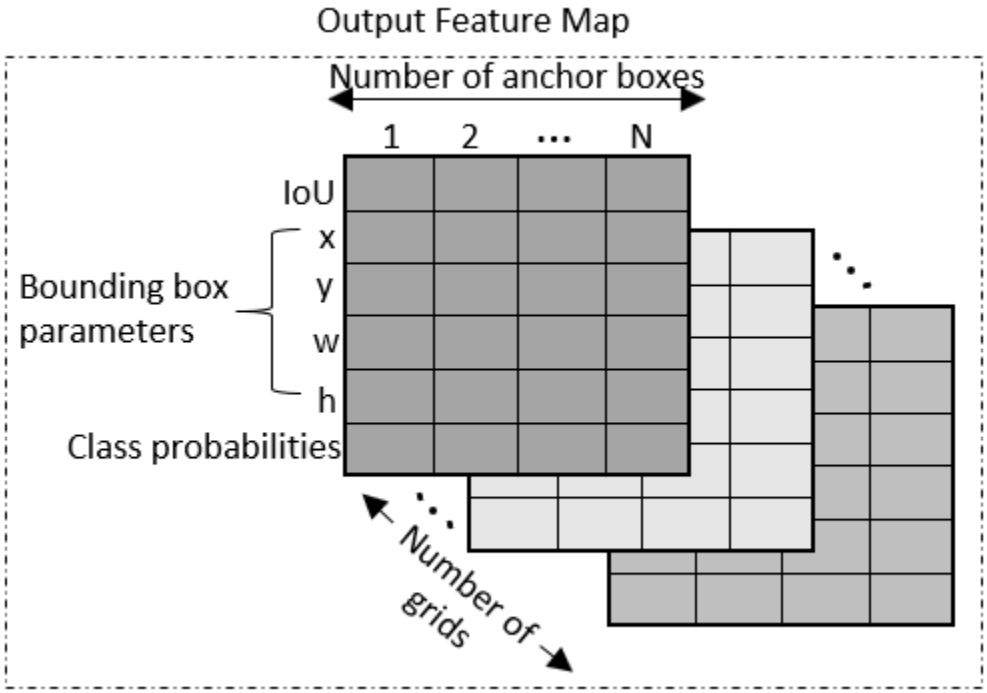
Extract Predictions from Output of ONNX Model

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,anchorB
```

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,finalActivationsSi

% 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','RatioTy

end

function bBoxes = rescaleBbox(xyPred,whPred,anchorBoxes,finalActivationsSize,inputImageSize)

% To rescale the bounding box parameters, compute the scaling factor by using the network paramet
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) = bw;
```

```
            bBoxes(4,anchorIdx,ind) = bh;
        end
    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 17-32.

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: 295×1 cell array of character vectors

    vehicle: 295×1 cell
```

Visualize Ground Truth Box Distribution

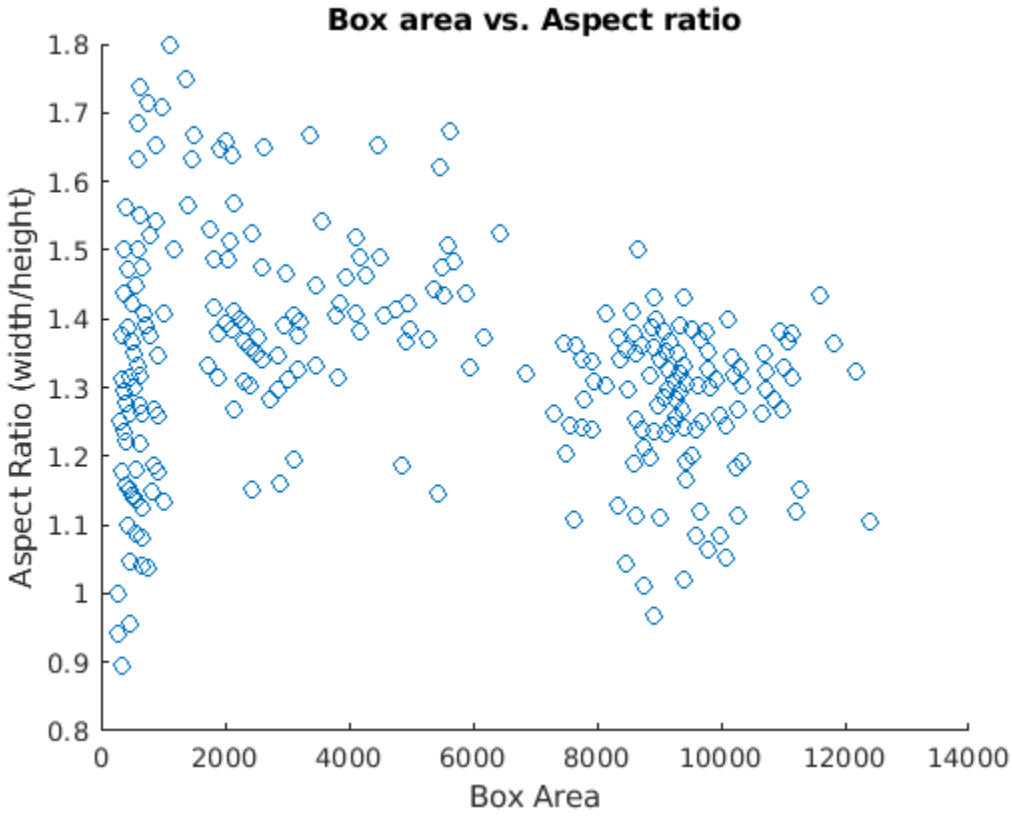
Visualize the labeled boxes to better understand the range of object sizes present in the data set.

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 = 5×2

    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 - bboxOverlapRatio(allBoxes,boxInCluster)`.

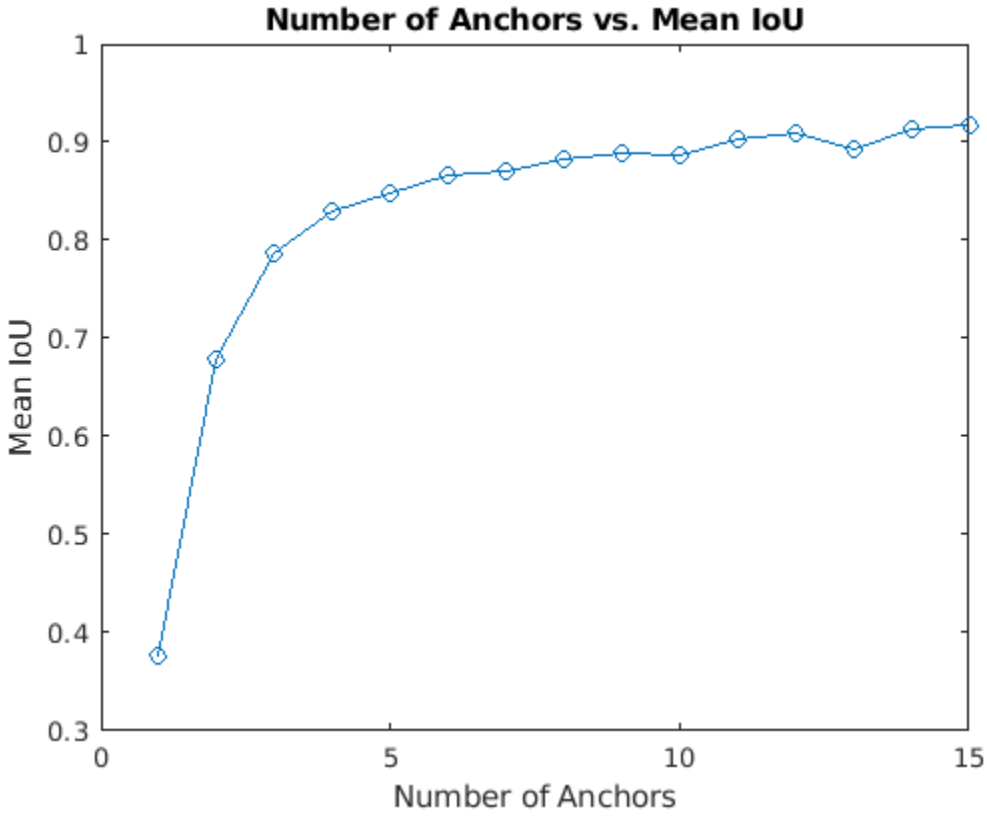
```
meanIoU
```

```
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

[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. 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.

Note: This example requires the Computer Vision Toolbox™ Model for YOLO v3 Object Detection. You can install the Computer Vision Toolbox Model for YOLO v3 Object Detection from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

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
    preTrainedDetector = downloadPretrainedYOLOv3Detector();
end
```

Load Data

This example uses a small labeled data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. 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 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 60% 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);
```

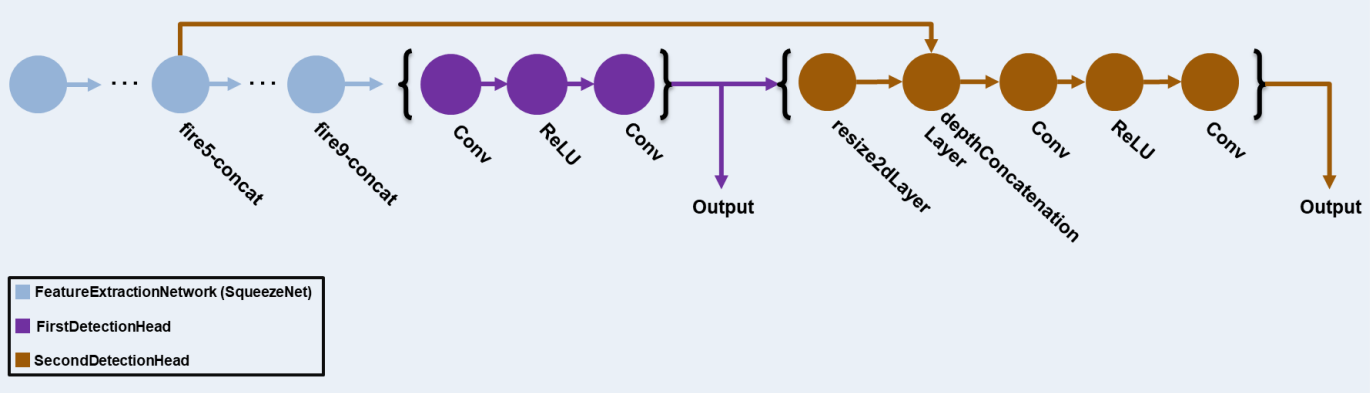
```
    augmentedData{k} = insertShape(data{1,1}, 'Rectangle', data{1,2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData, 'BorderSize', 10)
```

Define YOLO v3 Object Detector

The YOLO v3 detector 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 detector uses anchor boxes estimated using training data to have better initial priors corresponding to the type of data set and to help the detector learn to predict the boxes accurately. For information about anchor boxes, see “Anchor Boxes for Object Detection” on page 17-32.

The YOLO v3 network present in the YOLO v3 detector is illustrated in the following diagram.

You can use Deep Network Designer (Deep Learning Toolbox) to create the network shown in the diagram.



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];
```

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 3-319. 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;
[anchors, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchors = 6×2

    41    34
   163   130
    98    93
   144   125
    33    24
    69    66


meanIoU = 0.8507
```

Specify `anchorBoxes` to use in both the detection heads. `anchorBoxes` is a cell array of [Mx1], where M denotes the number of detection heads. Each detection head consists of a [Nx2] matrix of `anchors`, where N is the number of anchors to use. Select `anchorBoxes` for each detection head based on the feature map size. Use larger `anchors` at lower scale and smaller `anchors` at higher scale. To do so, sort the `anchors` 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 = anchors(:, 1).*anchors(:, 2);
[~, idx] = sort(area, 'descend');
anchors = anchors(idx, :);
anchorBoxes = {anchors(1:3,:)
    anchors(4:6,:)
    };
```

Load the SqueezeNet network pretrained on Imagenet data set and then specify the class names. You can also choose to load a different pretrained network trained on COCO data set such as `tiny-yolov3-coco` or `darknet53-coco` or Imagenet data set such as MobileNet-v2 or ResNet-18. YOLO v3 performs better and trains faster when you use a pretrained network.

```
baseNetwork = squeezenet;
classNames = trainingDataTbl.Properties.VariableNames(2:end);
```

Next, create the `yolov3ObjectDetector` object by adding the detection network source. Choosing the optimal detection network source requires trial and error, and you can use `analyzeNetwork` to find the names of potential detection network source within a network. For this example, use the `fire9-concat` and `fire5-concat` layers as `DetectionNetworkSource`.

```
yolov3Detector = yolov3ObjectDetector(baseNetwork, classNames, anchorBoxes, 'DetectionNetworkSou
```

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 by changing the classNames and anchorBoxes.

Preprocess Training Data

Preprocess the augmented training data to prepare for training. The `preprocess` method in `yolov3ObjectDetector`, applies the following preprocessing operations to the input data.

- Resize the images to the network input size by maintaining the aspect ratio.
- Scale the image pixels in the range `[0 1]`.

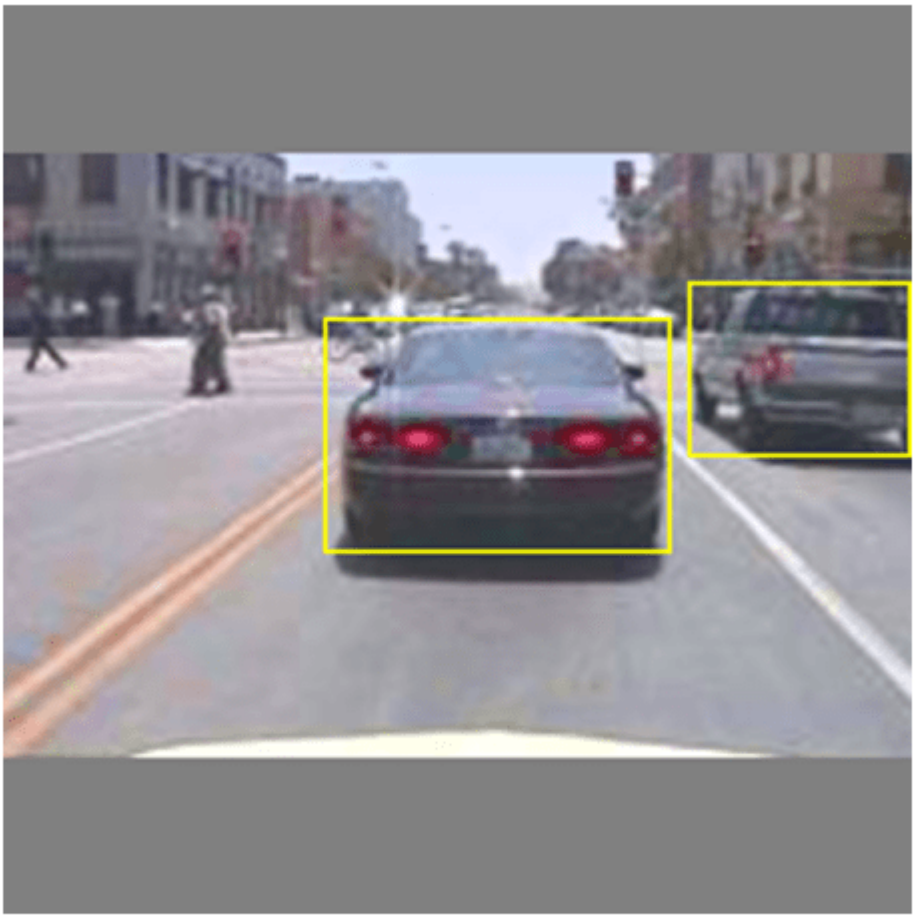
```
preprocessedTrainingData = transform(augmentedTrainingData, @(data)preprocess(yolov3Detector, da
```

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);
```

Specify Training Options

Specify these training options.

- Set the number of epochs to be 80.
- 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 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 = 80;
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. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

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"]);
```

Create the training progress plotter using supporting function `configureTrainingProgressPlotter` to see the plot while training the detector object with a custom training loop.

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 detector parameters using the `sgdmupdate` function.
- Update the `state` parameters of detector 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

    % 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, yolov3Detector, XTrain, YTra

            % Apply L2 regularization.
            gradients = dlupdate(@(g,w) g + l2Regularization*w, gradients, yolov3Detector.Learnab

            % Determine the current learning rate value.
            currentLR = piecewiseLearningRateWithWarmup(iteration, epoch, learningRate, warmupPe

            % Update the detector learnable parameters using the SGDM optimizer.
            [yolov3Detector.Learnables, velocity] = sgdmupdate(yolov3Detector.Learnables, gradie

            % Update the state parameters of dlnetwork.
            yolov3Detector.State = state;

            % Display progress.
            displayLossInfo(epoch, iteration, currentLR, lossInfo);

            % Update training plot with new points.
            updatePlots(lossPlotter, learningRatePlotter, iteration, currentLR, lossInfo.totalLos
        end
    end
else
    yolov3Detector = preTrainedDetector;
end
```

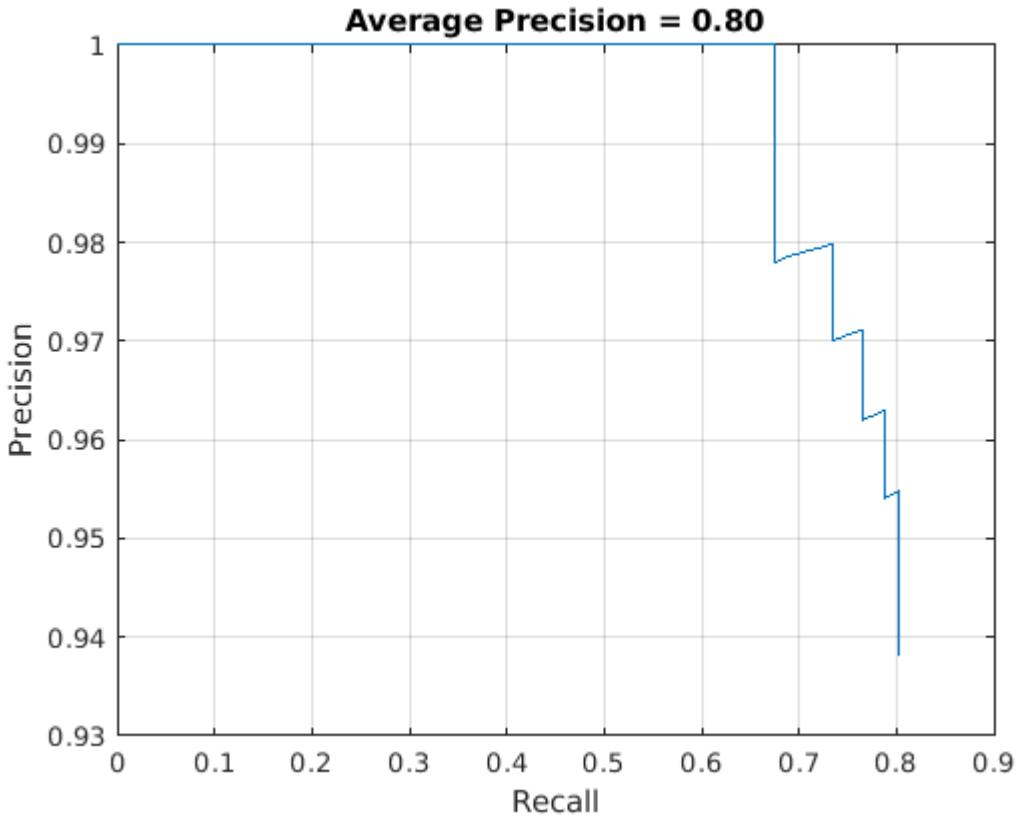
Evaluate Model

Computer Vision 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).

```
results = detect(yolov3Detector,testData,'MiniBatchSize',8);

% Evaluate the object detector using Average Precision metric.
[ap,recall,precision] = evaluateDetectionPrecision(results,testData);
```

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 detector for object detection.

```
% Read the datastore.
data = read(testData);

% Get the image.
I = data{1};

[bboxes,scores,labels] = detect(yolov3Detector,I);

% Display the detections on image.
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);

figure
imshow(I)
```

Supporting Functions

Model Gradients Function

The function `modelGradients` takes the `yolov3ObjectDetector` object, a mini-batch of input data `XTrain` with corresponding ground truth boxes `YTrain`, the specified penalty threshold as input arguments and returns the gradients of the loss with respect to the learnable parameters in `yolov3ObjectDetector`, the corresponding mini-batch loss information, 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 `forward` method.
- 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 `anchorBoxGenerator` method of `yolov3ObjectDetector`.

- 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(detector, XTrain, YTrain, penaltyThreshold)
inputImageSize = size(XTrain,1:2);

% Gather the ground truths in the CPU for post processing
YTrain = gather(extractdata(YTrain));

% Extract the predictions from the detector.
[gatheredPredictions, YPredCell, state] = forward(detector, XTrain);

% Generate target for predictions from the ground truth data.
[boxTarget, objectnessTarget, classTarget, objectMaskTarget, boxErrorScale] = generateTargets(gat
    YTrain, inputImageSize, detector.AnchorBoxes, penaltyThreshold);

% 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, detector.Learnables);
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),boxMask
lossY = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,2),boxDeltaTarget(:,2),boxMask
lossW = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,3),boxDeltaTarget(:,3),boxMask
lossH = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,4),boxDeltaTarget(:,4),boxMask
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'),objectnes
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'),classPred
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);
    bboxes = round(bboxes);
    labels = labels(indices);

    % Return original data only when all boxes are removed by warping.
    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(classNames')}, 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, 'Uni
len = max( cellfun(@(x)size(x,1), combinedResponses ) );
paddedBBoxes = cellfun( @(v) padarray(v,[len-size(v,1),0],0,'post'), combinedResponses, 'UniformO
YTrain = cat(4, paddedBBoxes{:,1});
end
```

Learning Rate Schedule Function

```
function currentLR = piecewiseLearningRateWithWarmup(iteration, epoch, learningRate, warmupPerioc
% 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
    % 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
```

Utility Functions

```
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 detector = downloadPretrainedYOLOv3Detector()
% Download a pretrained yolov3 detector.
if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.mat', 'file')
    if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.zip', 'file')
        disp('Downloading pretrained detector...');
        pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/yolov3SqueezeNetVehi
        websave('yolov3SqueezeNetVehicleExample_21aSPKG.zip', pretrainedURL);
    end
    unzip('yolov3SqueezeNetVehicleExample_21aSPKG.zip');
end
pretrained = load("yolov3SqueezeNetVehicleExample_21aSPKG.mat");
detector = pretrained.detector;
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. “YOLOv3: An Incremental Improvement.” Preprint, submitted April 8, 2018. https://arxiv.org/abs/1804.02767.

See Also

`detect` | `preprocess` | `forward` | `yolov3ObjectDetector` | `analyzeNetwork` | `evaluateDetectionPrecision` | `evaluateDetectionMissRate`

Related Examples

- “Getting Started with YOLO v3” on page 17-41
- “Object Detection Using YOLO v2 Deep Learning” on page 3-338

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 “Getting Started with YOLO v2” on page 17-37.

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/yolov2ResNet50VehicleExar
    websave('yolov2ResNet50VehicleExample_19b.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small vehicle dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. 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=4×2 table
              imageFilename                vehicle
    ___________________________________    ___________

    {'vehicleImages/image_00001.jpg'}    {1×4 double}
    {'vehicleImages/image_00002.jpg'}    {1×4 double}
    {'vehicleImages/image_00003.jpg'}    {1×4 double}
    {'vehicleImages/image_00004.jpg'}    {1×4 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.

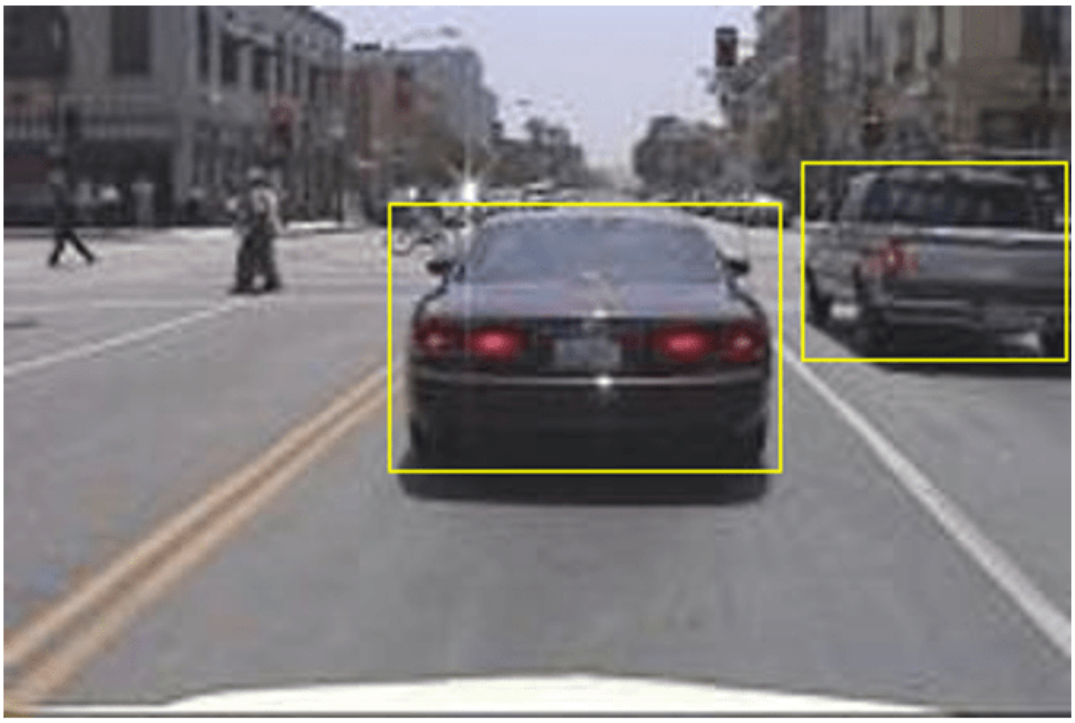
```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

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

   162   136
    85    80
   149   123
    43    32
    65    63
   117   105
    33    27


meanIoU = 0.8472
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” on page 3-319 (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” on page 17-32.

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 17-38.

Data Augmentation

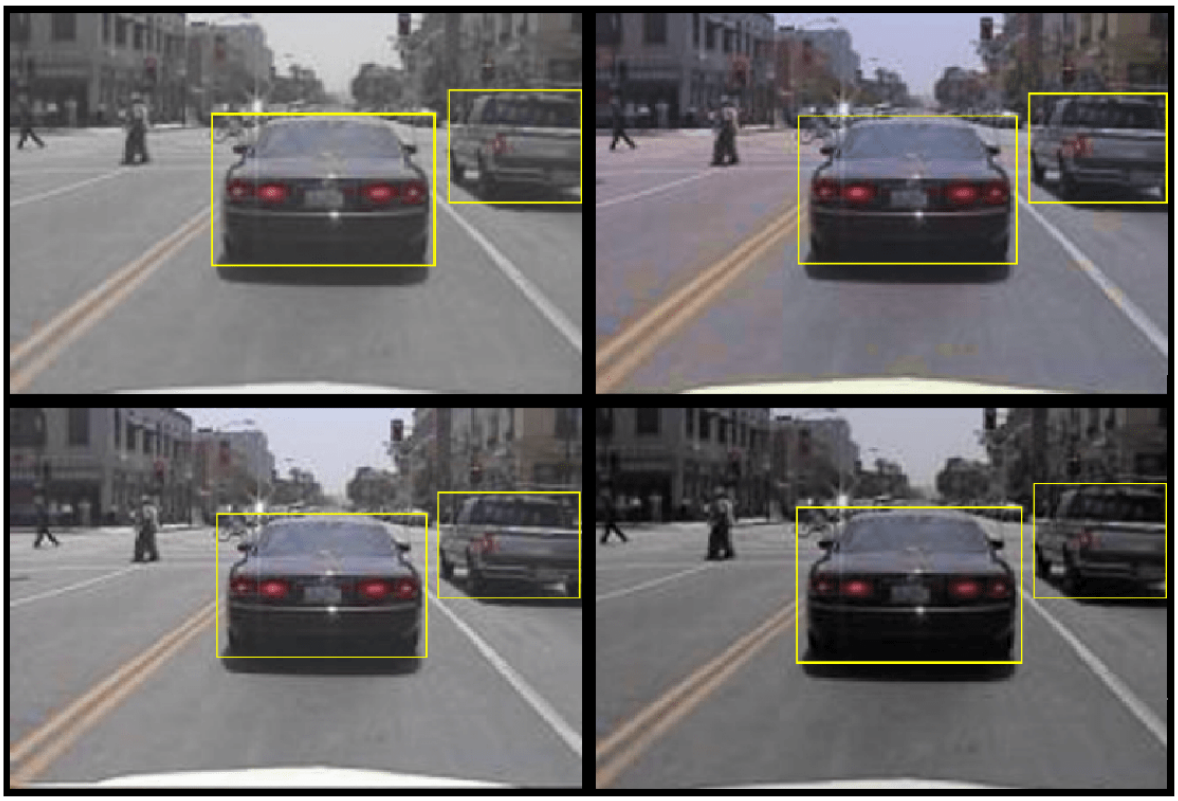
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation you can add more variety to the training data without actually having to increase the number of labeled training samples.

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
```

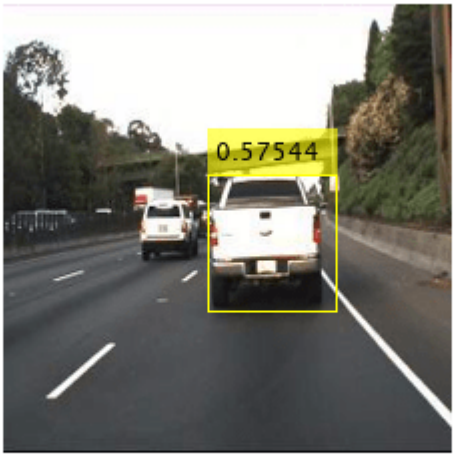
This example was verified on an NVIDIA™ Titan X GPU with 12 GB of memory. If your GPU has less memory, you may run out of memory. If this happens, lower the `'MiniBatchSize'` using the `trainingOptions` function. Training this network took approximately 7 minutes using this setup. Training time varies 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.

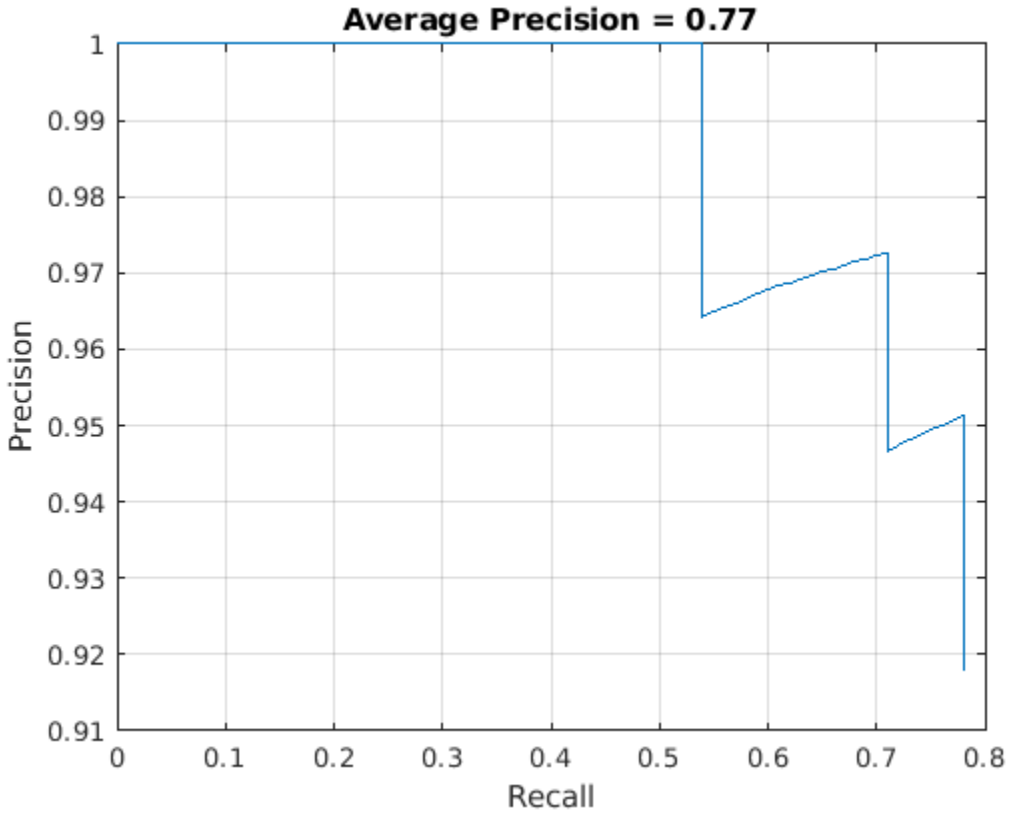
```
detectionResults = detect(detector, preprocessedTestData);
```

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);

% Sanitize box data, if needed.
A{2} = helperSanitizeBoxes(A{2}, sz);

% 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.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2},sz);

% Resize boxes to new image size.
data{2} = bboxresize(data{2},scale);
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.

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 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 paddi
     2   'yolov2Batch1'   Batch Normalization   Batch normalization
     3   'yolov2Relu1'    ReLU                  ReLU
     4   'yolov2Conv2'    Convolution           96 3x3 convolutions with stride [1  1] and paddi
     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] a
     2   'yolov2Batch1'        Batch Normalization       Batch normalization
     3   'yolov2Relu1'         ReLU                      ReLU
     4   'yolov2Conv2'         Convolution               96 3x3 convolutions with stride [1  1] a
     5   'yolov2Batch2'        Batch Normalization       Batch normalization
     6   'yolov2Relu2'         ReLU                      ReLU
     7   'yolov2ClassConv'     Convolution               20 1x1 convolutions with stride [1  1] a
     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 is highly recommended for running this example. Use of a GPU requires the Parallel Computing Toolbox™. For information about the supported compute capabilities, see “GPU Support by Release” (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 = 1×4

          32          32           3       50000
```

CIFAR-10 has 10 image categories. List the image categories:

```
numImageCategories = 10;
categories(trainingLabels)
```

```
ans = 10×1 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  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  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  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 is highly recommended for training.

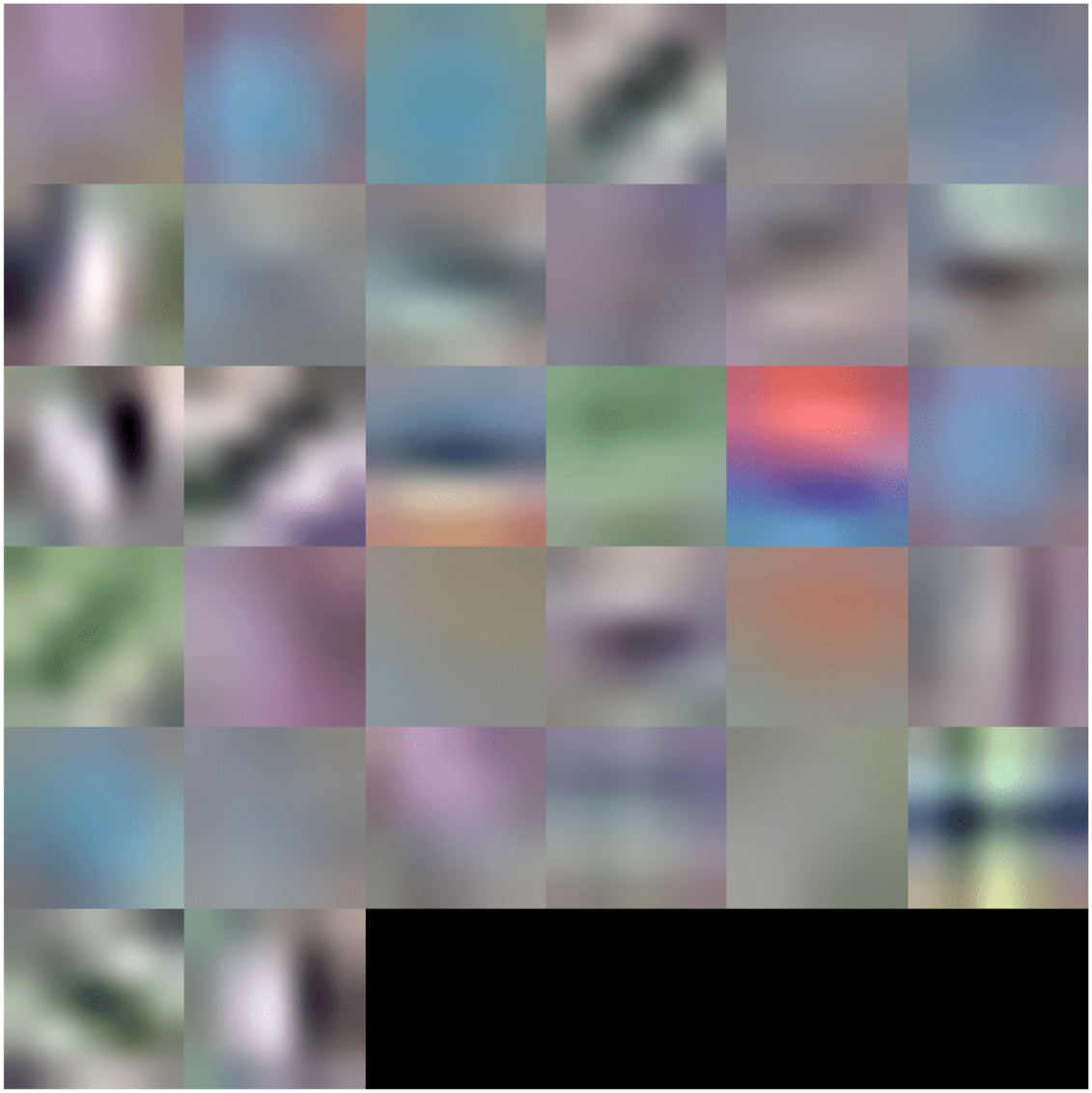
```
% A trained network is loaded from disk to save time when running the
% example. Set this flag to true to train the network.
```

```
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: 41×1 cell array of character vectors
    stopSign: 41×1 cell
    carRear: 41×1 cell
    carFront: 41×1 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 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 = 1×4

   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: [15×1 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.

See Also

`rcnnObjectDetector` | `trainingOptions` | `trainNetwork` | `trainRCNNObjectDetector` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `classify` | `detect` | `activations`

More About

- “Object Detection Using Faster R-CNN Deep Learning” on page 3-364
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

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/

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”.

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('fasterRCNNResNet50EndToEndVehicleExample.mat','file')
    disp('Downloading pretrained detector (118 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/fasterRCNNResNet50EndToEn
    websave('fasterRCNNResNet50EndToEndVehicleExample.mat',pretrainedURL);
end
```

Load Data Set

This example uses a small labeled dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. 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.

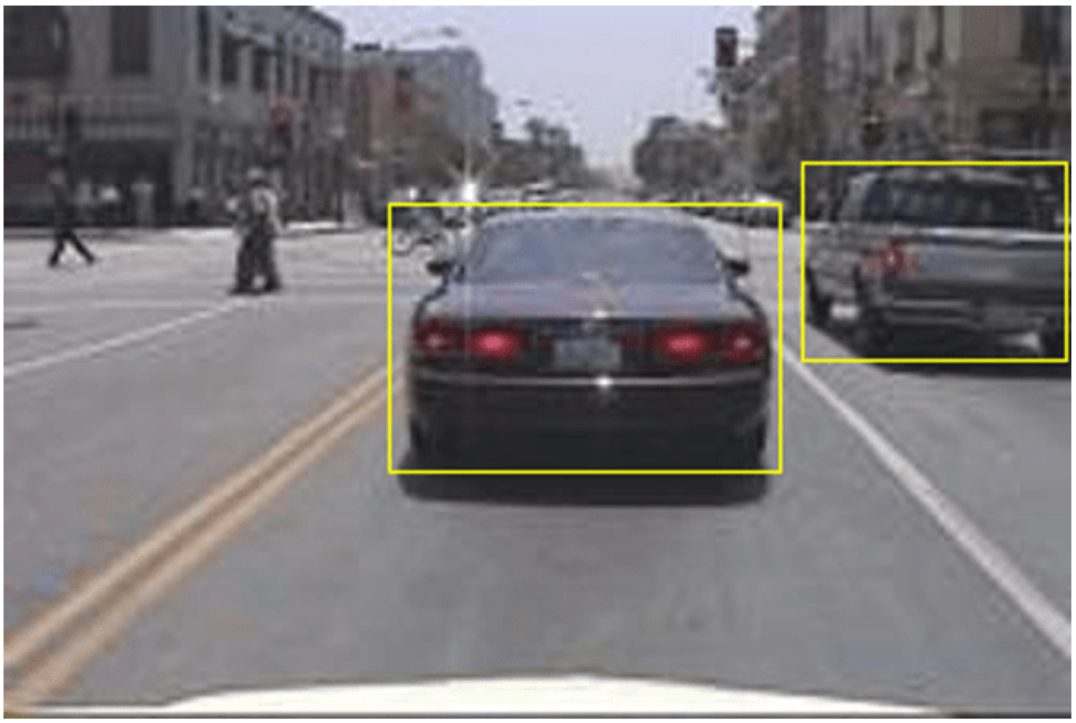
```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

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 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

    29    17
    46    39
   136   116
```

For more information on choosing anchor boxes, see“Estimate Anchor Boxes From Training Data” on page 3-319 (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” on page 17-32.

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 17-49.

Data Augmentation

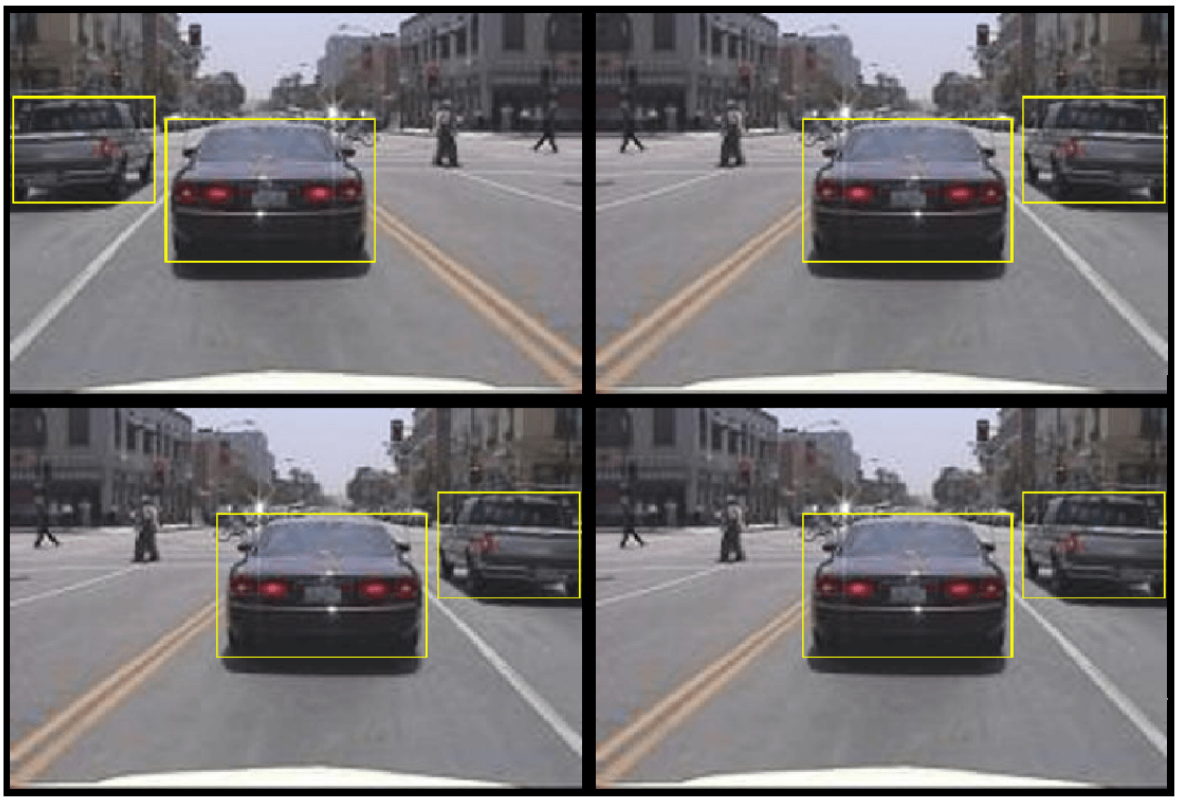
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

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 `doTraining` is true. Otherwise, load the pretrained network.

```
if doTraining
    % 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{3});
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.

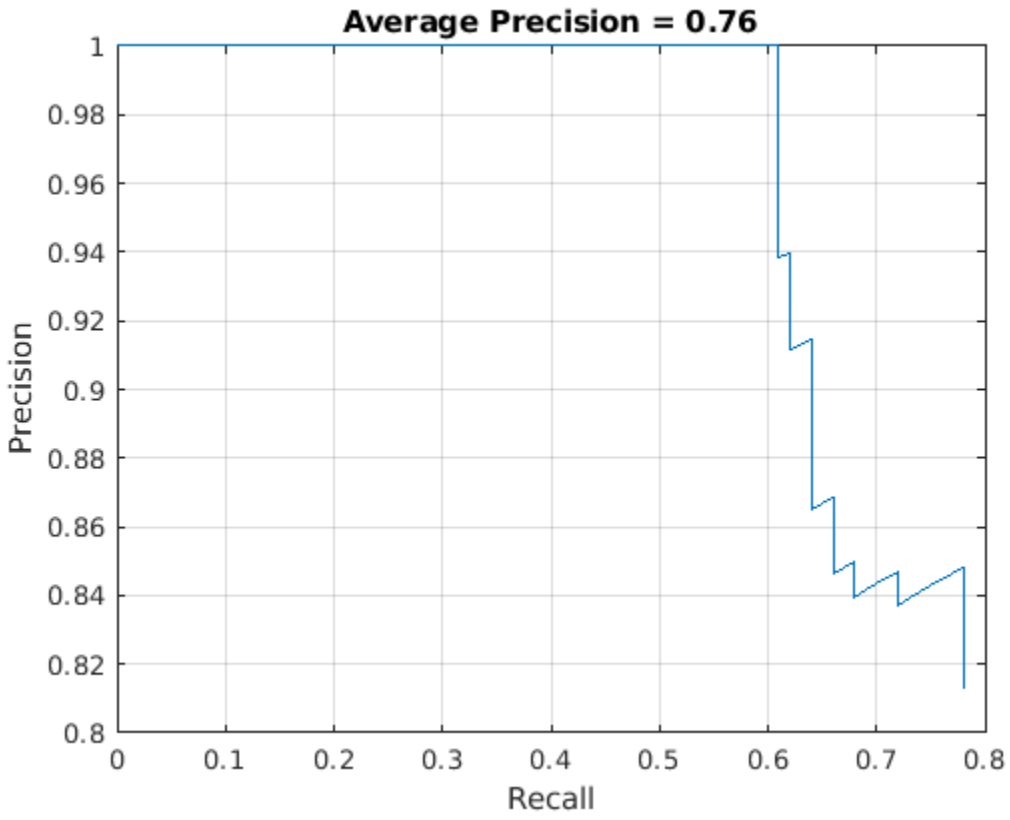
```
detectionResults = detect(detector,testData,'MinibatchSize',4);
```

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);
sz = size(data{1});
rout = affineOutputView(sz,tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Warp boxes.
data{2} = bboxwarp(data{2},tform,rout);
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Resize boxes.
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

`rcnnObjectDetector` | `trainingOptions` | `trainNetwork` | `trainRCNNObjectDetector` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `detect` | `insertObjectAnnotation` | `evaluateDetectionMissRate` | `evaluateDetectionPrecision`

More About

- “Train Object Detector Using R-CNN Deep Learning” on page 3-351
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Train Classification Network to Classify Object in 3-D Point Cloud

This example demontrates 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 appraoches.

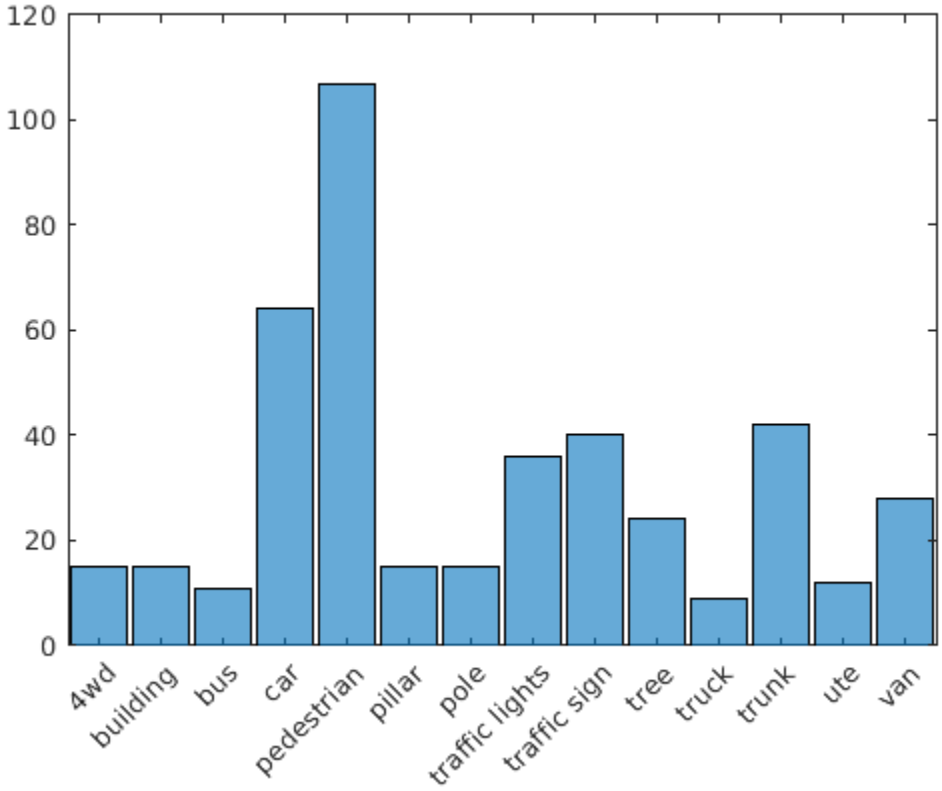
Import and Analyze Data

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 = loadSydneyUrbanObjectsData(dataPath,[1 2 3]);
dsVal = loadSydneyUrbanObjectsData(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`.

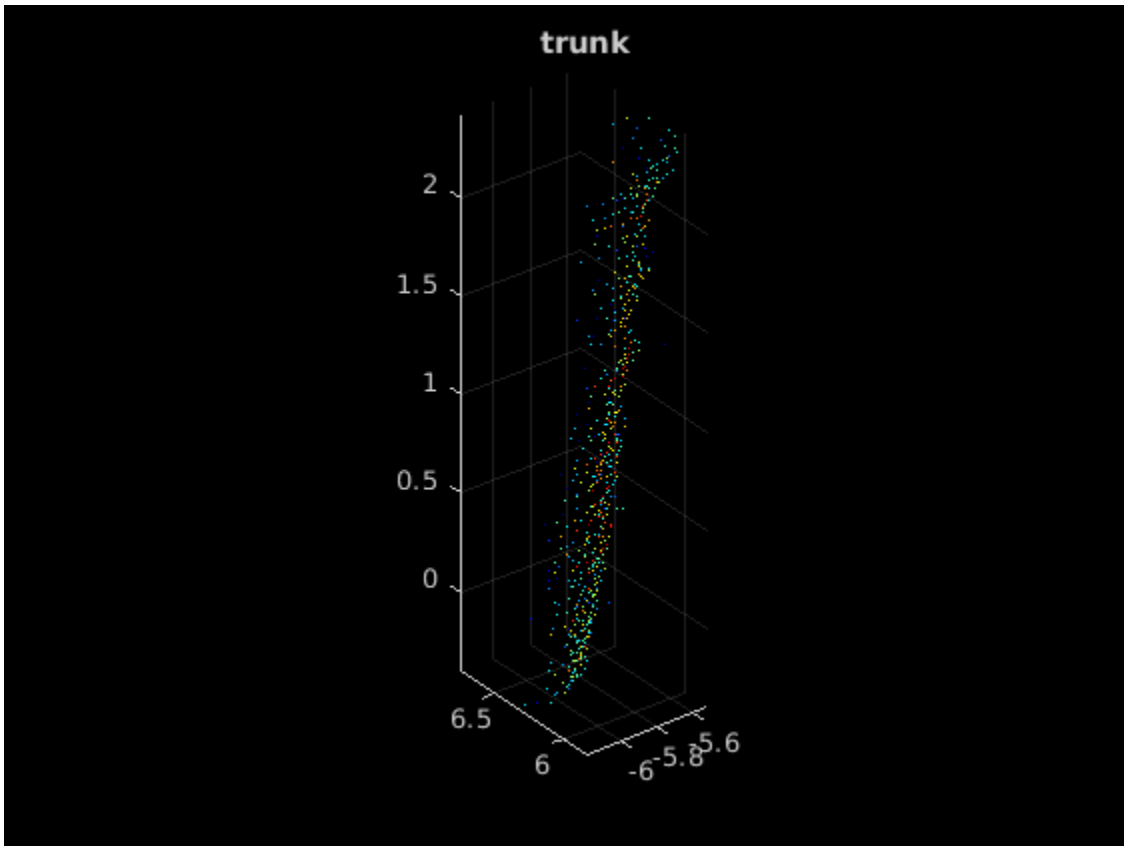
Data augmentation pipeline

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});
```

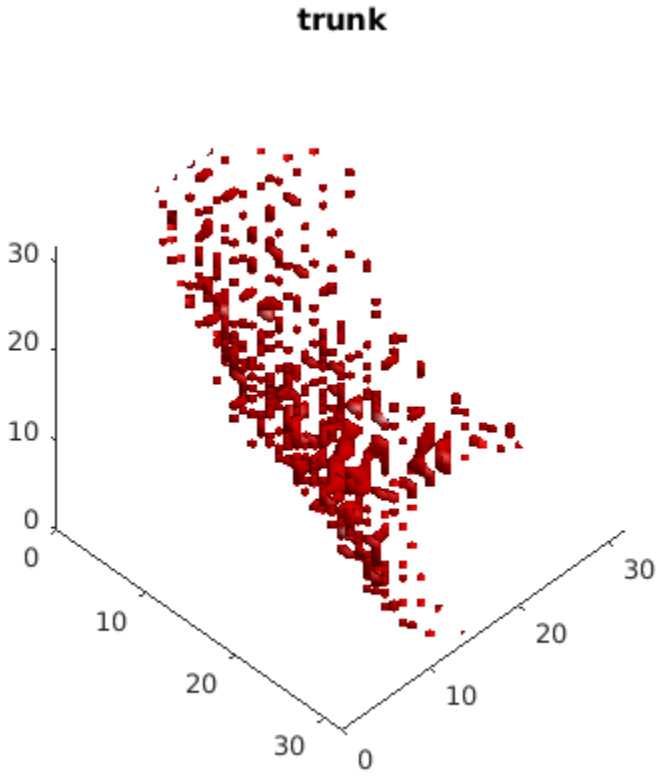


We next add a simple voxelization transform to each input point cloud as discussed in the previous example, to transform our input point cloud into a pseudo-image that can be used with a convolutional neural network. Use a simple occupancy grid.

```
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 voxelixation is working correctly.

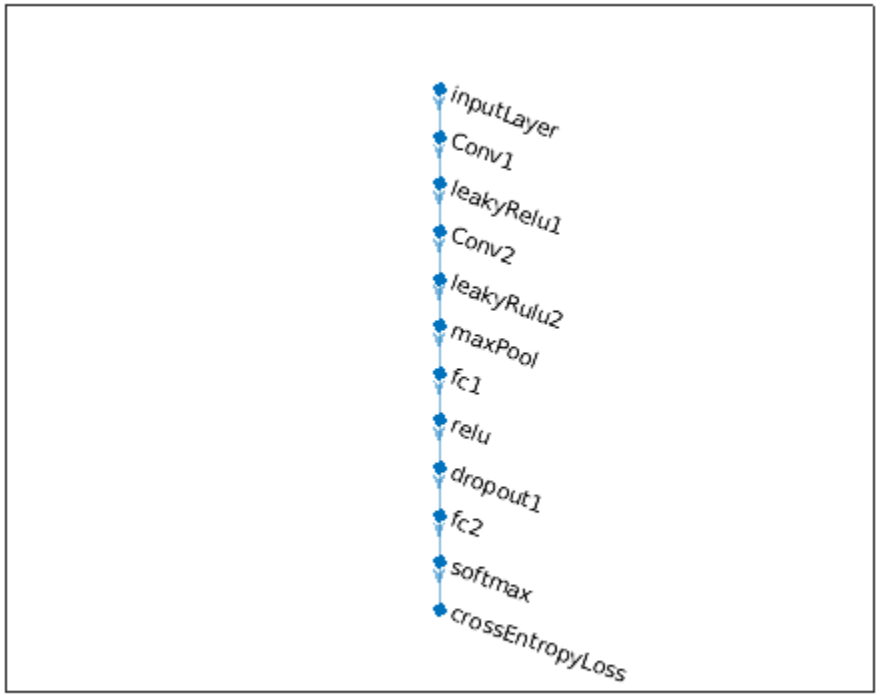
```
data = preview(dsTrain);
figure
p = patch(isosurface(data{1},0.5));
p.FaceColor = 'red';
p.EdgeColor = 'none';
daspect([1 1 1])
view(45,45)
camlight;
lighting phong
title(data{2});
```



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 useage 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 CPU.
|=========================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================
```

```
|       1 |           1 |       00:00:12 |        0.00% |        3.23% |       2.6579 |       2.6
|       4 |          50 |       00:01:53 |       31.25% |       29.03% |       2.1520 |       2.3
|       8 |         100 |       00:03:33 |       28.12% |       36.77% |       2.2633 |       2.1
|      12 |         150 |       00:05:11 |       43.75% |       46.45% |       2.0506 |       1.9
|      16 |         200 |       00:06:49 |       37.50% |       52.26% |       1.8627 |       1.6
|      20 |         250 |       00:08:35 |       50.00% |       59.35% |       1.8573 |       1.4
|      24 |         300 |       00:10:14 |       34.38% |       58.06% |       1.8636 |       1.4
|      27 |         350 |       00:11:51 |       62.50% |       61.94% |       1.4174 |       1.3
|      31 |         400 |       00:13:31 |       65.62% |       64.52% |       1.1966 |       1.2
|      35 |         450 |       00:15:09 |       56.25% |       61.94% |       1.3562 |       1.2
|      39 |         500 |       00:16:49 |       62.50% |       66.45% |       1.2819 |       1.1
|      43 |         550 |       00:18:27 |       56.25% |       65.16% |       1.4563 |       1.1
|      47 |         600 |       00:20:05 |       56.25% |       66.45% |       1.3096 |       1.1
|      50 |         650 |       00:21:40 |       56.25% |       65.16% |       1.0104 |       1.1
|      54 |         700 |       00:23:21 |       75.00% |       70.32% |       0.9403 |       1.0
|      58 |         750 |       00:25:00 |       65.62% |       71.61% |       1.0909 |       1.1
|      60 |         780 |       00:25:59 |       65.62% |       72.26% |       0.9628 |       1.0
|=========================================================================================================
```

Evaluate network

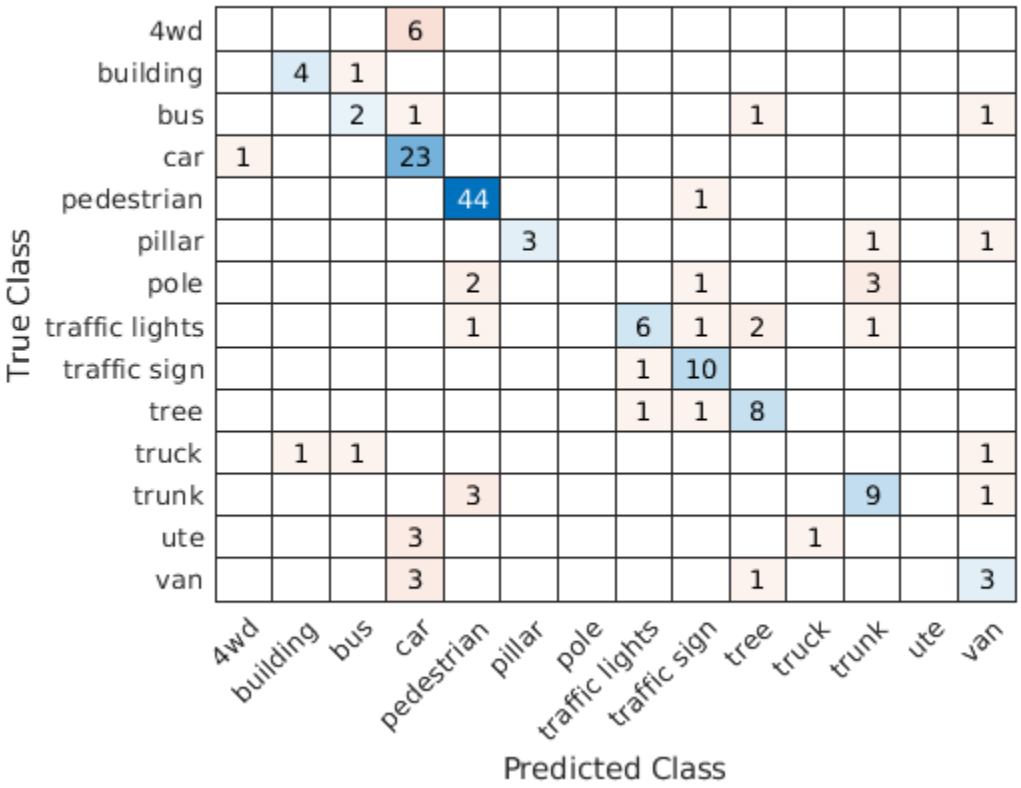
Following the structure of [1 on page 3-0], this example only forms a training and validation set from Sydney Urban Objects. Evaluate the performance of the trained network using the validation, since it was not used to train the network.

```
valLabelSet = transform(dsVal,@(data) data{2});
valLabels = readall(valLabelSet);
outputLabels = classify(voxnet,dsVal);
accuracy = nnz(outputLabels == valLabels) / numel(outputLabels);
disp(accuracy)
```

```
    0.7226
```

View the confusion matrix to study the accuracy across the various label categories

```
confusionchart(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 demonstate 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 achieve 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(url+name,dataLoc);
end

datasetPath = dataLoc.append('sydney-urban-objects-dataset');

end

function ds = loadSydneyUrbanObjectsData(datapath,folds)
% loadSydneyUrbanObjectsData Datastore with point clouds and
% associated categorical labels for Sydney Urban Objects dataset.
%
% ds = loadSydneyUrbanObjectsData(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 = loadSydneyUrbanObjectsData(___,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 = pcbin(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 localized 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://ssd.mathworks.com/supportfiles/vision/data/human-pose-estimatio
downloadTrainedOpenPoseNet(trainedOpenPoseNet_url,dataDir)
```

```
Pretrained OpenPose network already exists.
```

```
unzip(fullfile(dataDir,'human-pose-estimation.zip'),dataDir);
```

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);
```

Remove the unused output layers.

```
layers = removeLayers(layers,layers.OutputNames);
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 `dlarray`.

```
netInput = dlarray(netInput,"SSC");
```

Predict the heatmaps and part affinity fields (PAFs) , which are output from the 2-D output convolutional layers.

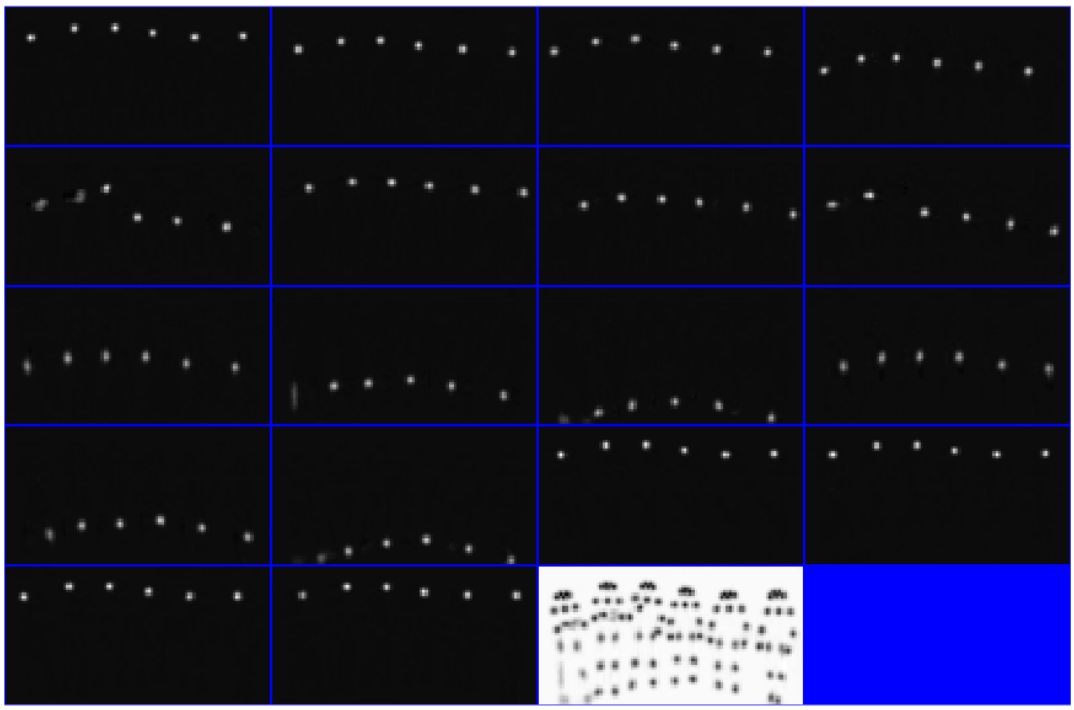
```
[heatmaps,pafs] = predict(net,netInput);
```

Get the numeric heatmap data stored in the `dlarray`. 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);
```

Get the numeric PAF data stored in the `dlarray`. 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);
```

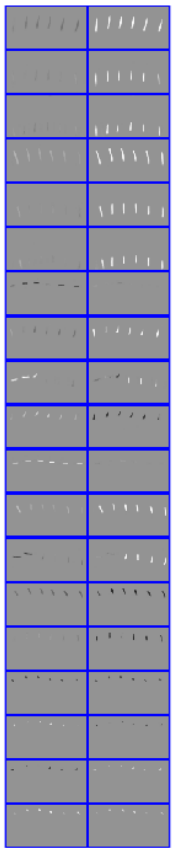
Display the PAFs in a montage, rescaling the data to the range [0, 1] expected of images of data type `single`. The two columns show the x- and y-components of the vector field, respectively. The body part pairings are in the order determined by the `params.PAF_INDEX` value.

- 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.
- 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.

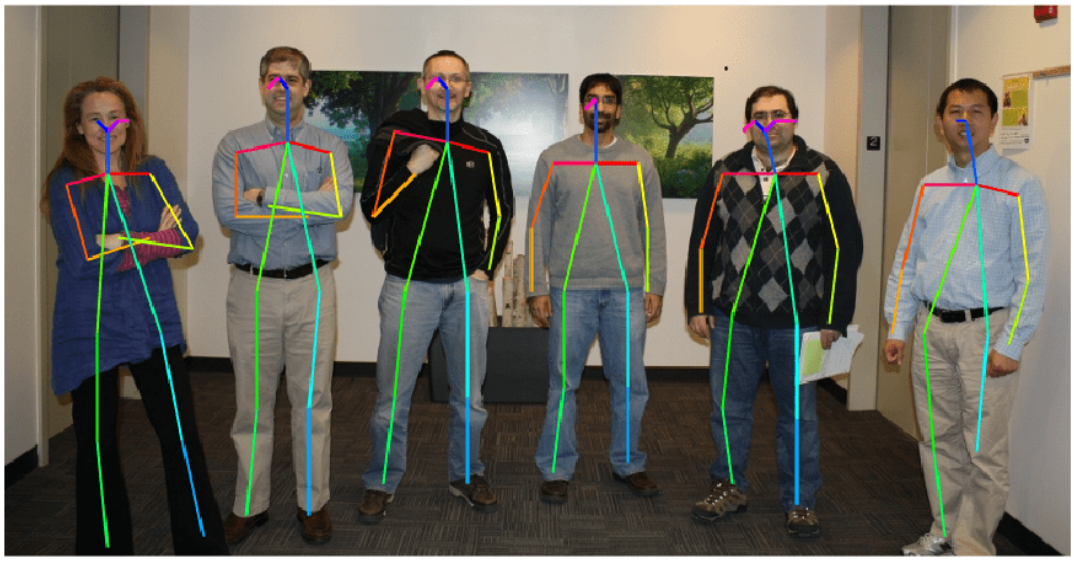
2. For each type of body part pairing, generate all possible pairs between detected body parts. For instance, generate all possible pairs between the six necks and the six left shoulders. The result is a bipartite graph.
3. Score the pairs by computing the line integral of the straight line connecting 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. https://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. https://arxiv.org/abs/1811.12004.

See Also

`importONNXLayers` | `dlnetwork` | `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.zi
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.z

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 colormap 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.

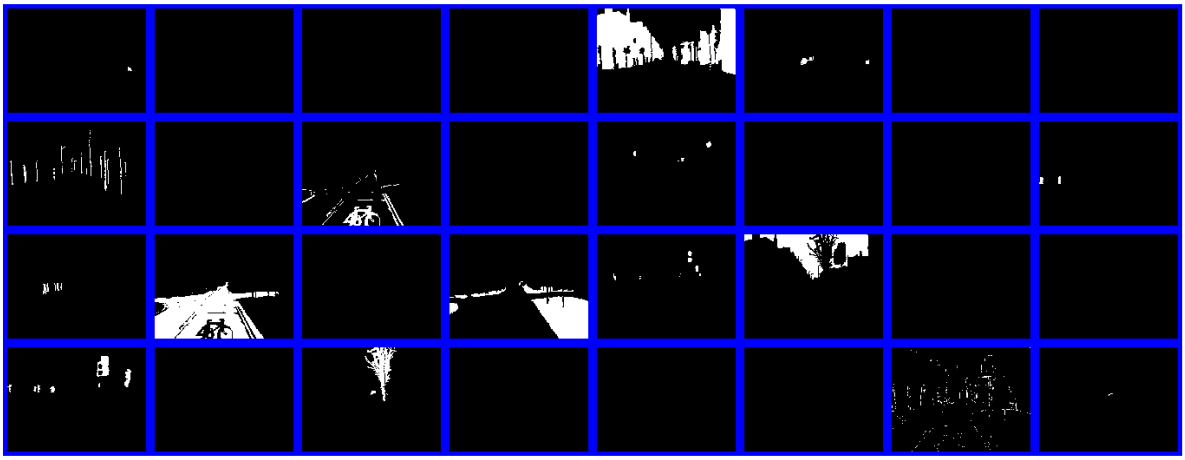
The `preprocessCamVidForPix2PixHD` function performs these operations:

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 pix2pixHD 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.

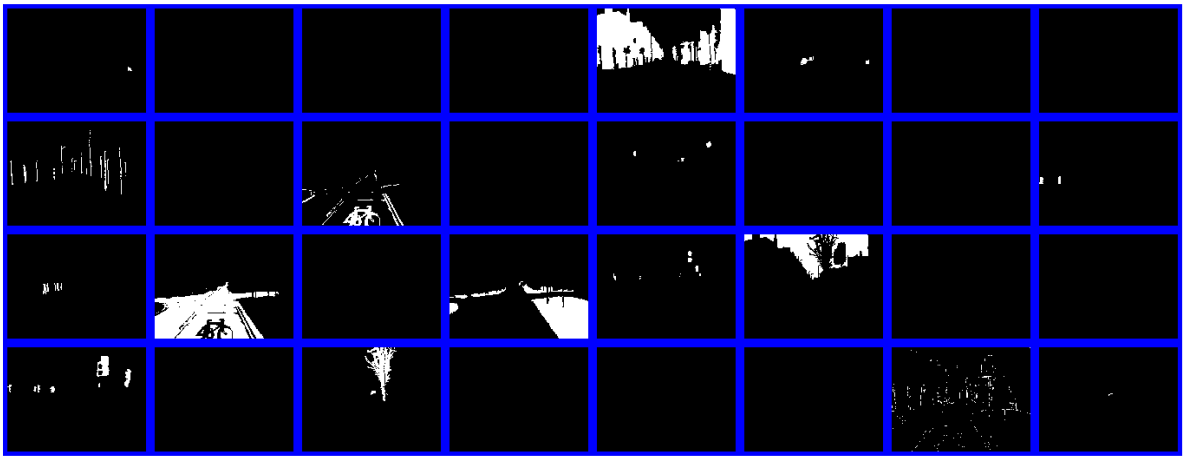
```
generatorInputSize = [imageSize numClasses];
```

Create the pix2pixHD generator network using the `pix2pixHDGlobalGenerator` function.

```
dlnetGenerator = pix2pixHDGlobalGenerator(generatorInputSize);
```

Display the network architecture.

```
analyzeNetwork(dlnetGenerator)
```



Note that this example shows the use of pix2pixHD global generator for generating images of size 576-by-768 pixels. To create local enhancer networks that generate images at higher resolution such as 1152-by-1536 pixels or even higher, you can use the `addPix2PixHDLocalEnhancer` function. The local enhancer networks help generate fine level details at very high resolutions.

Create Discriminator Network

Define the patch GAN discriminator networks that classifies an input image as either real (1) or fake (0). This example uses two discriminator networks at different input scales, also known as multiscale discriminators. The first scale is the same size as the image size, and the second scale is half the size of image size.

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;
```

Specify the input size of the first discriminator. Create the patch GAN discriminator with instance normalization using the `patchGANDiscriminator` function.

```
discriminatorInputSizeScale1 = [imageSize numChannelsDiscriminator];
dlnetDiscriminatorScale1 = patchGANDiscriminator(discriminatorInputSizeScale1,"NormalizationLaye
```

Specify the input size of the second discriminator as half the image size, then create the second patch GAN discriminator.

```
discriminatorInputSizeScale2 = [floor(imageSize)./2 numChannelsDiscriminator];
dlnetDiscriminatorScale2 = patchGANDiscriminator(discriminatorInputSizeScale2,"NormalizationLaye
```

Visualize the networks.

```
analyzeNetwork(dlnetDiscriminatorScale1);
analyzeNetwork(dlnetDiscriminatorScale2);
```

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. $\widehat{Y}_{generated}$ 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.

  $$lossAdversarialGenerator = (1 - \widehat{Y}_{generated})^2$$

- 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 $\widehat{Y}_{generated}$ 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

  $$lossFeatureMatching = \sum_{i=1}^{T} ||Y_{real} - \widehat{Y}_{generated}||_1$$

- 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 $\widehat{Y}_{VggGenerated}$ 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 .

  $$lossVgg = \sum_{i=1}^{T} ||Y_{VggReal} - \widehat{Y}_{VggGenerated}||_1$$

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 discriminator 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 = [];
trailingAvgDiscriminatorScale1 = [];
trailingAvgSqDiscriminatorScale1 = [];
```

```
trailingAvgDiscriminatorScale2 = [];
trailingAvgSqDiscriminatorScale2 = [];
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 generator 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.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (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,gradParamsDScale1,gradParamsDScale2,lossGGAN,lossGFM,lossGVGG,lossD] = d
                    @modelGradients,dlInputSegMap,dlRealImage,dlnetGenerator,dlnetDiscriminatorScale1

                % Update the generator parameters
                [dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = adamupdate( ...
                    dlnetGenerator,gradParamsG, ...
                    trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
                    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

                % Update the discriminator scale1 parameters
                [dlnetDiscriminatorScale1,trailingAvgDiscriminatorScale1,trailingAvgSqDiscriminatorSc
                    dlnetDiscriminatorScale1,gradParamsDScale1, ...
                    trailingAvgDiscriminatorScale1,trailingAvgSqDiscriminatorScale1,iteration, ...
                    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

                % Update the discriminator scale2 parameters
                [dlnetDiscriminatorScale2,trailingAvgDiscriminatorScale2,trailingAvgSqDiscriminatorSc
                    dlnetDiscriminatorScale2,gradParamsDScale2, ...
                    trailingAvgDiscriminatorScale2,trailingAvgSqDiscriminatorScale2,iteration, ...
                    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

                % Plot and display various losses
                lossPlotter = updateTrainingProgressPlotter(lossPlotter,iteration, ...
                    epoch,numEpochs,lossD,lossGGAN,lossGFM,lossGVGG);
            end
        end
        save('trainedPix2PixHDNet.mat','dlnetGenerator');

    else
        trainedPix2PixHDNet_url = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedPix2PixH
        netDir = fullfile(tempdir,'CamVid');
        downloadTrainedPix2PixHDNet(trainedPix2PixHDNet_url,netDir);
        load(fullfile(netDir,'trainedPix2PixHDv2.mat'));
    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.

```
segMapOneHot = onehotencode(segMap,3,'single');
```

Create `dlarray` objects 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(segMapOneHot,'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 Image
```

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 using the `onehotencode` (Deep Learning Toolbox) function.
- 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 Image
```

Evaluate Generated Images from Custom Pixel Label Images

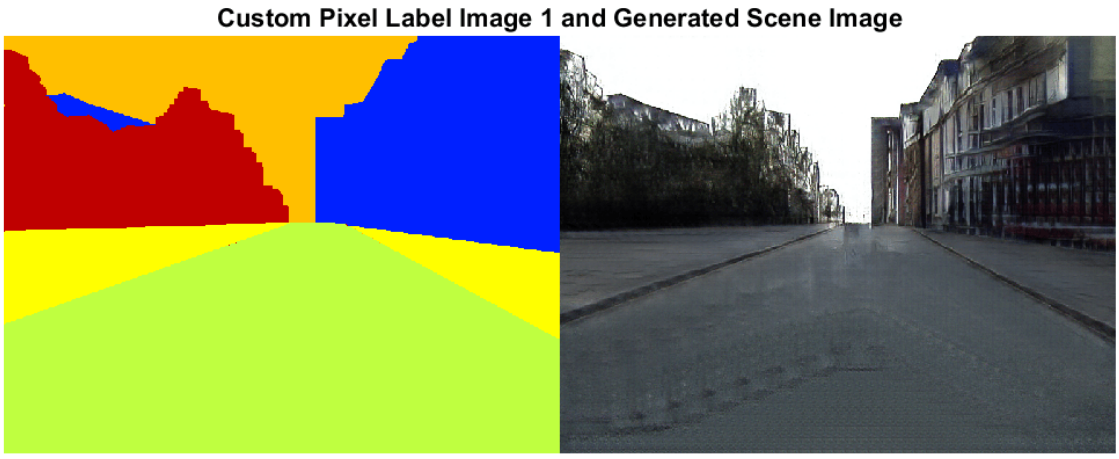
To evaluate how well the network generalizes to pixel label images outside the CamVid data set, generate scene images from custom pixel label images. This example uses pixel label images that were created using the Image Labeler app. The pixel label images are attached to the example as supporting files. No ground truth images are available.

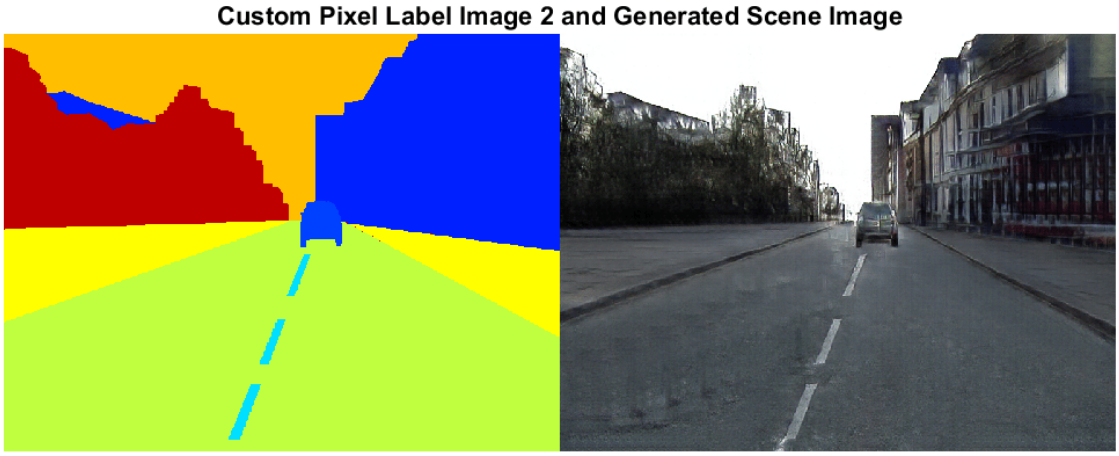
Create a pixel label datastore that reads and processes the pixel label images in the current example directory.

```
cpxds = 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,gradParamsDScale1,gradParamsDScale2,lossGGAN,lossGFM,lossGVGG,lossD] = mode

    % 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
% for first scale.
[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
% for second scale.
[DLossScale2,GLossScale2,realPredScale2D,fakePredScale2G] = pix2pixHDAdverserialLoss(inpDisc

% Compute the feature matching loss for first scale.
FMLossScale1 = pix2pixHDFeatureMatchingLoss(realPredScale1D,fakePredScale1G);
FMLossScale1 = FMLossScale1 * lambdaFeatureMatching;

% Compute the feature matching loss for second scale.
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,'RetainData',true);

% Compute the combined discriminator loss
lossDCombined = (DLossScale1 + DLossScale2)/2 * lambdaDiscriminator;

% Compute gradients for the discriminator scale1
gradParamsDScale1 = dlgradient(lossDCombined,discriminatorScale1.Learnables,'RetainData',true

% Compute gradients for the discriminator scale2
gradParamsDScale2 = dlgradient(lossDCombined,discriminatorScale2.Learnables);

% Log the values for displaying later
lossD = gather(extractdata(lossDCombined));
lossGGAN = gather(extractdata(GLossScale1 + GLossScale2));
lossGFM  = gather(extractdata(FMLossScale1 + FMLossScale2));
```

```
    lossGVGG = gather(extractdata(VGGLoss));
end
```

Adversarial Loss Function

The helper function `pix2pixHDAdverserialLoss` 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,genPredFtrsD] = pix2pixHDAdverserialLoss(inpReal,inpGenerated

    % Discriminator layer names containing feature maps
    featureNames = {'act_top','act_mid_1','act_mid_2','act_tail','conv2d_final'};

    % 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
    GLoss = (1 - genPredD).^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

`vgg19` | `imageDatastore` | `pixelLabelDatastore` | `trainingOptions` | `trainNetwork` | `transform` | `combine`

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)

Train 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 17-62.

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 9-14.

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.

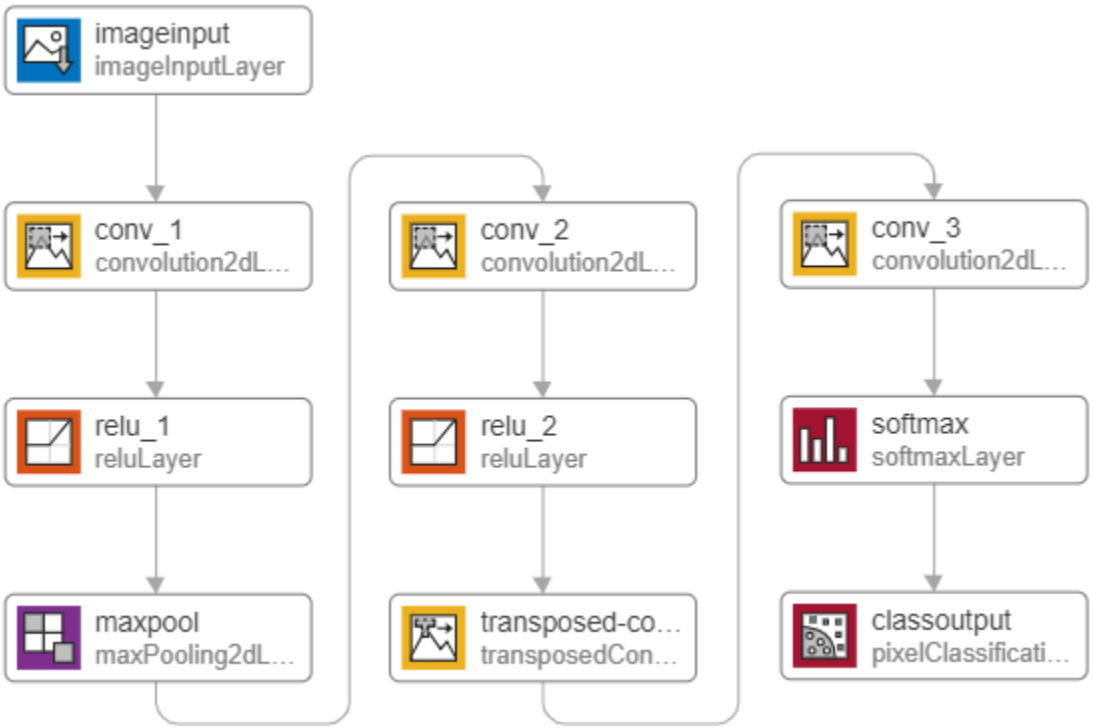
```
deepNetworkDesigner
```

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])
```

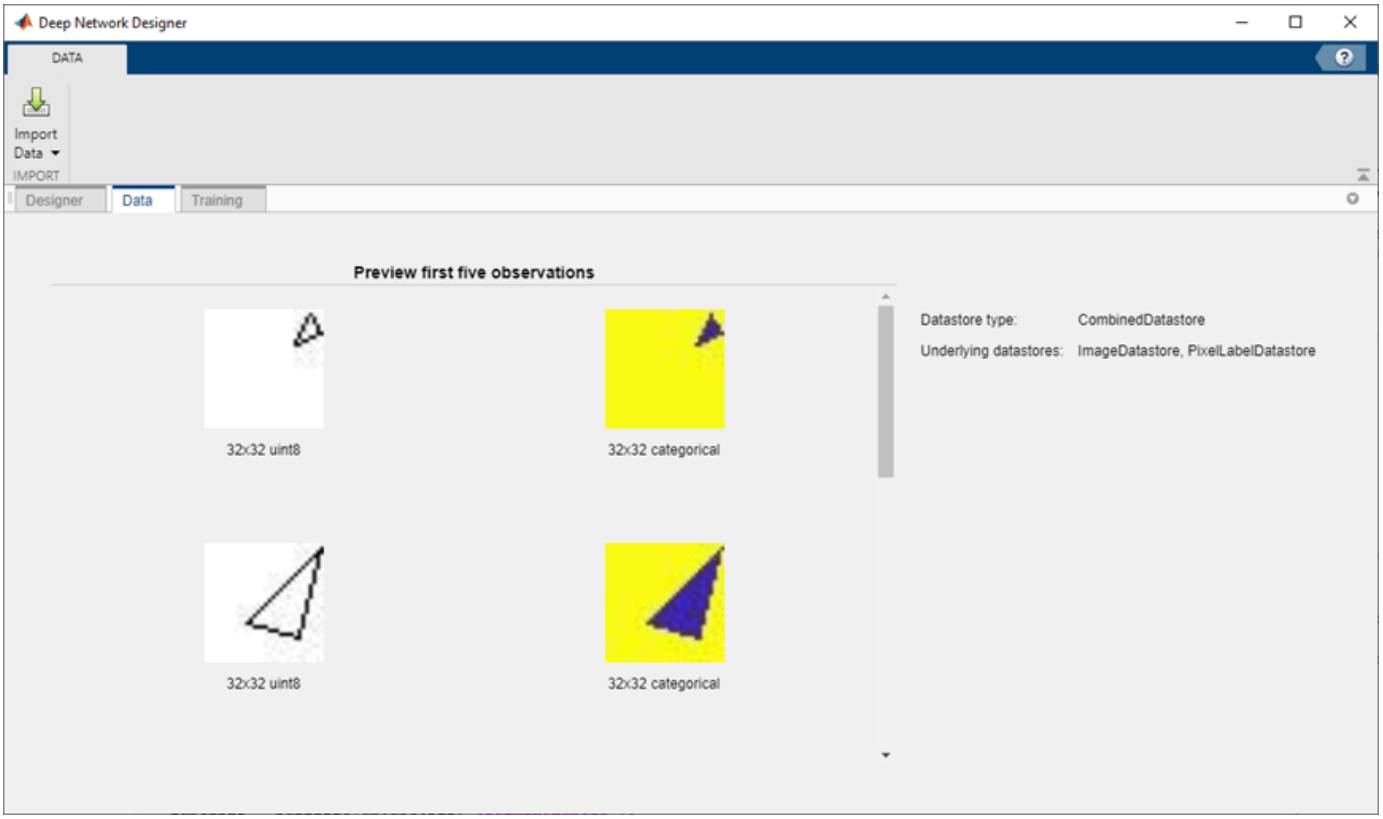
```
convolution2dLayer([3,3],64,'Padding',[1,1,1,1])
reluLayer
maxPooling2dLayer([2,2],'Stride',[2,2])
convolution2dLayer([3,3],64,'Padding',[1,1,1,1])
reluLayer
transposedConv2dLayer([4,4],64,'Stride',[2,2],'Cropping',[1,1,1,1])
convolution2dLayer([1,1],2)
softmaxLayer
pixelClassificationLayer
];
```

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**.

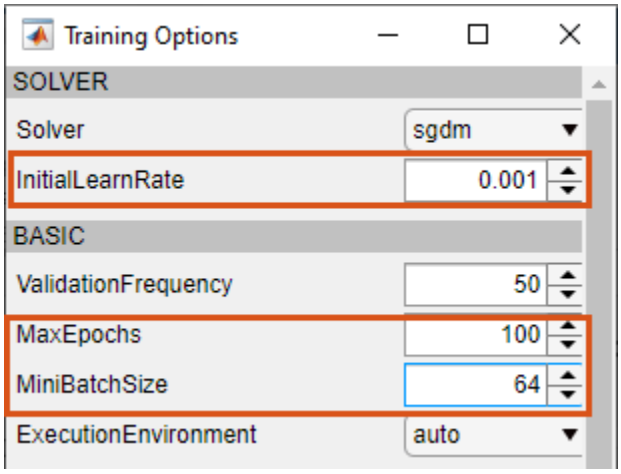
Deep Network Designer displays a preview of the imported semantic segmentation data. The preview displays the training images and the ground truth pixel labels. The network requires input images (left) and returns a classification for each pixel as either triangle or background (right).



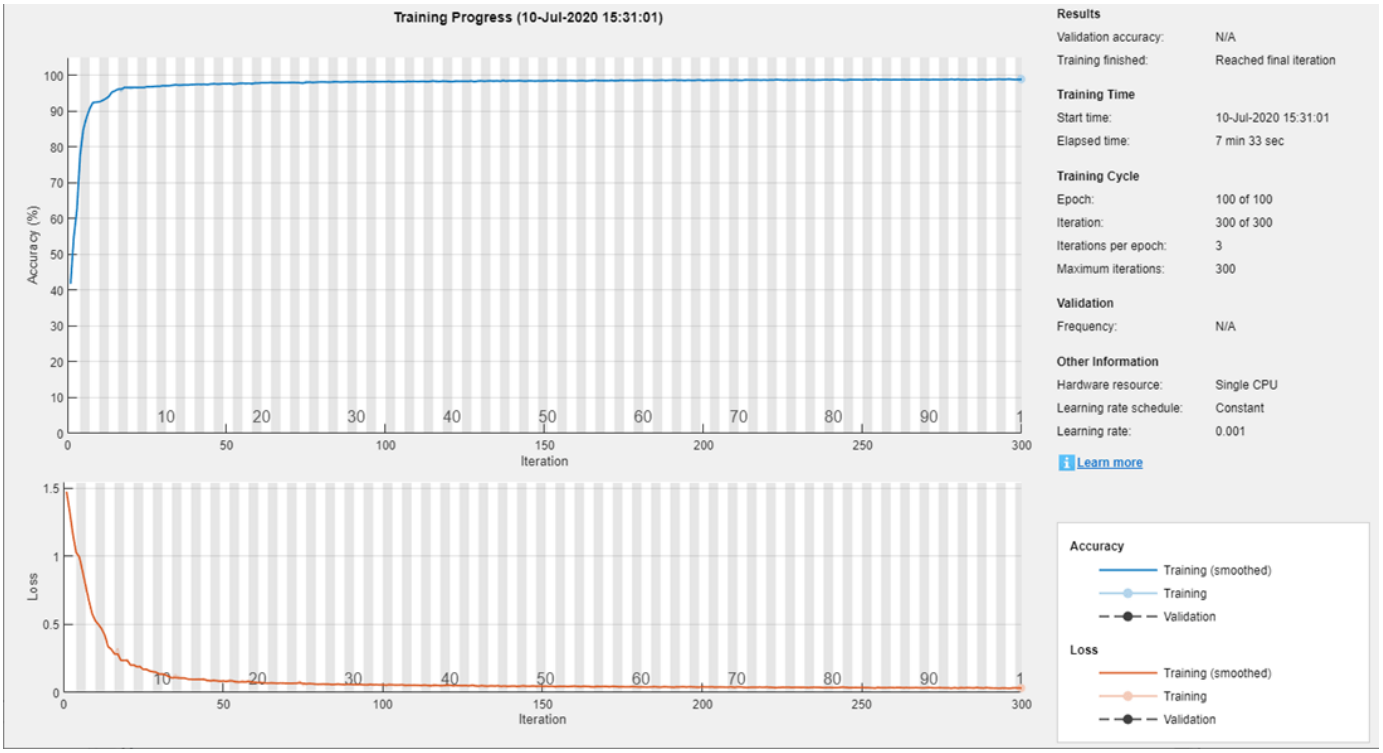
Train Network

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

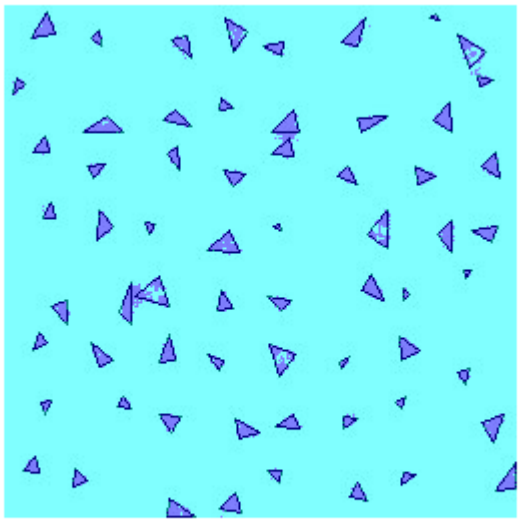
Make predictions using test data and the trained network.

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
```

```
ans=3×3 table
        Name           Type         Group
    _____________    _________    ________

    {'stopSign'}    Rectangle    {'None'}
    {'carRear' }    Rectangle    {'None'}
    {'carFront'}    Rectangle    {'None'}
```

Select the stop sign data for training.

```
stopSignGroundTruth = selectLabelsByName(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 21.6979 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);
```

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  |  Mini-batch  |  Mini-batch  |  Mini-batch  |  Base Lea
|         |             |   (hh:mm:ss)   |     Loss     |   Accuracy   |     RMSE     |      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
```

Perform Instance Segmentation Using Mask R-CNN

This example shows how to segment individual instances of people and cars using a multiclass Mask region-based convolutional neural network (R-CNN).

Instance segmentation is a computer vision technique in which you detect and localize objects while simultaneously generating a segmentation map for each of the detected instances.

This example first shows how to perform instance segmentation using a pretrained Mask R-CNN that detects two classes. Then, you can optionally download a data set and train a multiclass Mask R-CNN using transfer learning.

Perform Instance Segmentation Using Pretrained Mask R-CNN

Specify `dataFolder` as the desired location of the pretrained network and data.

```
dataFolder = fullfile(tempdir,"coco");
```

Download the pretrained Mask R-CNN. The network is stored as a `maskrcnn` object.

```
trainedMaskRCNN_url = "https://www.mathworks.com/supportfiles/vision/data/maskrcnn_object_person_
helper.downloadTrainedMaskRCNN(trainedMaskRCNN_url,dataFolder);
pretrained = load(fullfile(dataFolder,"maskrcnn_object_person_car_v2.mat"));
net = pretrained.net;
```

Read a test image that contains objects of the target classes.

```
imTest = imread("visionteam.jpg");
```

Segment the objects and their masks using the `segmentObjects` function. The `segmentObjects` function performs these preprocessing steps on the input image before performing prediction:

1. Zero center the images using the COCO data set mean.
2. Resize the image to the input size of the network, while maintaining the aspect ratio (letter boxing).

```
[masks,labels,scores,boxes] = segmentObjects(net,imTest,Threshold=0.98);
```

Visualize the predictions by overlaying the detected masks on the image using the `insertObjectMask` function.

```
overlayedImage = insertObjectMask(imTest,masks);
imshow(overlayedImage)
```

Show the bounding boxes and labels on the objects.

```
showShape("rectangle",gather(boxes),Label=labels,LineColor=r)
```

Download Training Data

Create directories to store the COCO training images and annotation data.

```
imageFolder = fullfile(dataFolder,"images");
captionsFolder = fullfile(dataFolder,"annotations");
if ~exist(imageFolder,"dir")
    mkdir(imageFolder)
    mkdir(captionsFolder)
end
```

The COCO 2014 training images data set [2] on page 3-0 consists of 82,783 images. The annotations data contains at least five captions corresponding to each image. Download the COCO 2014 training images and captions from https://cocodataset.org/#download by clicking the "2014 Train images" and "2014 Train/Val annotations" links, respectively. Extract the image files into the folder specified by `imageFolder`. Extract the annotation files into the folder specified by `captionsFolder`.

```
annotationFile = fullfile(captionsFolder,"instances_train2014.json");
str = fileread(annotationFile);
```

Read and Preprocess Training Data

To train a Mask R-CNN, you need this data.

- RGB images that serve as input to the network, specified as *H*-by-*W*-by-3 numeric arrays.
- Bounding boxes for objects in the RGB images, specified as *NumObjects*-by-4 matrices, with rows in the format [*x y w h*]).
- Instance labels, specified as *NumObjects*-by-1 string vectors.
- Instance masks. Each mask is the segmentation of one instance in the image. The COCO data set specifies object instances using polygon coordinates formatted as *NumObjects*-by-2 cell arrays.

Each row of the array contains the (*x*,*y*) coordinates of a polygon along the boundary of one instance in the image. However, the Mask R-CNN in this example requires binary masks specified as logical arrays of size *H*-by-*W*-by-*NumObjects*.

Initialize Training Data Parameters

```
trainClassNames = {'person', 'car'};
numClasses = length(trainClassNames);
imageSizeTrain = [800 800 3];
```

Format COCO Annotation Data as MAT Files

The COCO API for MATLAB enables you to access the annotation data. Download the COCO API for MATLAB from https://github.com/cocodataset/cocoapi by clicking the "Code" button and selecting "Download ZIP." Extract the `cocoapi-master` directory and its contents to the folder specified by `dataFolder`. If needed for your operating system, compile the gason parser by following the instructions in the `gason.m` file within the `MatlabAPI` subdirectory.

Specify the directory location for the COCO API for MATLAB and add the directory to the path.

```
cocoAPIDir = fullfile(dataFolder,"cocoapi-master","MatlabAPI");
addpath(cocoAPIDir);
```

Specify the folder in which to store the MAT files.

```
unpackAnnotationDir = fullfile(dataFolder,"annotations_unpacked","matFiles");
if ~exist(unpackAnnotationDir,'dir')
    mkdir(unpackAnnotationDir)
end
```

Extract the COCO annotations to MAT files using the `unpackAnnotations` helper function, which is attached to this example as a supporting file in the folder `helper`. Each MAT file corresponds to a single training image and contains the file name, bounding boxes, instance labels, and instance masks for each training image. The function converts object instances specified as polygon coordinates to binary masks using the `poly2mask` function.

```
helper.unpackAnnotations(trainClassNames,annotationFile,imageFolder,unpackAnnotationDir);
```

Create Datastore

The Mask R-CNN expects input data as a 1-by-4 cell array containing the RGB training image, bounding boxes, instance labels, and instance masks.

Create a file datastore with a custom read function, `cocoAnnotationMATReader`, that reads the content of the unpacked annotation MAT files, converts grayscale training images to RGB, and returns the data as a 1-by-4 cell array in the required format. The custom read function is attached to this example as a supporting file in the folder `helper`.

```
ds = fileDatastore(unpackAnnotationDir, ...
    ReadFcn=@(x)helper.cocoAnnotationMATReader(x,imageFolder));
```

Preview the data returned by the transformed datastore.

```
data = preview(ds)
```

```
data=1×4 cell array
    {428×640×3 uint8}    {16×4 double}    {16×1 categorical}    {428×640×16 logical}
```

Create Mask R-CNN Network Layers

The Mask R-CNN builds upon a Faster R-CNN with a ResNet-50 base network. To transfer learn on the pretrained Mask R-CNN network, use the `maskrcnn` object to load the pretrained network and customize the network for the new set of classes and input size. By default, the `maskrcnn` object uses the same anchor boxes as used for training with COCO data set.

```
net = maskrcnn("resnet50-coco",trainClassNames,InputSize=imageSizeTrain)
```

```
net =
  maskrcnn with properties:

      ModelName: 'maskrcnn'
     ClassNames: {'person'  'car'}
      InputSize: [800 800 3]
    AnchorBoxes: [15×2 double]
```

If you want to use custom anchor boxes specific to the training data set, you can estimate the anchor boxes using the `estimateAnchorBoxes` function. The, specify the anchor boxes using the `AnchorBoxes` name-value argument when you create the `maskrcnn` object.

Train Network

Specify the options for SGDM optimization and train the network for 15 epochs.

Specify the `ExecutionEnvironment` name-value argument as `"gpu"` to train on a GPU. It is recommended to train on a GPU with at least 12 MB of available memory. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
options = trainingOptions("sgdm", ...
    InitialLearnRate=0.001, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=1, ...
    LearnRateDropFactor=0.95, ...
    Plot="none", ...
    Momentum=0.9, ...
    MaxEpochs=10, ...
    MiniBatchSize=2, ...
    BatchNormalizationStatistics="moving", ...
    ResetInputNormalization=false, ...
    ExecutionEnvironment="gpu", ...
    VerboseFrequency=50);
```

To train the Mask R-CNN network, set the `doTraining` variable in the following code to `true`. Train the network using the `trainMaskRCNN` function. Because the training data set is similar to the data that the pretrained network is trained on, you can freeze the weights of the feature extraction backbone using the `FreezeSubNetwork` name-value argument.

```
doTraining = false;
if doTraining
    [net,info] = trainMaskRCNN(ds,net,options,FreezeSubNetwork="backbone");
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save("trainedMaskRCNN-"+modelDateTime+".mat","net");
end
```

Using the trained network, you can perform instance segmentation on test images, such as demonstrated in the section Perform Instance Segmentation Using Pretrained Mask R-CNN on page 3-0 .

References

[1] He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. “Mask R-CNN.” Preprint, submitted January 24, 2018. https://arxiv.org/abs/1703.06870.

[2] Lin, Tsung-Yi, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. “Microsoft COCO: Common Objects in Context,” May 1, 2014. https://arxiv.org/abs/1405.0312v3.

See Also

`maskrcnn` | `trainMaskRCNN` | `segmentObjects` | `transform` | `insertObjectMask`

More About

- “Getting Started with Mask R-CNN for Instance Segmentation” on page 17-55
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Datastores for Deep Learning” (Deep Learning Toolbox)

Object Detection Using YOLO v4 Deep Learning

This example shows how to detect objects in images using you only look once version 4 (YOLO v4) deep learning network. In this example, you will

- Configure a dataset for training, validation, and testing of YOLO v4 object detection network. You will also perform data augmentation on the training dataset to improve the network efficiency.
- Compute anchor boxes from the training data to use for training the YOLO v4 object detection network.
- Create a YOLO v4 object detector by using the `yolov4ObjectDetector` function and train the detector using `trainYOLOv4ObjectDetector` function.

This example also provides a pretrained YOLO v4 object detector to use for detecting vehicles in an image. The pretrained network uses CSPDarkNet-53 as the backbone network and is trained on a vehicle dataset. For information about YOLO v4 object detection network, see “Getting Started with YOLO v4” on page 17-44.

Load Dataset

This example uses a small vehicle dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 datasets, available at the Caltech Computational Vision website created by Pietro Perona and used with permission. Each image contain one or two labeled instances of a vehicle. A small dataset is useful for exploring the YOLO v4 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. The first column contain the image file paths and the second column contain the bounding boxes.

```
% Display first few rows of the data set.
vehicleDataset(1:4,:)
```

```
ans=4×2 table
              imageFilename                    vehicle
    ____________________________________    _______________

    {'vehicleImages/image_00001.jpg'}    {[220 136 35 28]}
    {'vehicleImages/image_00002.jpg'}    {[175 126 61 45]}
    {'vehicleImages/image_00003.jpg'}    {[108 120 45 33]}
    {'vehicleImages/image_00004.jpg'}    {[124 112 38 36]}
```

```
% 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("default");
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);
```

Use `validateInputData` to detect invalid images, bounding boxes or labels i.e.,

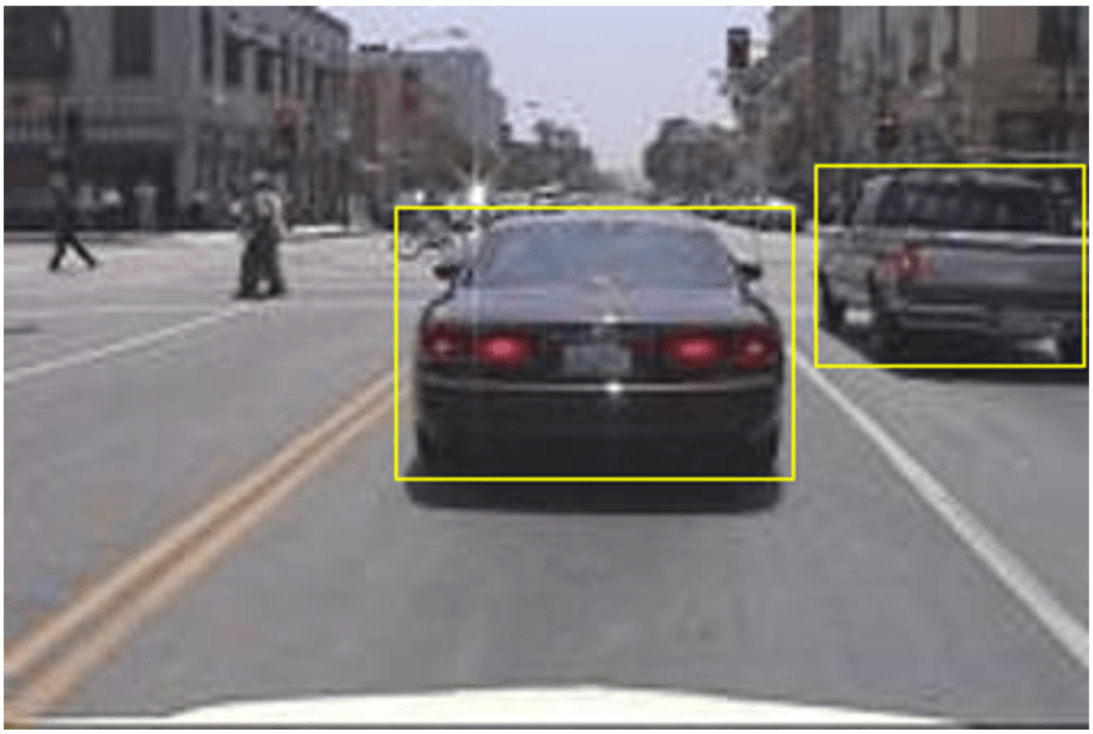
- Samples with invalid image format or containing NaNs
- Bounding boxes containing zeros/NaNs/Infs/empty
- Missing/non-categorical labels.

The values of the bounding boxes must be finite positive integers and must not be NaN. The height and the width of the bounding box values must be positive and lie within the image boundary.

```
validateInputData(trainingData);
validateInputData(validationData);
validateInputData(testData);
```

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)
```



```
reset(trainingData);
```

Create a YOLO v4 Object Detector Network

Specify the network input size to be used for training.

```
inputSize = [608 608 3];
```

Specify the name of the object class to detect.

```
className = "vehicle";
```

Use the `estimateAnchorBoxes` function 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 size of the network by using the `preprocessData` helper function.

```
rng("default")
trainingDataForEstimation = transform(trainingData,@(data)preprocessData(data,inputSize));
numAnchors = 9;
[anchors,meanIoU] = estimateAnchorBoxes(trainingDataForEstimation,numAnchors);

area = anchors(:, 1).*anchors(:,2);
[~,idx] = sort(area,"descend");
```

```
anchors = anchors(idx,:);
anchorBoxes = {anchors(1:3,:)
    anchors(4:6,:)
    anchors(7:9,:)
    };
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” on page 3-319 (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” on page 17-32.

Create the YOLO v4 object detector by using the `yolov4ObjectDetector` function. specify the name of the pretrained YOLO v4 detection network trained on COCO dataset. Specify the class name and the estimated anchor boxes.

```
detector = yolov4ObjectDetector("csp-darknet53-coco",className,anchorBoxes,InputSize=inputSize);
```

Perform Data Augmentation

Perform data augmentation to improve training accuracy. Use the `transform` function to apply custom data augmentations to the training data. The `augmentData` helper function applies the following augmentations to the input data:

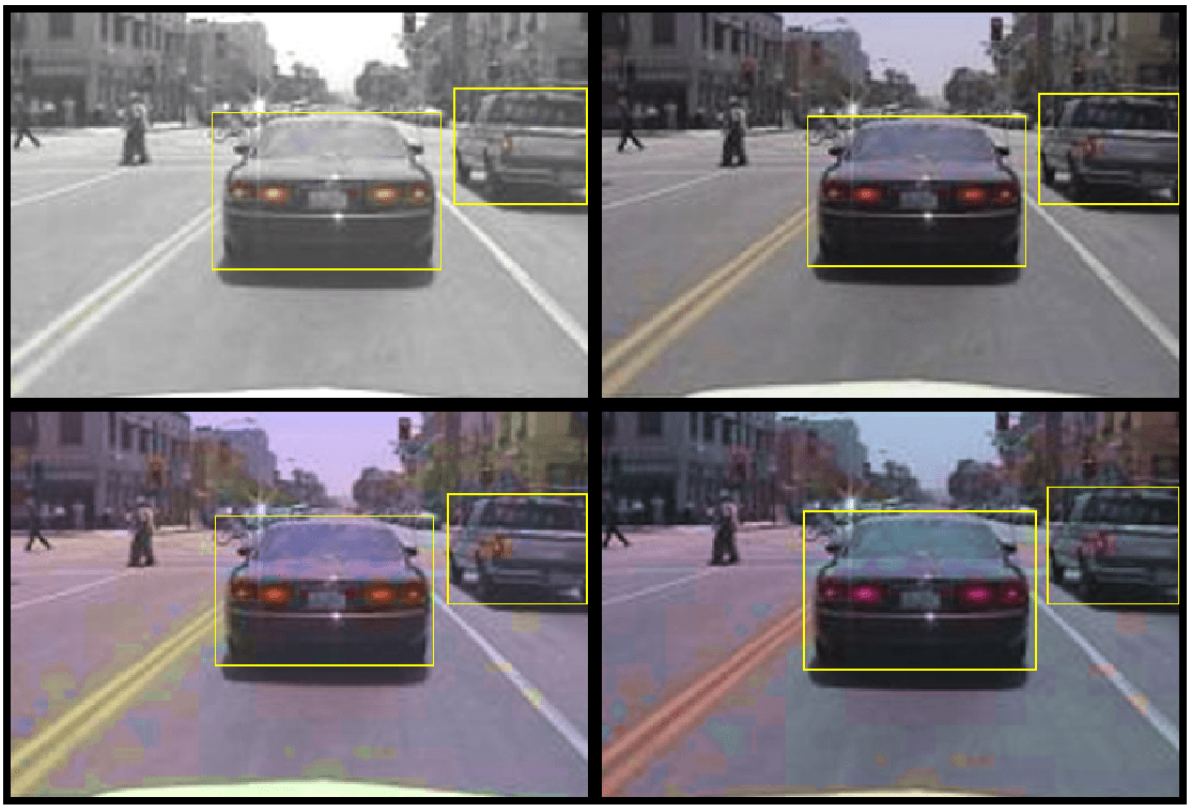
- Color jitter augmentation in HSV space
- Random horizontal flip
- Random scaling by 10 percent

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 and display samples of 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)
```



Specify Training Options

Use `trainingOptions` to specify network training options. Train the object detector using the Adam solver for 70 epochs with a constant learning rate 0.001. "`ResetInputNormalization`" should be set to false and `BatchNormalizationStatistics` should be set to "`moving`". Set "`ValidationData`" to the validation data. Use "`ExecutionEnvironment`" to determine what hardware resources will be used to train the network. Default value for this is "auto" which selects a GPU if it is available, otherwise selects the CPU. 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("adam",...
    GradientDecayFactor=0.9,...
    SquaredGradientDecayFactor=0.999,...
    InitialLearnRate=0.001,...
    LearnRateSchedule="none",...
    MiniBatchSize=4,...
    L2Regularization=0.0005,...
    MaxEpochs=70,...
    BatchNormalizationStatistics="moving",...
    DispatchInBackground=true,...
    ResetInputNormalization=false,...
    Shuffle="every-epoch",...
    VerboseFrequency=20,...
```

```
    CheckpointPath=tempdir,...
    ValidationData=validationData);
```

Train YOLO v4 Object Detector

Use the `trainYOLOv4ObjectDetector` function to train YOLO v4 object detector. This example is run on an NVIDIA™ Titan RTX GPU with 24 GB of memory. Training this network took approximately 10 hours using this setup. The training time will vary depending on the hardware you use. Instead of training the network, you can also use a pretrained YOLO v4 object detector in the Computer Vision Toolbox ™.

Download the pretrained detector by using the `downloadPretrainedYOLOv4Detector` helper function. Set the `doTraining` value to false. If you want to train the detector on the augmented training data, set the `doTraining` value to true.

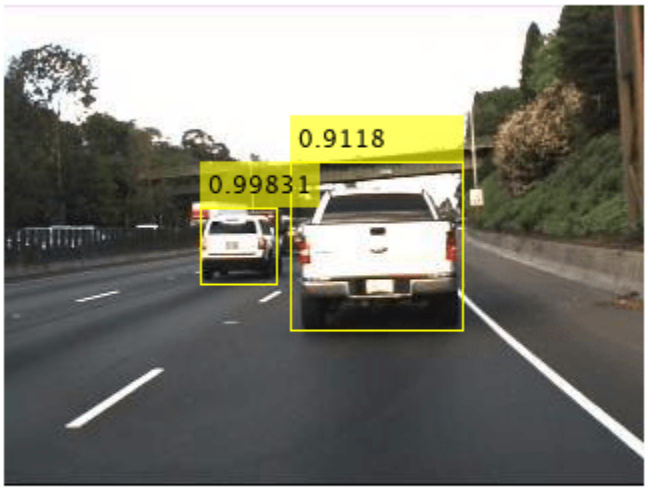
```
doTraining = false;
if doTraining
    % Train the YOLO v4 detector.
    [detector,info] = trainYOLOv4ObjectDetector(augmentedTrainingData,detector,options);
else
    % Load pretrained detector for the example.
    detector = downloadPretrainedYOLOv4Detector();
end
```

Run the detector on a test image.

```
I = imread("highway.png");
[bboxes,scores,labels] = 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).

Run the detector on all the test images.

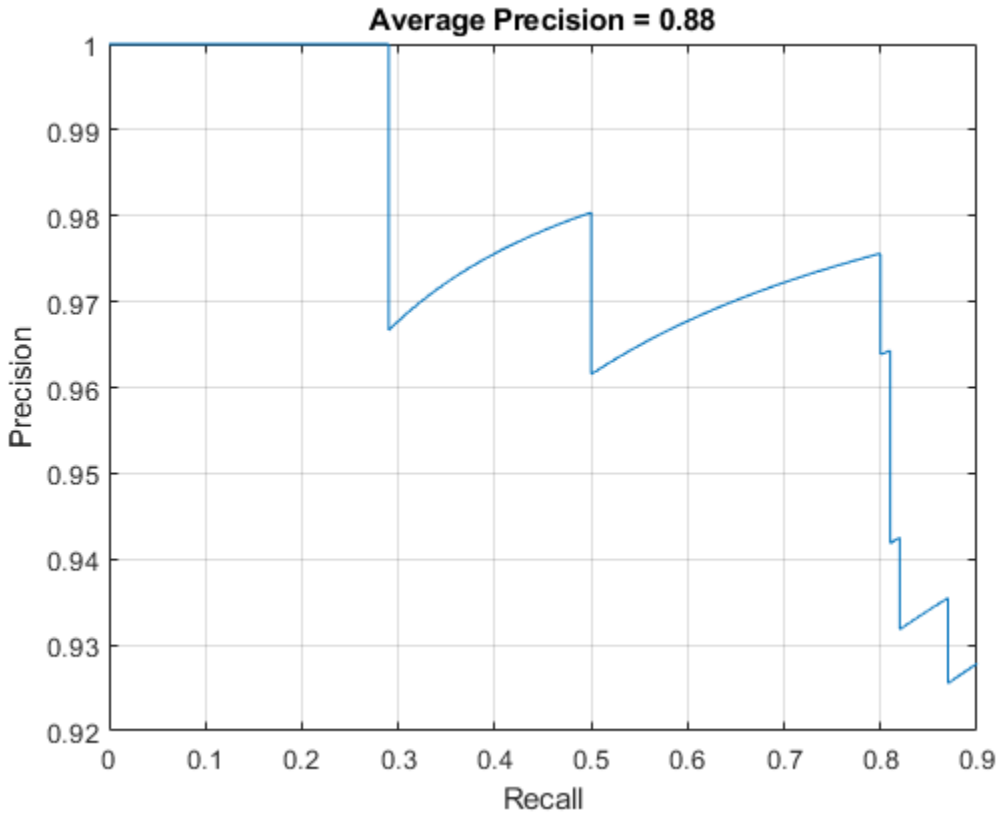
```
detectionResults = detect(detector,testData);
```

Evaluate the object detector using 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

Helper function for performing data augmentation.

```
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.
    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);

    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
```

Helper function for downloading the pretrained YOLO v4 object detector.

```
function detector = downloadPretrainedYOLOv4Detector()
% Download a pretrained yolov4 detector.
if ~exist("yolov4CSPDarknet53VehicleExample_22a.mat", "file")
    if ~exist("yolov4CSPDarknet53VehicleExample_22a.zip", "file")
        disp("Downloading pretrained detector...");
        pretrainedURL = "https://ssd.mathworks.com/supportfiles/vision/data/yolov4CSPDarknet53Vel
        websave("yolov4CSPDarknet53VehicleExample_22a.zip", pretrainedURL);
    end
    unzip("yolov4CSPDarknet53VehicleExample_22a.zip");
end
pretrained = load("yolov4CSPDarknet53VehicleExample_22a.mat");
detector = pretrained.detector;
end
```

References

[1] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. “YOLOv4: Optimal Speed and Accuracy of Object Detection.” 2020, *arXiv:2004.10934*. https://arxiv.org/abs/2004.10934.

See Also

`yolov4ObjectDetector` | `trainYOLOv4ObjectDetector` | `detect` | `evaluateDetectionPrecision` | `trainingOptions` | `transform`

Related Examples

- “Object Detection Using YOLO v3 Deep Learning” on page 3-323
- “Multiclass Object Detection Using YOLO v2 Deep Learning” on page 3-3

More About

- “Getting Started with YOLO v4” on page 17-44
- “Anchor Boxes for Object Detection” on page 17-32
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

Feature Detection and Extraction Examples

- “Automatically Detect and Recognize Text Using MSER and OCR” on page 4-2
- “Automatically Detect and Recognize Text Using Pretrained CRAFT Network and OCR” on page 4-14
- “Digit Classification Using HOG Features” on page 4-21
- “Find Image Rotation and Scale Using Automated Feature Matching” on page 4-29
- “Feature Based Panoramic Image Stitching” on page 4-34
- “Cell Counting” on page 4-40
- “Object Counting” on page 4-43
- “Pattern Matching” on page 4-45
- “Recognize Text Using Optical Character Recognition (OCR)” on page 4-50
- “Cell Counting” on page 4-64

Automatically Detect and Recognize Text Using MSER and OCR

This example shows how to detect regions in an image that contain text by using maximally stable extremal regions (MSER) feature detector. 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.

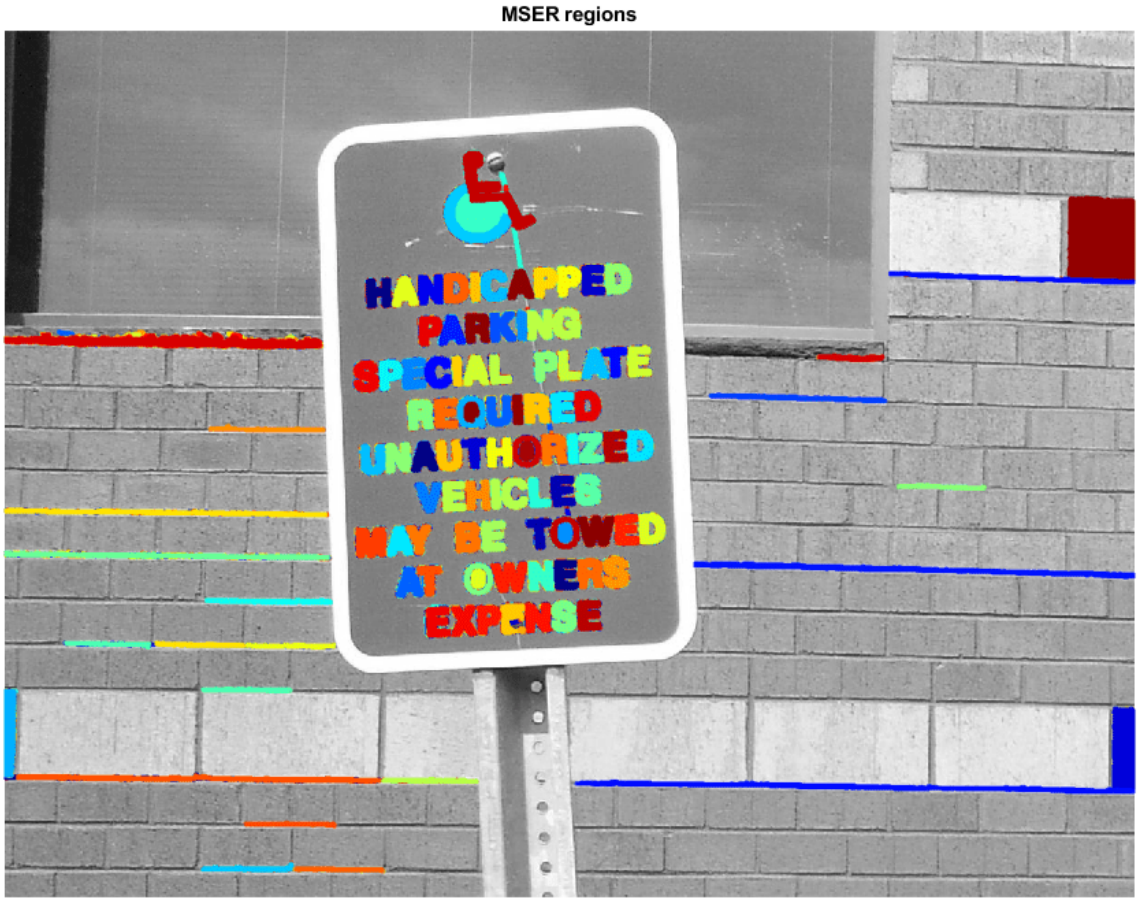
Segmenting text from an unstructured scene greatly helps with additional tasks such as optical character recognition (OCR). The automated text detection algorithm in this example detects a large number of text region candidates and progressively removes those less likely to contain text.

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 = im2gray(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
```



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
```

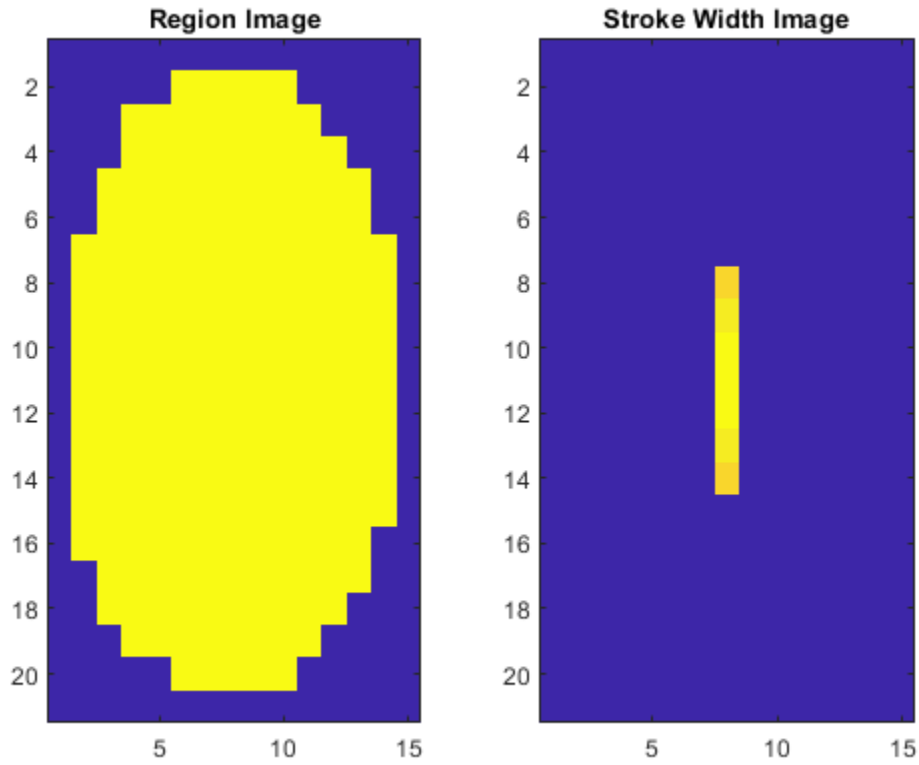
Step 3: Remove Non-Text Regions Based On Stroke Width Variation

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:

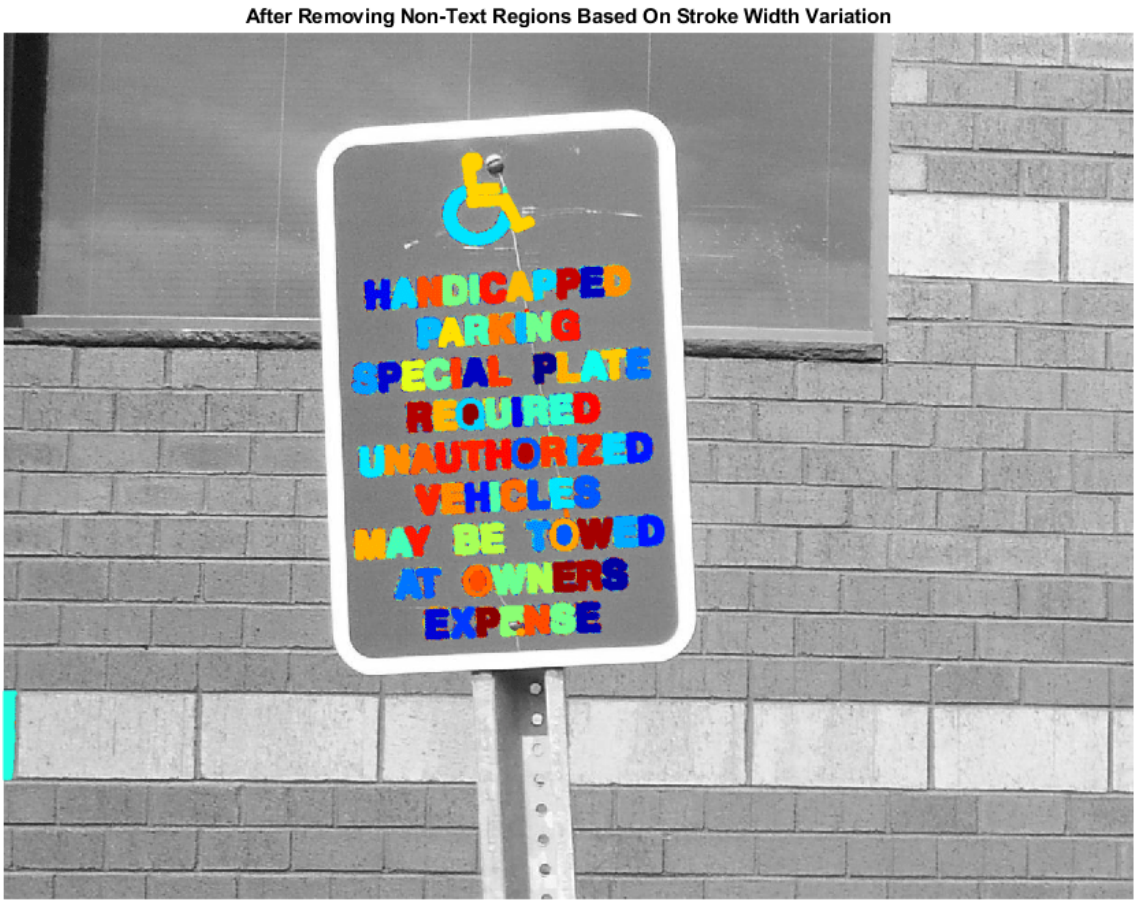
```
% Compute the stroke width variation metric
strokeWidthValues = distanceImage(skeletonImage);
strokeWidthMetric = std(strokeWidthValues)/mean(strokeWidthValues);
```

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
```



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','I'}, 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(mserStats.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
expandedBBoxes = [xmin ymin xmax-xmin+1 ymax-ymin+1];
IExpandedBBoxes = insertShape(colorImage,'Rectangle',expandedBBoxes,'LineWidth',3);

figure
imshow(IExpandedBBoxes)
title('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 `bboxOverlapRatio` 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 = bboxOverlapRatio(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')
```

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.

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.

See Also

Related Examples

- “Recognize Text Using Optical Character Recognition (OCR)” on page 4-50
- “Automatically Detect and Recognize Text Using Pretrained CRAFT Network and OCR” on page 4-14

Automatically Detect and Recognize Text Using Pretrained CRAFT Network and OCR

This example shows how to perform text recognition by using the character region awareness for text detection (CRAFT) deep learning model and optical character recognition (OCR). In the example, you use a pretrained CRAFT deep learning network to detect the text regions in the input image. You can modify the region threshold and the affinity threshold values of the CRAFT model to localise an entire paragraph, a sentence or a word. Then, you use OCR to recognize the characters in the detected text regions.

Read Image

Read an image into the MATLAB® workspace.

```
I = imread("handicapSign.jpg");
```

Detect Text Regions

Detect text regions in the input image by using the `detectTextCRAFT` function. The `CharacterThreshold` value is the region threshold to use for localizing each character in the image. The `LinkThreshold` value is the affinity threshold that defines the score for grouping two detected texts into a single instance. You can fine-tune the detection results by modifying the region and affinity threshold values. Increase the value of the affinity threshold for more word-level and character-level detections. For information about the effect of the affinity threshold on the detection results, see the “Detect Characters by Modifying Affinity Threshold” example.

To detect each word on the parking sign, set the value of the region threshold to 0.3. The default value for the affinity threshold is 0.4. The output is a set of bounding boxes that localize the words in the image scene. The bounding box specifies the spatial coordinates of the detected text regions in the image.

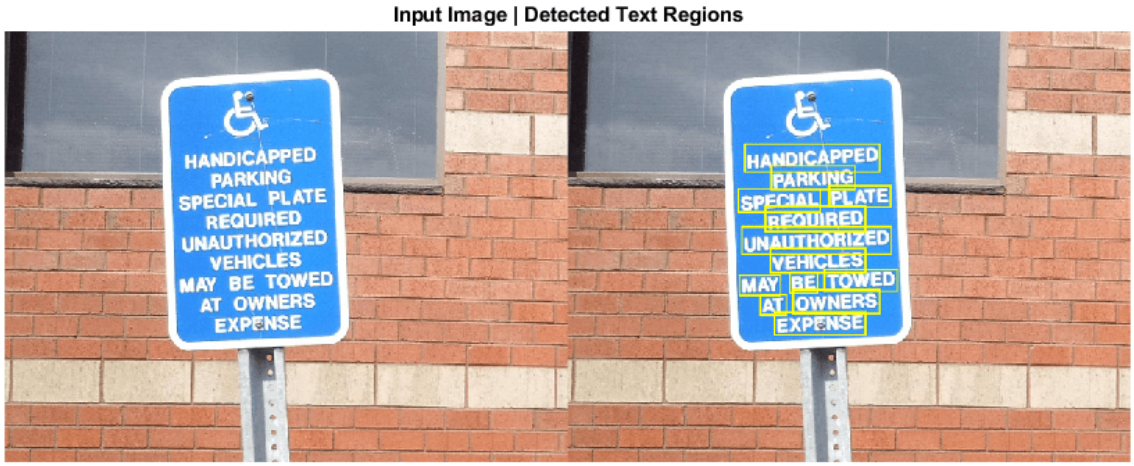
```
bbox = detectTextCRAFT(I,CharacterThreshold=0.3);
```

Draw the output bounding boxes on the image by using the `insertShape` function.

```
Iout = insertShape(I,"Rectangle",bbox,LineWidth=4);
```

Display the input image and the output text detections.

```
fig = figure(Position=[1 1 600 600]);
ax = gca;
montage({I;Iout},Parent=ax);
title("Input Image | Detected Text Regions")
```



Recognize Text

Recognize the text within the bounding boxes by using the `ocr` function and display the results. The output is an `ocrText` object containing information about the recognized text, the recognition confidence, and the location of the text in the original image.

```
output = ocr(I,bbox);
```

Display the recognized words.

```
disp([output.Words])
```

```
  Columns 1 through 4

    {'SPECIAL'}    {'MAY'}    {'unififfioiglgen'}    {'HANoIcI§I_>I_>...'}

  Columns 5 through 8

    {'E'E_(v)U|R\E|3'}    {'VEHICLES'}    {'FARKING'}    {'EXPENSE'}

  Columns 9 through 11

    {'‘owuens'}    {'TOWED'}    {'PLATE'}
```

Analyze Recognition Results

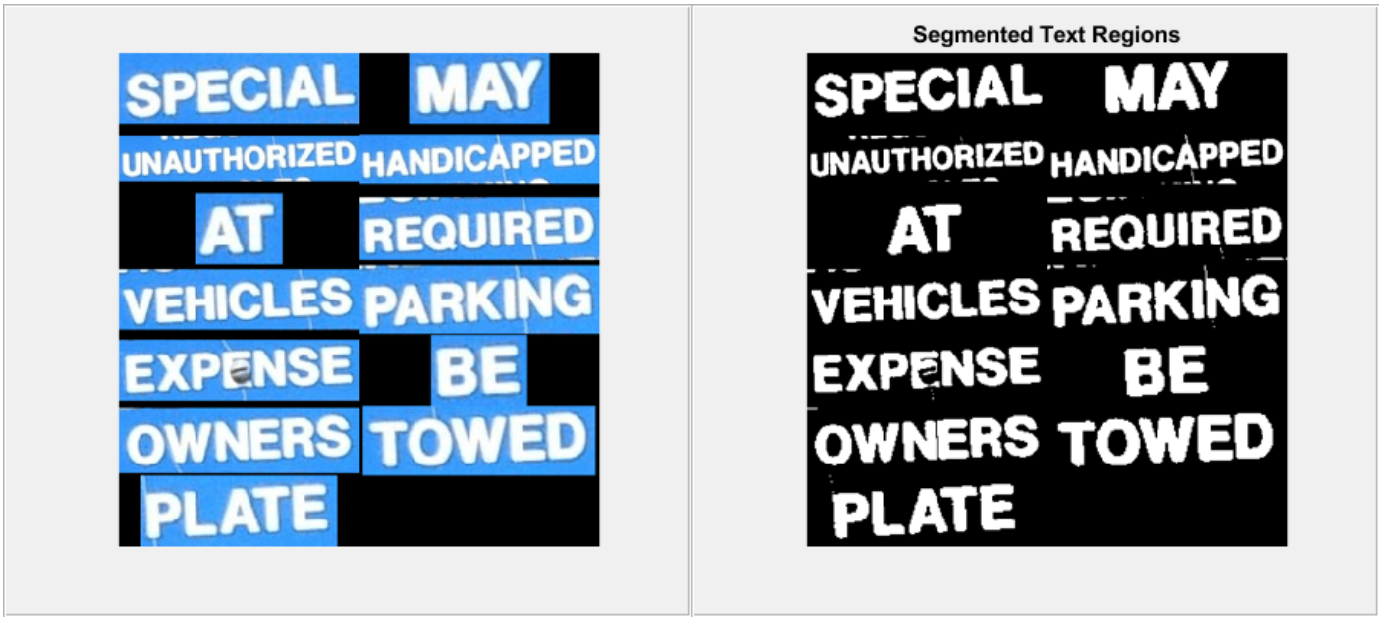
Out of 13 words on the parking sign only 6 words are recognized correctly. The words `UNAUTHORIZED`, `HANDICAPPED`, `REQUIRED`, `PARKING`, `BE`, and `OWNERS` are not recognized correctly. The performance of the OCR method depend on the text detection results and the characteristics of the image background. As a preprocessing step, the `ocr` function performs binarization to segment the text regions from the background. For good segmentation results, the image background must be uniform and the text regions must have high contrast over the background. Otherwise, the segmented text regions might have outliers which in turn affect the recognition results.

You can use `imbinarize` function to check the initial binarization step because both `ocr` and the default '`global`' method in `imbinarize` use Otsu's method for image binarization.

```
Idet = cell(1,size(bbox,1));
Iseg = cell(1,size(bbox,1));
for i=1:size(bbox,1)
    roi = bbox(i,:);
    Idet{i} = I(roi(2):roi(2)+roi(4),roi(1):roi(1)+roi(3),:);
    Iseg{i} = imbinarize(rgb2gray(Idet{i}));
end
```

Display the text detections obtained using `detectTextCRAFT` function and the corresponding segmentation results. You can notice that the segmented text regions corresponding to the words `UNAUTHORIZED`, `HANDICAPPED`, and `REQUIRED` contain outliers.

```
fig1 = figure;
set(fig1,Position=[1 1 900 400])
hPanel1 = uipanel(fig1,Position=[0 0 0.5 1]);
hPlot1 = axes(hPanel1);
hPanel2 = uipanel(fig1,Position=[0.5 0 0.5 1]);
hPlot2 = axes(hPanel2);
montage(Idet,Parent=hPlot1)
montage(Iseg,Parent=hPlot2)
title("Segmented Text Regions",Parent=hPlot2)
```



Improve Recognition Results

To improve the recognition results, preprocess the detected text regions so that the input to `ocr` is well localized detections and have no outliers.

Reduce the number of outliers in the segmented text regions by computing tightly localized detections using the `detectTextCRAFT` function. Increase the value of the region threshold to reduce the number of false detections and compute bounding boxes that tightly localize the text regions in the image. Set the value of `CharacterThreshold` parameter to 0.55 and compute the text detections. The default value for the affinity threshold is 0.4.

```
newBbox = detectTextCRAFT(I,CharacterThreshold=0.55);
```

Draw the output bounding boxes on the image by using the `insertShape` function. Display the detection results. You can notice that the text regions in the image are tightly localized.

```
Iout = insertShape(I,"Rectangle",newBbox,LineWidth=3);
figure
imshow(Iout)
title("Detected Text Regions for Region Threshold = 0.55")
```

Recognize the text within the bounding boxes by using the `ocr` function and display the recognized words.

```
output = ocr(I,newBbox);
disp([output.Words])
```

```
  Columns 1 through 4

    {'SPECIAL'}    {'UNAUTHORIZED'}    {'HANDICAPPED'}    {'REQUIRED'}

  Columns 5 through 8

    {'VEHICLES'}    {'PARKING'}    {'EXPENSE'}    {'PLATE'}
```

The words on the parking sign are correctly recognized except for these small words: `MAY`, `BE`, and `AT`. This is because the bounding box is too tight for OCR to detect small length words.

Follow these preprocessing steps to further improve the recognition accuracy.

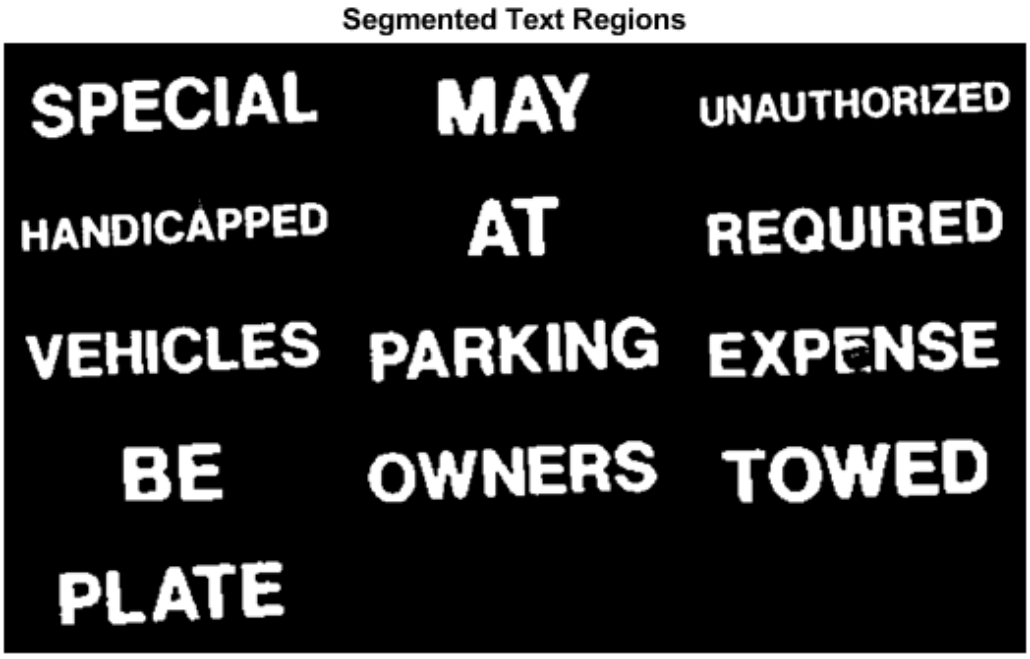
- Adjust the contrast of the detected text regions by using `imadjust` function. Contrast enhancement improves segmentation accuracy when the input is a low contrast image.
- Segment the text regions from the image background by using `imbinarize` function. You can also use other segmentation methods like k-means clustering or adaptive thresholding based on the complexity of the image scene.
- Pad the image by adding more pixels along the image boundary. If the intensity value of the foreground text region is `1` then the padding pixel value must be `0` and vice-versa.
- Perform morphological erosion to remove small outliers in the segmented text region, if any.

Recognize the text within the bounding boxes by using the `ocr` function. To remove blank spaces, use `deblank` function.

```
Icorrect = cell(1,size(newBbox,1));
finalOutput = cell(1,size(newBbox,1));
recognizedWords = cell(1,size(newBbox,1));
for i=1:size(newBbox,1)
    roi = newBbox(i,:);
    Icrop = I(roi(2):roi(2)+roi(4),roi(1):roi(1)+roi(3),:);
    Ipreprocess = rgb2gray(Icrop);
    Ipreprocess = imadjust(Ipreprocess);
    Isegment = imbinarize(Ipreprocess);
    Isegment = padarray(Isegment,[15 15],0,'both');
    se = strel('square',2);
    Icorrect{i} = imerode(Isegment,se);
    finalOutput{i} = ocr(Icorrect{i});
    recognizedWords{i} = [deblank(finalOutput{i}.Text)];
end
```

Display the segmentation results. Now the text detections are tightly localized, the segmented texts does not have any outliers. The area of the segmented text regions has also increased because of padding.

```
figure(Position=[1 1 400 400]);
ax3 = gca;
montage(Icorrect,Parent=ax3)
title("Segmented Text Regions")
```



Display the results and annotate the recognized words on the parking sign.

```
disp(recognizedWords)
```

```
  Columns 1 through 5

    {'SPECIAL'}    {'MAY'}    {'UNAUTHORIZED'}    {'HANDICAPPED'}    {'AT'}

  Columns 6 through 10

    {'REQUIRED'}    {'VEHICLES'}    {'PARKING'}    {'EXPENSE'}    {'BE'}

  Columns 11 through 13

    {'OWNERS'}    {'TOWED'}    {'PLATE'}
```

```
Iannotate = I;
for cnt = 1:size(finalOutput,2)
    if ~isempty(finalOutput{cnt}.Words)
        Iannotate = insertObjectAnnotation(Iannotate,"Rectangle",newBbox(cnt,:),finalOutput{cnt}
    end
end
figure
imshow(Iannotate)
```

See Also

`detectTextCRAFT` | `ocr` | `insertObjectAnnotation` | `insertShape` | `ocrText` | `deblank` | `imbinarize`

Related Examples

- “Recognize Text Using Optical Character Recognition (OCR)” on page 4-50
- “Automatically Detect and Recognize Text Using MSER and OCR” on page 4-2

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=10×2 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=10×2 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.

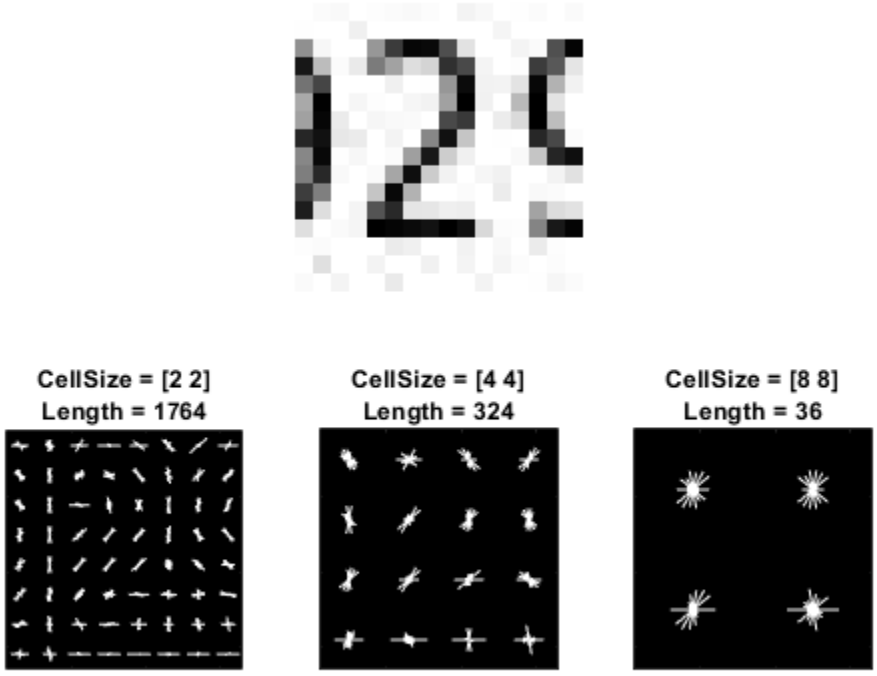
```
% Show pre-processing results
exTestImage = readimage(testSet,37);
processedImage = imbinarize(im2gray(exTestImage));

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 = im2gray(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, cellSi

% 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, cell
% 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 = im2gray(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.

See Also

Related Examples

- “Automatically Detect and Recognize Text Using MSER and OCR” on page 4-2
- “Recognize Text Using Optical Character Recognition (OCR)” on page 4-50

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');
```

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.

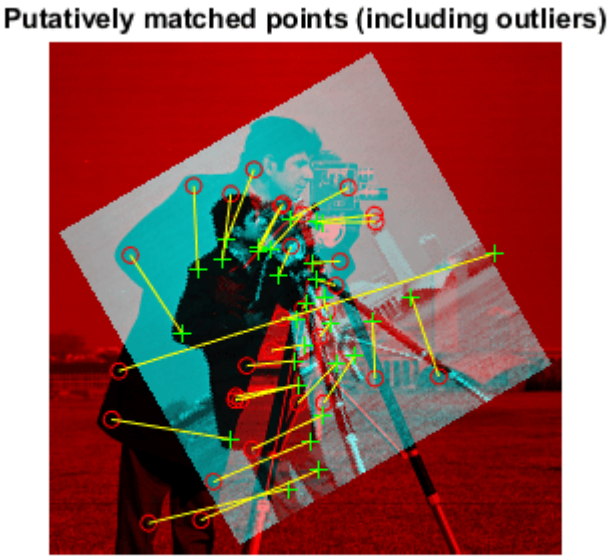
```
indexPairs = matchFeatures(featuresOriginal, featuresDistorted);
```

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)');
```



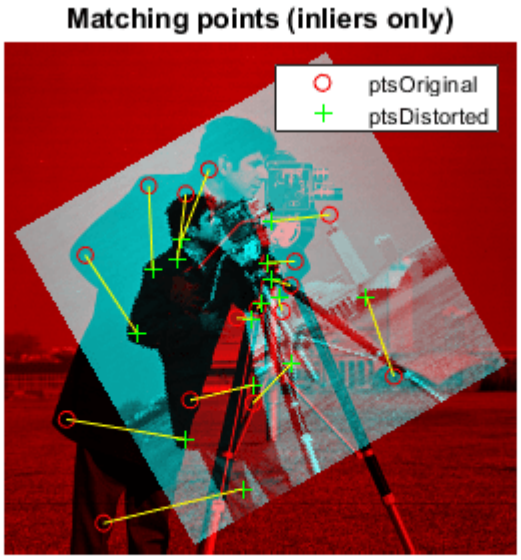
Step 4: Estimate Transformation

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 $T(n)*T(n-1)*\ldots*T(1)$.

```
% Read the first image from the image set.
I = readimage(buildingScene,1);

% Initialize features for I(1)
grayImage = im2gray(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 = im2gray(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.

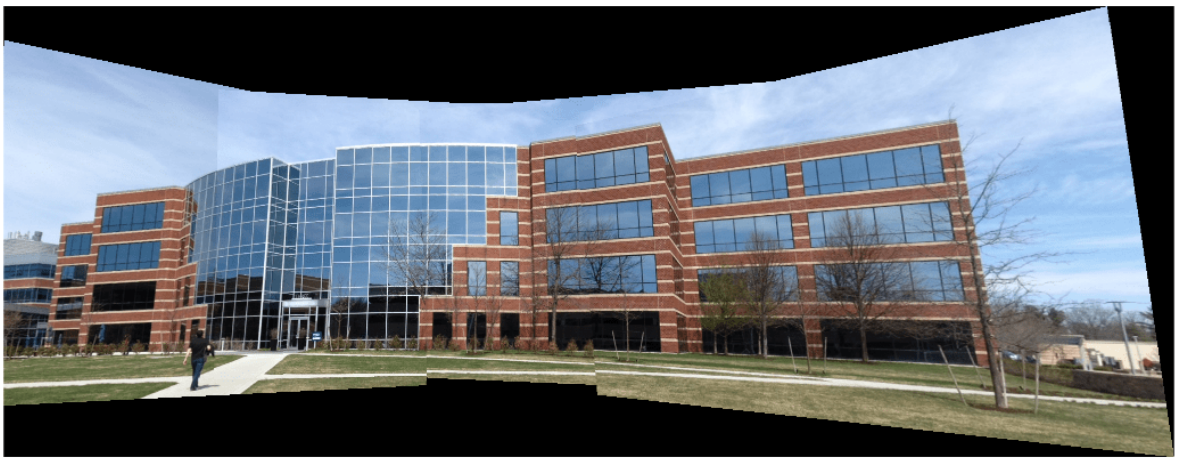
Use the `outputLimits` method to compute the minimum and maximum output limits over all transformations. These values are used to automatically compute the size of the panorama.

```
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.

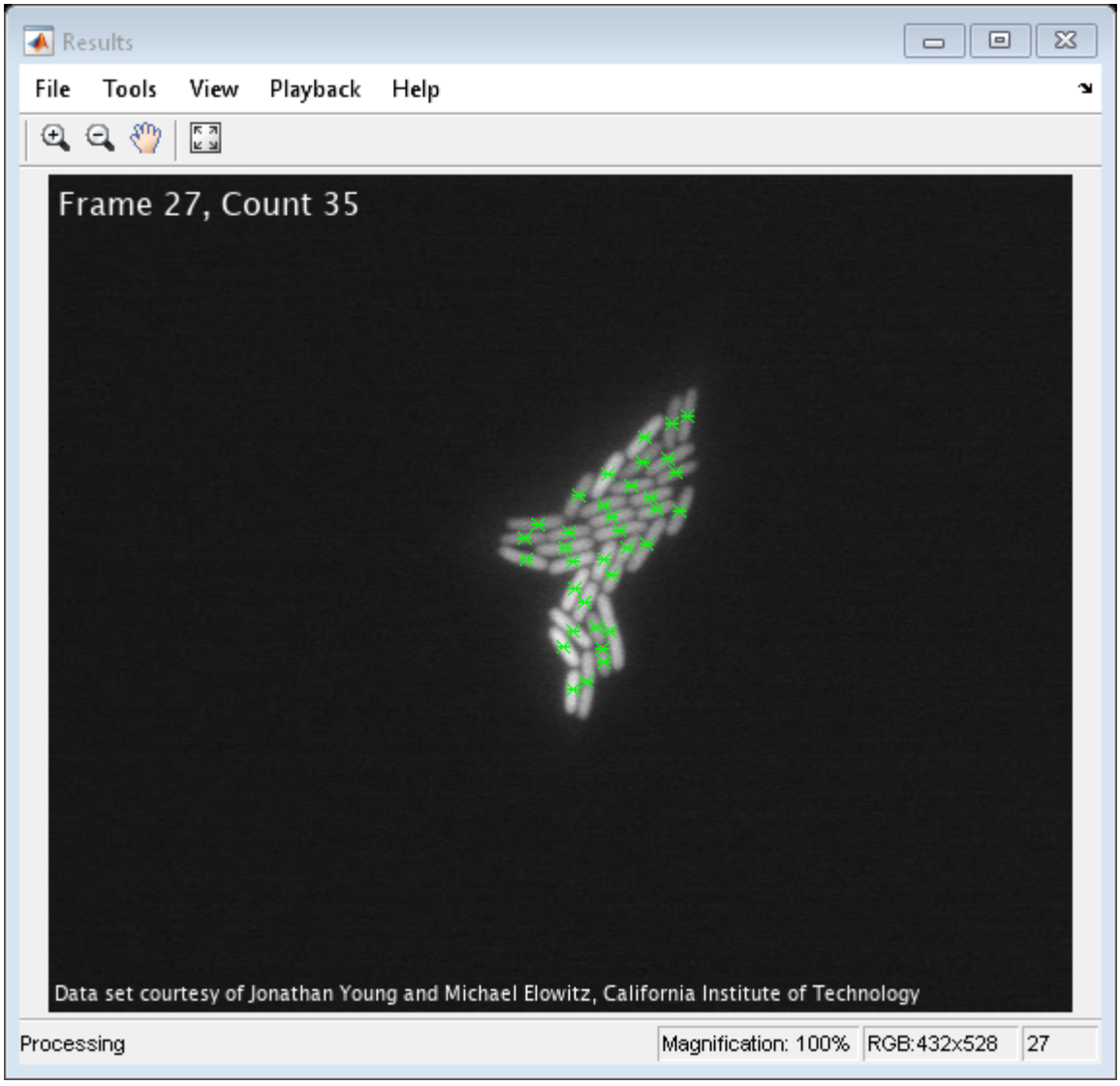
```
hVideo = vision.VideoPlayer;
hVideo.Name  = 'Results';
hVideo.Position(1) = round(hVideo.Position(1));
hVideo.Position(2) = round(hVideo.Position(2));
hVideo.Position([4 3]) = 30+VideoSize;
```

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 = im2gray(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;
    pause(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 = im2gray(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);
    pause(1);
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

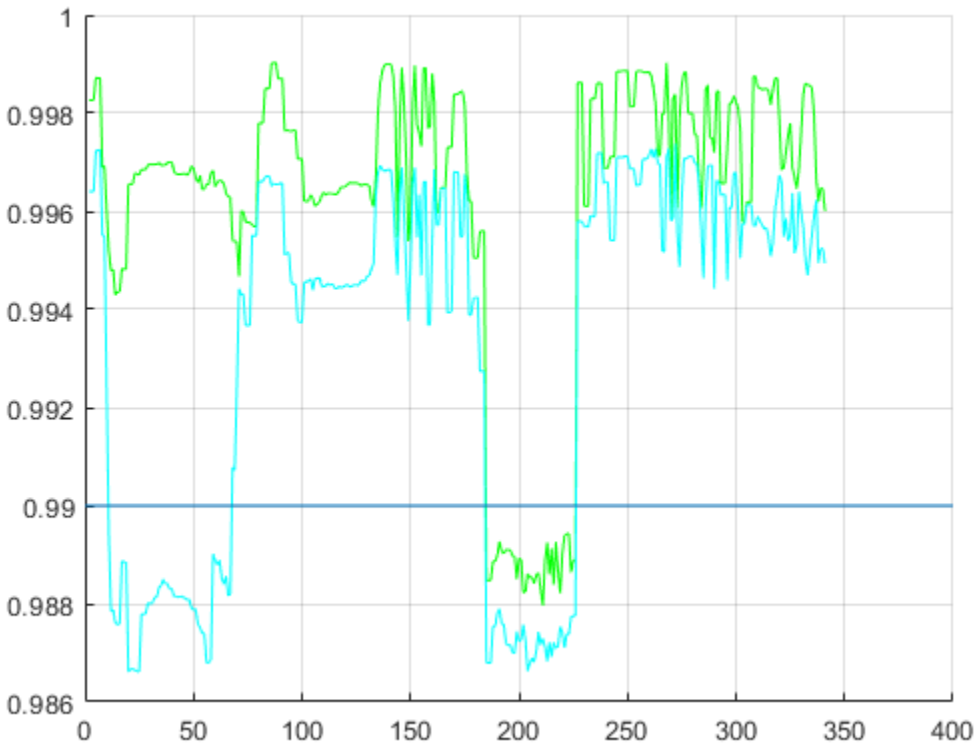
```
hPlot = videopatternplots('setup',numberOfTargets, threshold);
```

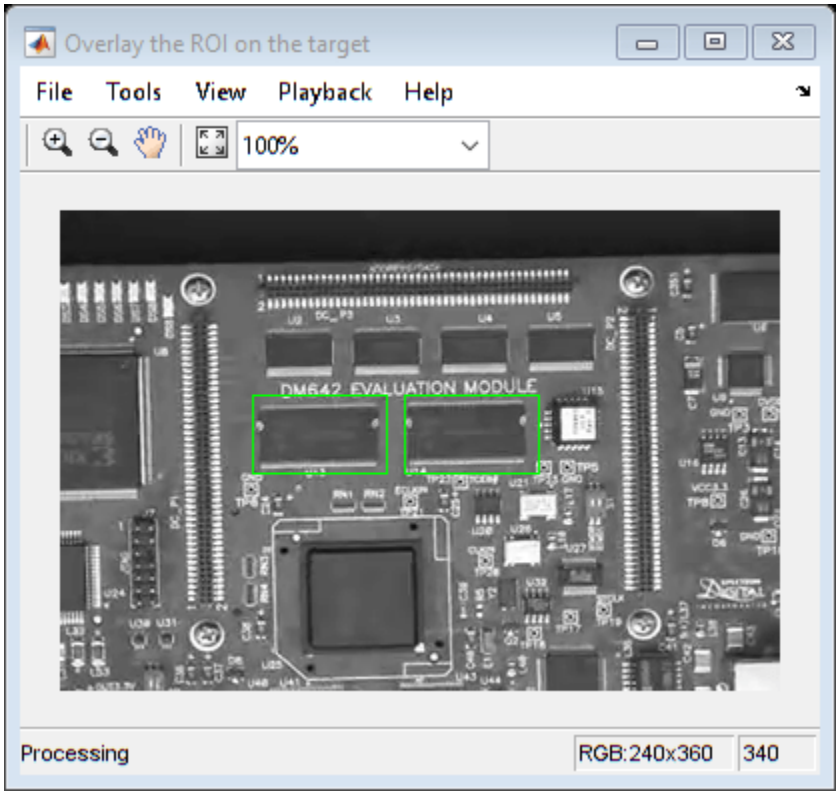
Search for a Template in Video

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
```


1
0.998
0.996
0.994
0.992
0.99
0.988
0.986
0
50
100
150
200
250
300
350
400



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.

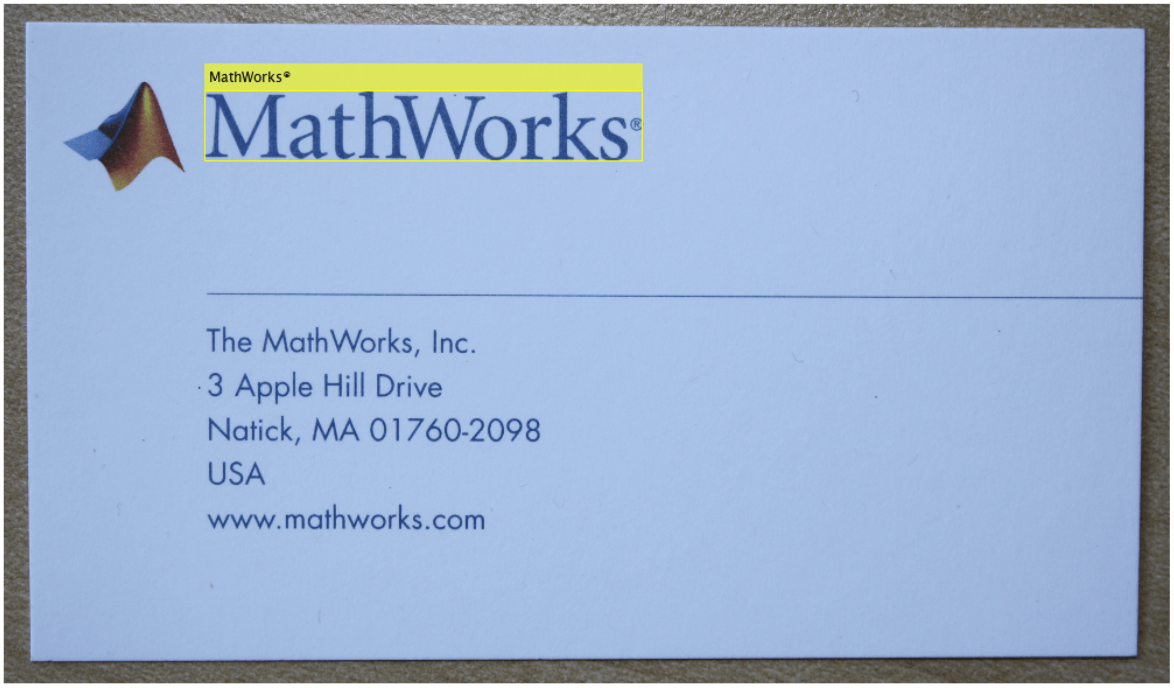
```
% Load an image.
I = imread('businessCard.png');

% 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 = im2gray(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.

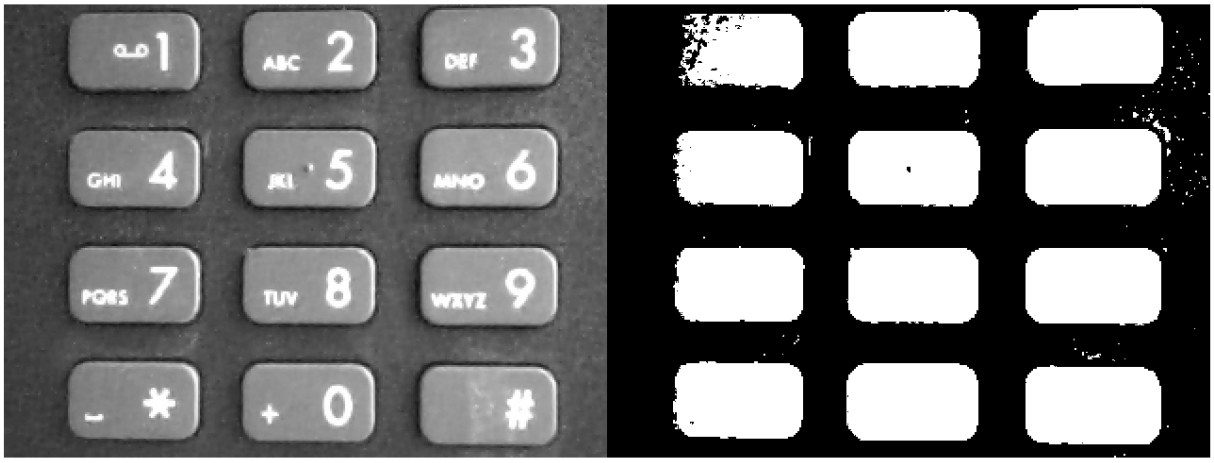
```
% Set 'TextLayout' to 'Block' to instruct ocr to assume the image
% contains just one block of text.
results = ocr(I,'TextLayout','Block');

results.Text
```

```
ans =

  0×0 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');
```

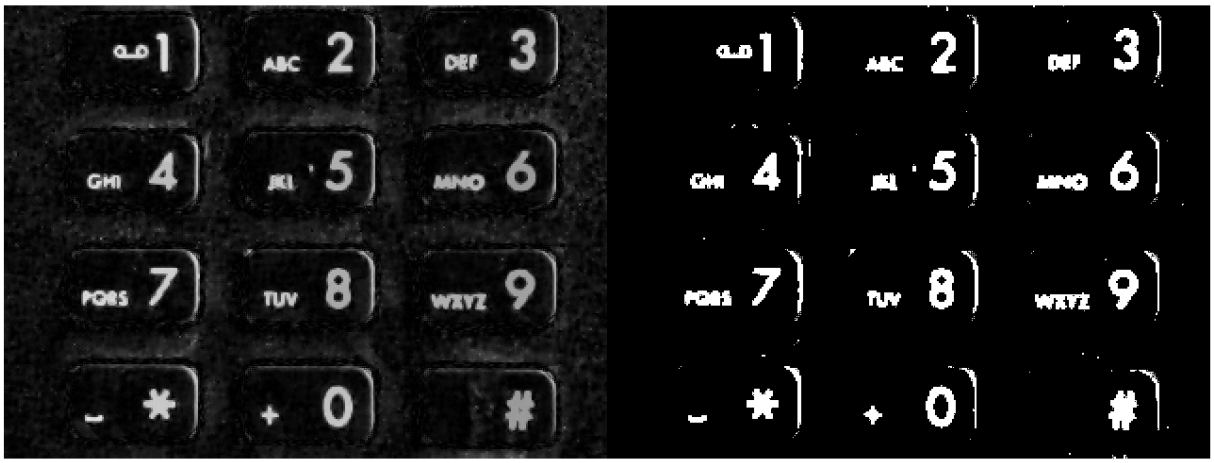


After thresholding, the binary image contains no text. This is why `ocr` failed to recognize any text in the original image. You can help improve the results by pre-processing the image to improve text segmentation. The next part of the example explores two useful pre-processing techniques.

Image Pre-processing Techniques To Improve Results

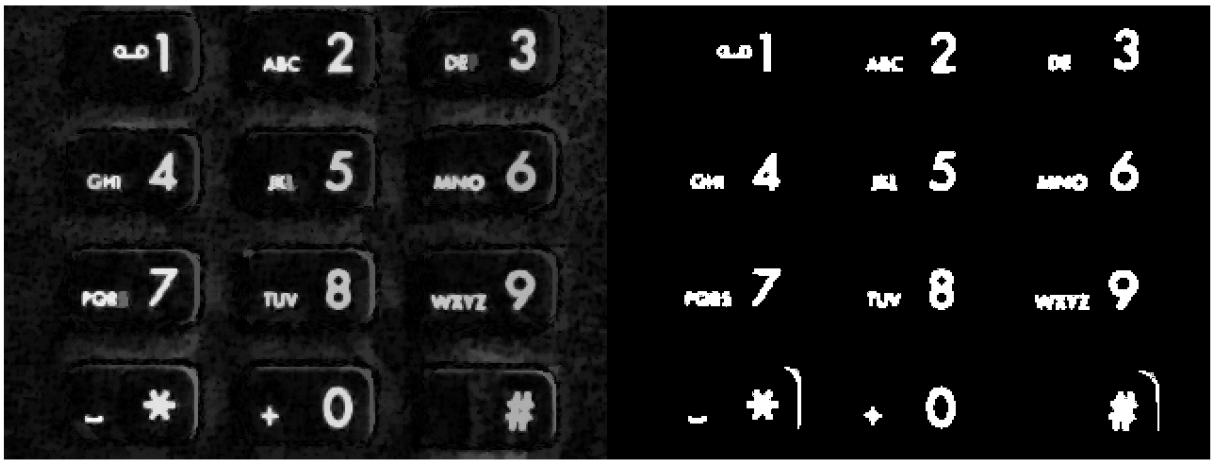
The poor text segmentation seen above is caused by the non-uniform background in the image, i.e. the light-gray keys surrounded by dark gray. You can use the following pre-processing technique to remove the background variations and improve the text segmentation. Additional details about this technique 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. For information about how to automatically detect text regions, see “Automatically Detect and Recognize Text Using MSER and OCR” on page 4-2 and “Automatically Detect and Recognize Text Using Pretrained CRAFT Network and OCR” on page 4-14. 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 “Automatically Detect and Recognize Text Using Pretrained CRAFT Network and OCR” on page 4-14 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.

See Also

Related Examples

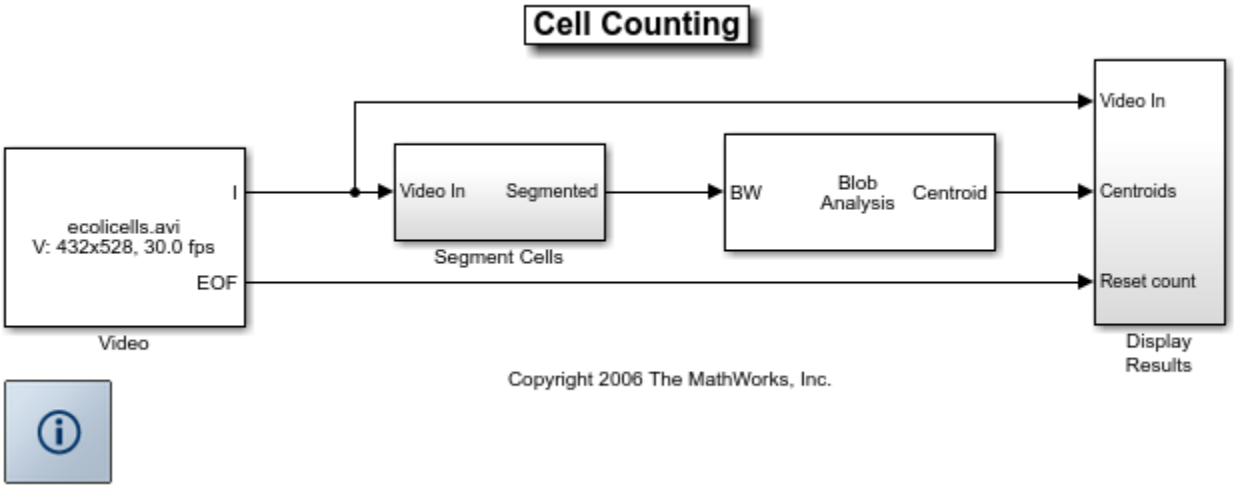
- “Automatically Detect and Recognize Text Using Pretrained CRAFT Network and OCR” on page 4-14
- “Automatically Detect and Recognize Text Using MSER and OCR” on page 4-2

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.

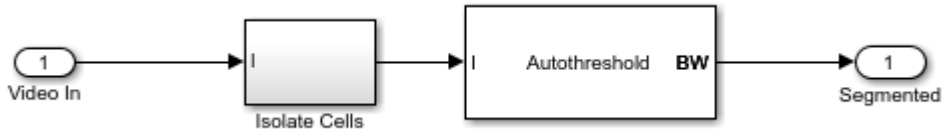
Example Model

The following figure shows the Cell Counting example model.

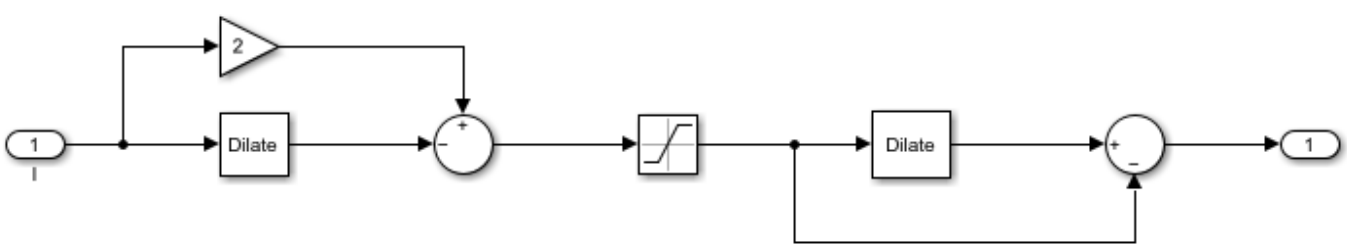


Segment Cells Subsystem

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.



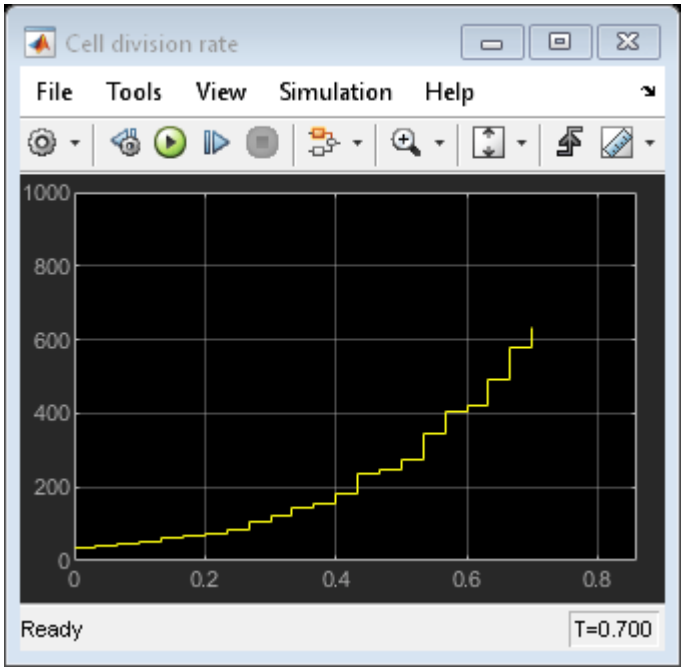
Isolate Cells subsystem:



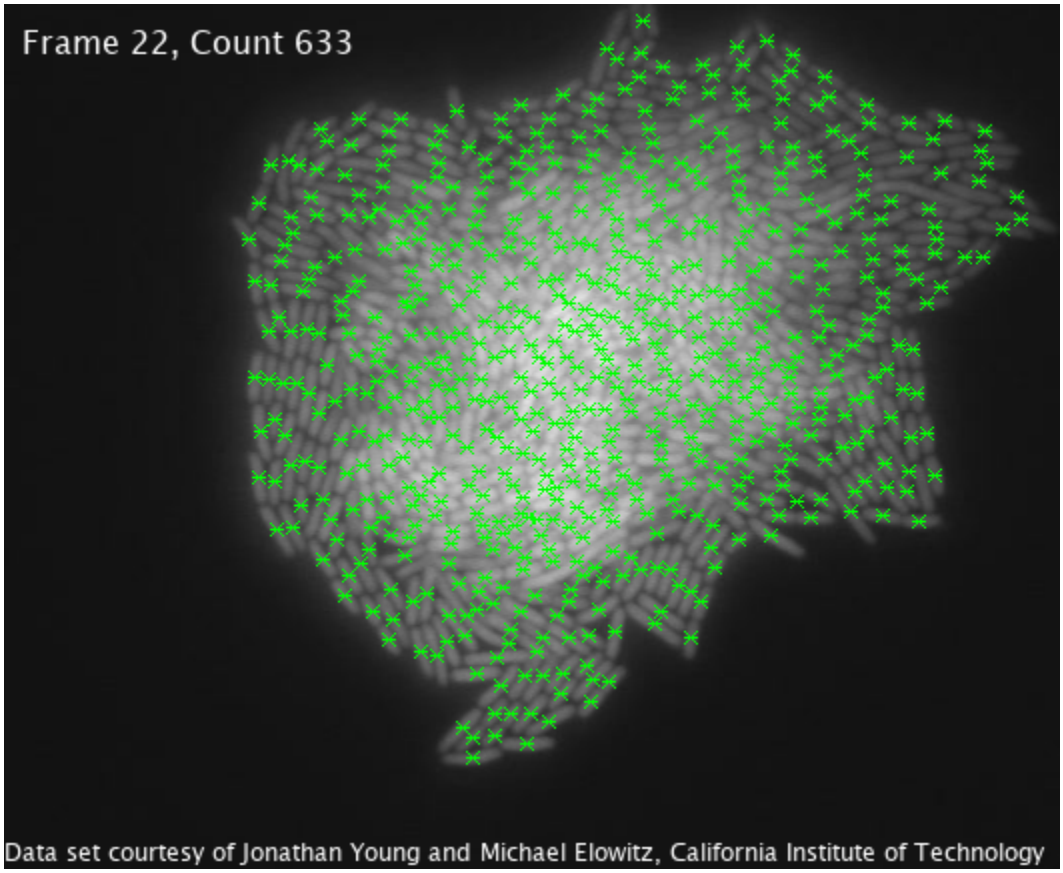
Cell Counting Results

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

- “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment” on page 5-2
- “Ground Plane and Obstacle Detection Using Lidar” on page 5-12
- “Augment Point Cloud Data For Deep Learning” on page 5-21
- “Import Point Cloud Data For Deep Learning” on page 5-26
- “Encode Point Cloud Data For Deep Learning” on page 5-30
- “Build a Map from Lidar Data” on page 5-35
- “Build a Map from Lidar Data Using SLAM” on page 5-55
- “3-D Point Cloud Registration and Stitching” on page 5-71

Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment

This example shows how to record synthetic lidar sensor data from a 3D simulation environment, and develop a simultaneous localization and mapping (SLAM) algorithm using the recorded data. The simulation environment uses the Unreal Engine® by Epic Games®.

Introduction

Automated Driving Toolbox™ integrates an Unreal Engine simulation environment in Simulink®. Simulink blocks related to this simulation environment can be found in the `drivingsim3d` library. These blocks provide the ability to:

- Select different scenes in the 3D simulation environment
- Place and move vehicles in the scene
- Attach and configure sensors on the vehicles
- Simulate sensor data based on the environment around the vehicle

This powerful simulation tool can be used to supplement real data when developing, testing, and verifying the performance of automated driving algorithms, making it possible to test scenarios that are difficult to reproduce in the real world.

In this example, you evaluate a lidar perception algorithm using synthetic lidar data generated from the simulation environment. The example walks you through the following steps:

- Record and visualize synthetic lidar sensor data from the simulation environment.
- Develop a perception algorithm to build a map using SLAM in MATLAB®.

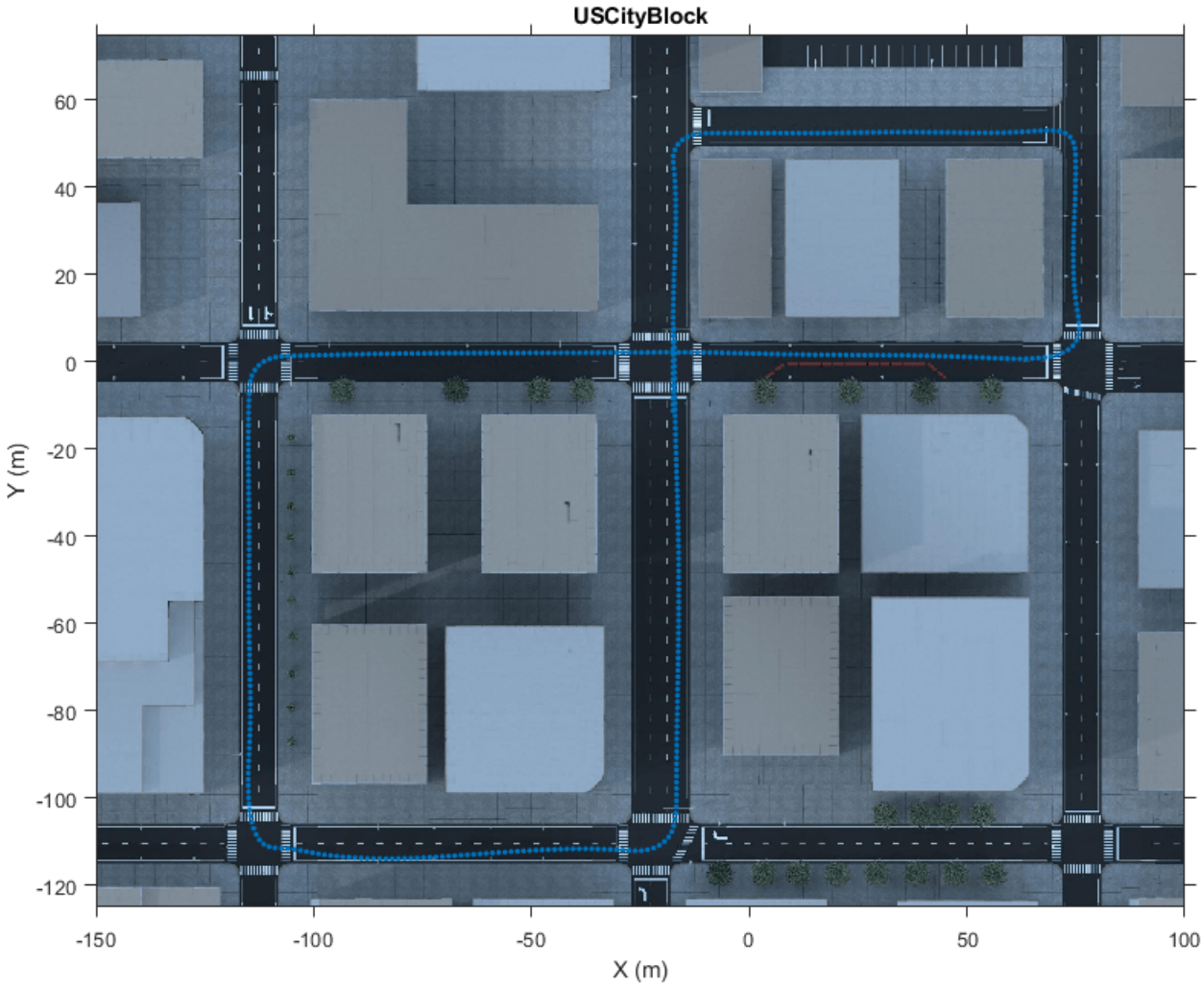
Set Up Scenario in Simulation Environment

First, set up a scenario in the simulation environment that can be used to test the perception algorithm. Use a scene depicting a typical city block with a single vehicle that is the vehicle under test. You can use this scene to test the performance of the algorithm in an urban road setting.

Next, select a trajectory for the vehicle to follow in the scene. The “Select Waypoints for Unreal Engine Simulation” (Automated Driving Toolbox) example describes how to interactively select a sequence of waypoints from a scene and generate a vehicle trajectory. This example uses a recorded drive segment obtained using the `helperSelectSceneWaypoints` function, as described in the waypoint selection example.

```
% Load reference path for recorded drive segment
xData   = load('refPosesX.mat');
yData   = load('refPosesY.mat');
yawData = load('refPosesT.mat');

% Set up workspace variables used by model
refPosesX = xData.refPosesX;
refPosesY = yData.refPosesY;
refPosesT = yawData.refPosesT;

% Display path on scene image
sceneName = 'USCityBlock';
hScene = figure;
```

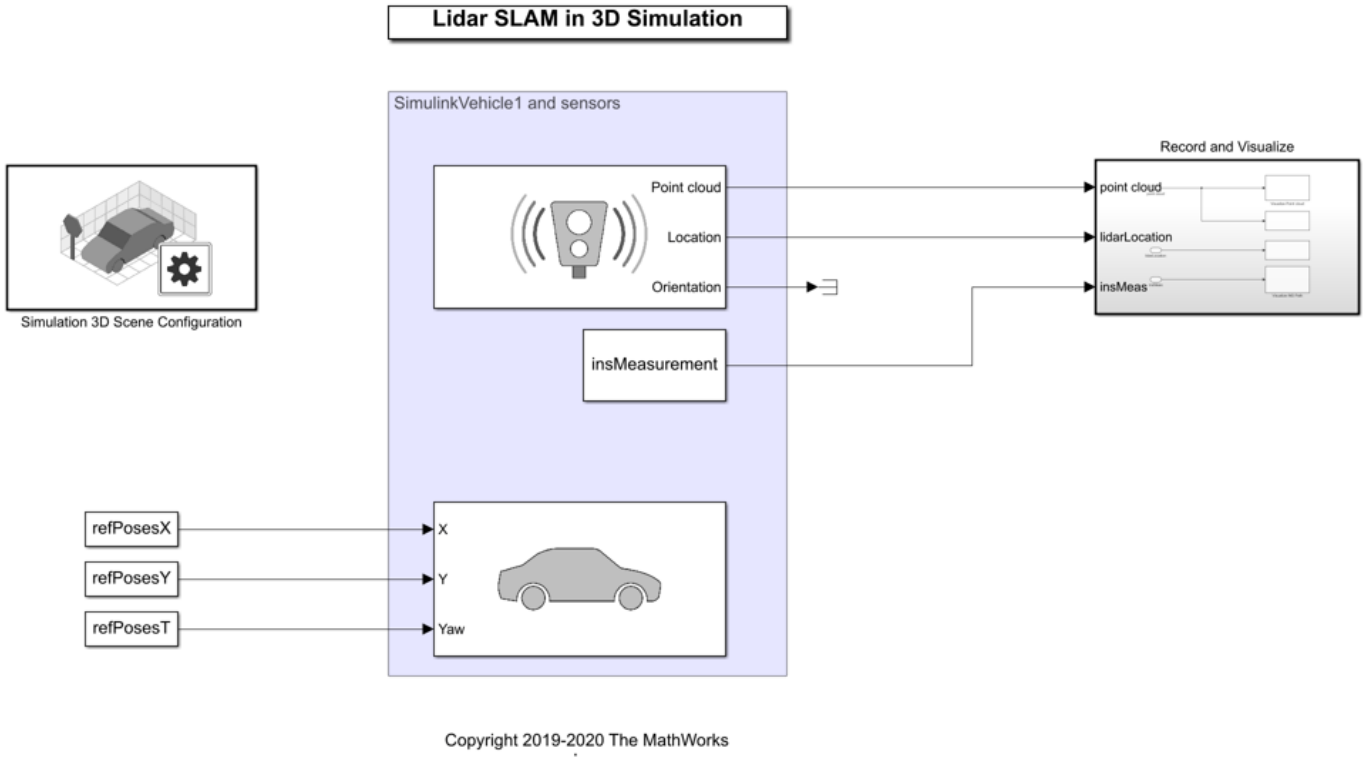
```
helperShowSceneImage(sceneName);
hold on
scatter(refPosesX(:,2), refPosesY(:,2), 7, 'filled')

% Adjust axes limits
xlim([-150 100])
ylim([-125 75])
```



The `LidarSLAMIn3DSimulation` Simulink model is configured with the US City Block (Automated Driving Toolbox) scene using the Simulation 3D Scene Configuration (Automated Driving Toolbox) block. The model places a vehicle on the scene using the Simulation 3D Vehicle with Ground Following (Automated Driving Toolbox) block. A lidar sensor is attached to the vehicle using the

Simulation 3D Lidar (Automated Driving Toolbox) block. In the block dialog box, use the **Mounting** tab to adjust the placement of the sensor. Use the **Parameters** tab to configure properties of the sensor to simulate different lidar sensors. In this example, the lidar is mounted on the center of the roof. The lidar sensor is configured to model a typical Velodyne® HDL-32E sensor.

```
close(hScene)

if ~ispc
    error(['3D Simulation is only supported on Microsoft', char(174), ' Windows', char(174), '.'])
end

% Open the model
modelName = 'LidarSLAMIn3DSimulation';
open_system(modelName);
snapnow;
```



The model records and visualizes the synthetic lidar data. The recorded data is available through the simulation output, and can be used for prototyping your algorithm in MATLAB. Additionally, the model uses a From Workspace (Simulink) block to load simulated measurements from an Inertial Navigation Sensor (INS). The INS data was obtained by using an `insSensor` (Automated Driving Toolbox) object, and saved in a MAT file.

The rest of the example follows these steps:

1. Simulate the model to record synthetic lidar data generated by the sensor and save it to the workspace.
2. Use the sensor data saved to the workspace to develop a perception algorithm in MATLAB. The perception algorithm builds a map of the surroundings using SLAM.

3 Visualize the results of the built map.

Record and Visualize Synthetic Lidar Sensor Data

The **Record and Visualize** subsystem records the synthetic lidar data to the workspace using a To Workspace (Simulink) block. The **Visualize Point Cloud** MATLAB Function block uses a `pcplayer` object to visualize the streaming point clouds. The **Visualize INS Path** MATLAB Function block visualizes the streaming INS data.

Simulate the model. The streaming point cloud display shows the synthetic lidar sensor data. The scene display shows the synthetic INS sensor data. Once the model has completed simulation, the `simOut` variable holds a structure with variables written to the workspace. The `helperGetPointCloud` function extracts the sensor data into an array of `pointCloud` objects. The `pointCloud` object is the fundamental data structure used to hold lidar data and perform point cloud processing in MATLAB. Additionally, INS data is loaded from a MAT file, which will later be used to develop the perception algorithm. The INS data was obtained using the `insSensor` (Automated Driving Toolbox) object. The INS data has been processed to contain [x, y, theta] poses in world coordinates.

```
% Update simulation stop time to end when reference path is completed
simStopTime = refPosesX(end,1);
set_param(gcs, 'StopTime', num2str(simStopTime));

% Load INS data from MAT file
data = load('insMeasurement.mat');
insData = data.insMeasurement.signals.values;

% Run the simulation
simOut = sim(modelName);

% Create a pointCloud array from the recorded data
ptCloudArr = helperGetPointCloud(simOut);
```

Use Recorded Data to Develop Perception Algorithm

The synthetic lidar sensor data can be used to develop, experiment with, and verify a perception algorithm in different scenarios. This example uses an algorithm to build a 3D map of the environment from streaming lidar data. Such an algorithm is a building block for applications like localization. It can also be used to create high-definition (HD) maps for geographic regions that can then be used for online localization. The map building algorithm is encapsulated in the `helperLidarMapBuilder` class. This class uses point cloud and lidar processing capabilities in MATLAB. For more details, see “Point Cloud Processing”.

The `helperLidarMapBuilder` class takes incoming point clouds from a lidar sensor and progressively builds a map using the following steps:

1 **Preprocess point cloud:** Preprocess each incoming point cloud to remove the ground plane and ego vehicle.

2 **Register point clouds:** Register the incoming point cloud with the last point cloud using a normal distribution transform (NDT) registration algorithm. The `pcregisterndt` function performs the registration. To improve accuracy and efficiency of registration, `pcdownsample` is used to downsample the point cloud prior to registration. An initial transform estimate can substantially improve registration performance. In this example, INS measurements are used to accomplish this.

3. **Register point clouds:** Use the estimated transformation obtained from registration to transform the incoming point cloud to the frame of reference of the map.
4. **Update view set:** Add the incoming point cloud and the estimated absolute pose as a view in a `pcviewset` object. Add a connection between the current and previous view with the relative transformation between them.

The `updateMap` method of the `helperLidarMapBuilder` class accomplishes these steps. The `helperEstimateRelativeTransformationFromINS` function computes an initial estimate for registration from simulated INS sensor readings.

Such an algorithm is susceptible to drift while accumulating a map over long sequences. To reduce the drift, it is typical to detect loop closures and use graph SLAM to correct the drift. See “Build a Map from Lidar Data Using SLAM” (Automated Driving Toolbox) example for a detailed treatment. The `configureLoopDetector` method of the `helperLidarMapBuilder` class configures loop closure detection. Once it is configured, loop closure detection takes place each time `updateMap` is invoked, using the following functions and classes:

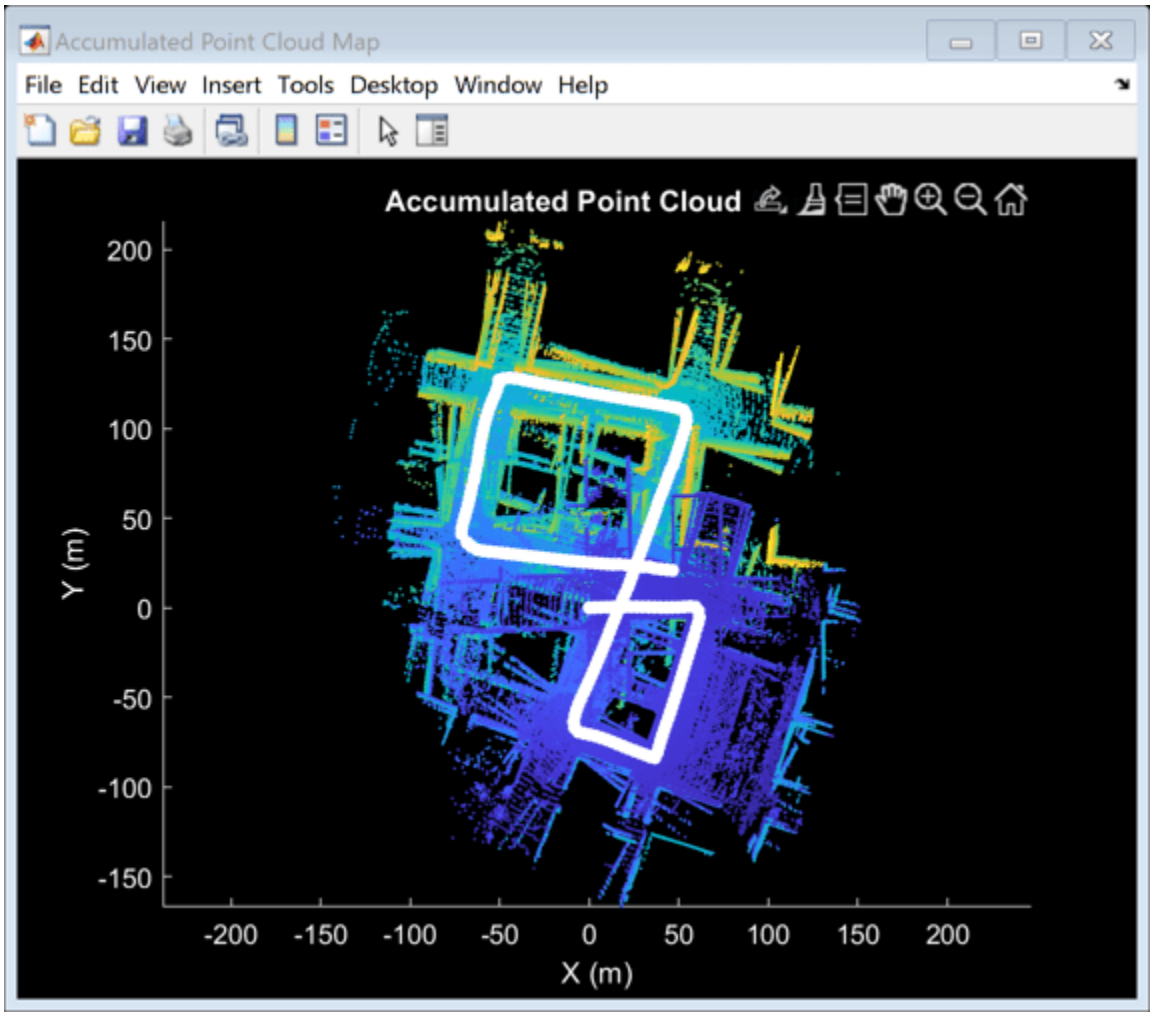
- `pcviewset`: Manages data associated with point cloud odometry like point clouds, poses and connections.
- `scanContextDescriptor`: Extracts scan context descriptors from each incoming point cloud. Scan context is a 2-D global feature descriptor that is used for loop closure detection.
- `scanContextLoopDetector`: Manages scan context descriptors and detects loop closures. It uses `scanContextDistance` to compute the distance between scan context descriptors and select the closest feature matches.

Then, the example uses point cloud registration to accept or reject loop closure candidates and to find the loop closure transformation.

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

% Create a lidar map builder
mapBuilder = helperLidarMapBuilder('DownsamplePercent', 0.25, ...
    'RegistrationGridStep', 3.5, 'Verbose', true);

% Configure the map builder to detect loop closures
configureLoopDetector(mapBuilder, ...
    'LoopConfirmationRMSE',  2, ...
    'SearchRadius',          0.15, ...
    'DistanceThreshold',     0.07);

% Loop through the point cloud array and progressively build a map
skipFrames = 5;
numFrames  = numel(ptCloudArr);
exitLoop   = false;

prevInsMeas = insData(1, :);
for n = 1 : skipFrames : numFrames

    insMeas = insData(n, :);

    % Estimate initial transformation using INS
    initTform = helperEstimateRelativeTransformationFromINS(insMeas, prevInsMeas);

    % Update map with new lidar frame
```

```
    updateMap(mapBuilder, ptCloudArr(n), initTform);

    % Update top-view display
    isDisplayOpen = updateDisplay(mapBuilder, exitLoop);

    % Check and exit if needed
    exitLoop = ~isDisplayOpen;

    prevInsMeas = insMeas;
end

snapnow;

% Close display
closeDisplay = true;
updateDisplay(mapBuilder, closeDisplay);
```

```
Loop closure candidate found between view Id 211 and 4 with RMSE 1.135051...
Accepted
```



The accumulated drift progressively increases over time resulting in an unusable map.

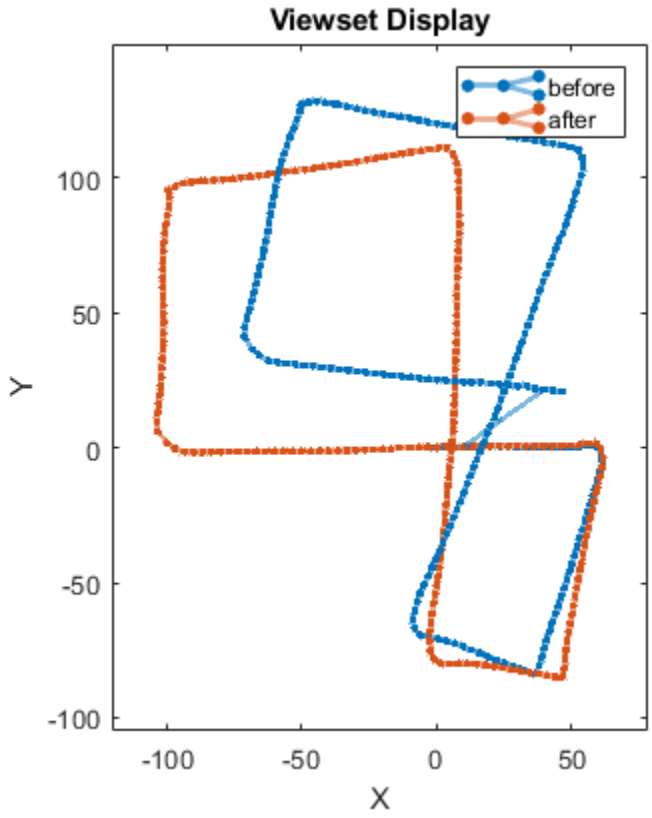
Once sufficient loop closures are detected, the accumulated drift can be corrected using pose graph optimization. This is accomplished by the `optimizeMapPoses` method of the `helperLidarMapBuilder` class, which uses `createPoseGraph` to create a pose graph and `optimizePoseGraph` (Navigation Toolbox) to optimize the pose graph.

After the pose graph has been optimized, rebuild the map using the updated poses. This is accomplished by the `rebuildMap` method of `helperLidarMapBuilder` using `pcalign`.

Use `optimizeMapPoses` and `rebuildMap` to correct for the drift and rebuild the map. Visualize the view set before and after pose graph optimization.

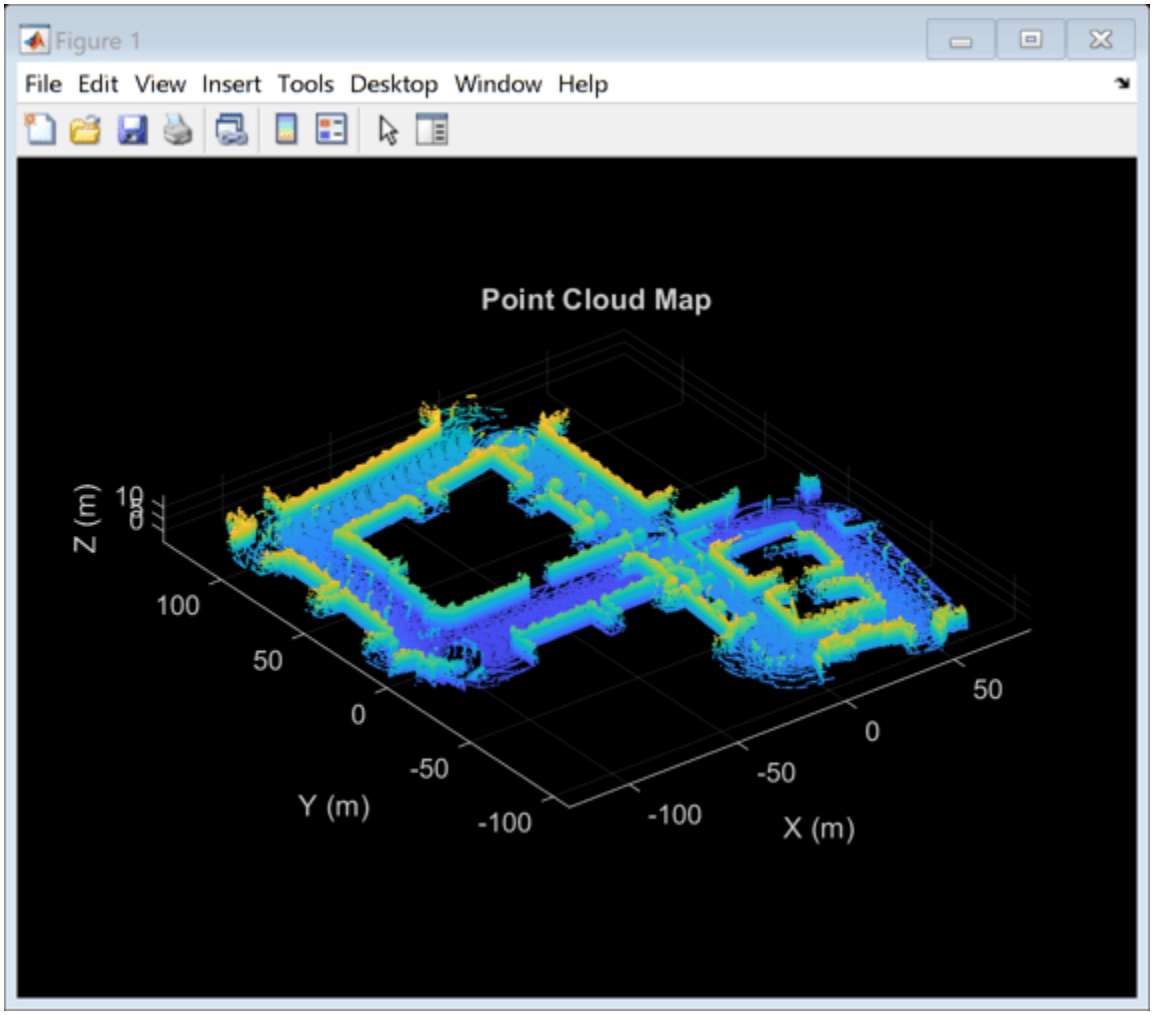
```
% Visualize viewset before pose graph optimization
hFigViewset = figure;
hG = plot(mapBuilder.ViewSet);
view(hG.Parent, 2);
title('Viewset Display')

% Optimize pose graph and rebuild map
optimizeMapPoses(mapBuilder);
rebuildMap(mapBuilder);

% Overlay viewset after pose graph optimization
hold(hG.Parent, 'on');
plot(mapBuilder.ViewSet);
hold(hG.Parent, 'off');

legend(hG.Parent, 'before', 'after')
```

```
Optimizing pose graph...done
Rebuilding map...done
```



Visualize the accumulated point cloud map computed using the recorded data.

```
close(hFigViewset)

hFigMap = figure;
pcshow(mapBuilder.Map)

% Customize axes labels and title
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Z (m)')
title('Point Cloud Map')

helperMakeFigurePublishFriendly(hFigMap);
```



By changing the scene, placing more vehicles in the scene, or updating the sensor mounting and parameters, the perception algorithm can be stress-tested under different scenarios. This approach can be used to increase coverage for scenarios that are difficult to reproduce in the real world.

```
% Close windows
close(hFigMap)
close_system(modelName)
```

Supporting Functions

helperGetPointCloud Extract an array of `pointCloud` objects.

```
function ptCloudArr = helperGetPointCloud(simOut)

% Extract signal
ptCloudData = simOut.ptCloudData.signals.values;

% Create a pointCloud array
ptCloudArr = pointCloud(ptCloudData(:,:,:,1));

for n = 2 : size(ptCloudData,4)
    ptCloudArr(end+1) = pointCloud(ptCloudData(:,:,:,n));  %#ok<AGROW>
```

```
end
end
```

helperMakeFigurePublishFriendly Adjust figure so that screenshot captured by publish is correct.

```
function helperMakeFigurePublishFriendly(hFig)

if ~isempty(hFig) && isvalid(hFig)
    hFig.HandleVisibility = 'callback';
end
end
```

Additional supporting functions or classes used in the example are included below.

helperLidarMapBuilder progressively builds a lidar map using point cloud scans. Each point cloud is processed to remove the ground plane and the ego vehicle, and registered against the previous point cloud. A point cloud map is then progressively built by aligning and merging the point clouds.

helperEstimateRelativeTransformationFromINS estimates a relative transformation from INS data.

helperShowSceneImage displays top-view image of the Unreal scene.

helperUpdatePolyline updates a polyline position used in conjunction with helperShowSceneImage.

Ground Plane and Obstacle Detection Using Lidar

This example shows how to process 3-D lidar data from a sensor mounted on a vehicle by segmenting the ground plane and finding nearby obstacles. This can facilitate drivable path planning for vehicle navigation. The example also shows how to visualize streaming lidar data.

Create a Velodyne File Reader

The lidar data used in this example was recorded using a Velodyne® HDL32E sensor mounted on a vehicle. Set up a `velodyneFileReader` object to read the recorded PCAP file.

```
fileName    = 'lidarData_ConstructionRoad.pcap';
deviceModel = 'HDL32E';

veloReader = velodyneFileReader(fileName, deviceModel);
```

Read a Lidar Scan

Each scan of lidar data is stored as a 3-D point cloud. Efficiently processing this data using fast indexing and search is key to the performance of the sensor processing pipeline. This efficiency is achieved using the `pointCloud` object, which internally organizes the data using a K-d tree data structure.

The `veloReader` constructs an organized `pointCloud` for each lidar scan. The `Location` property of the `pointCloud` is an M-by-N-by-3 matrix, containing the XYZ coordinates of points in meters. The point intensities are stored in `Intensity`.

```
% Read a scan of lidar data
ptCloud = readFrame(veloReader) %#ok<NOPTS>
```

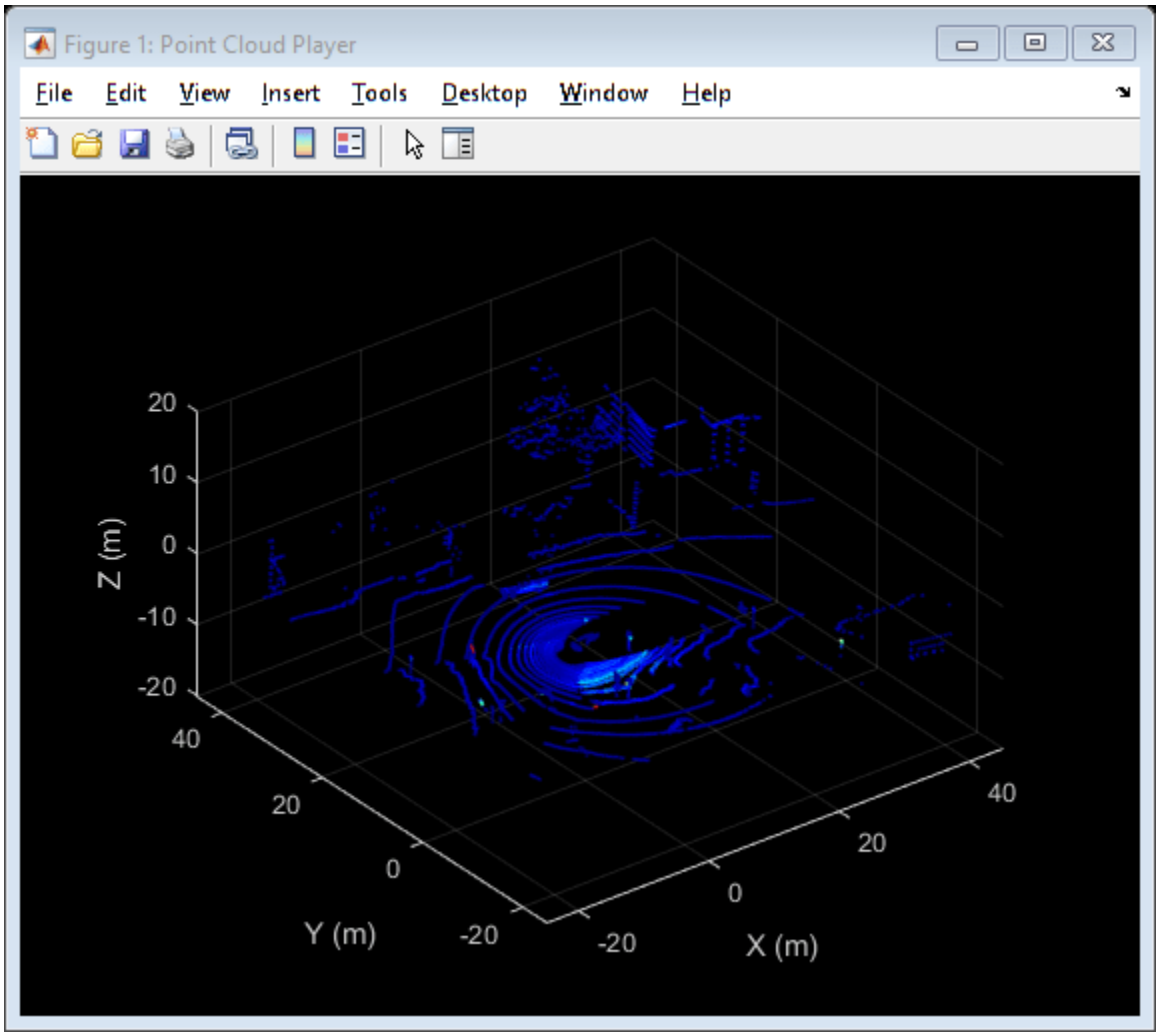
```
ptCloud =

  pointCloud with properties:

     Location: [32×1083×3 single]
        Count: 34656
      XLimits: [-80.0444 87.1780]
      YLimits: [-85.6287 92.8721]
      ZLimits: [-21.6060 14.3558]
        Color: []
       Normal: []
    Intensity: [32×1083 uint8]
```

Setup Streaming Point Cloud Display

The `pcplayer` can be used to visualize streaming point cloud data. Set up the region around the vehicle to display by configuring `pcplayer`.

```
% Specify limits of point cloud display
xlimits = [-25 45]; % meters
ylimits = [-25 45];
zlimits = [-20 20];

% Create a pcplayer
lidarViewer = pcplayer(xlimits, ylimits, zlimits);
```

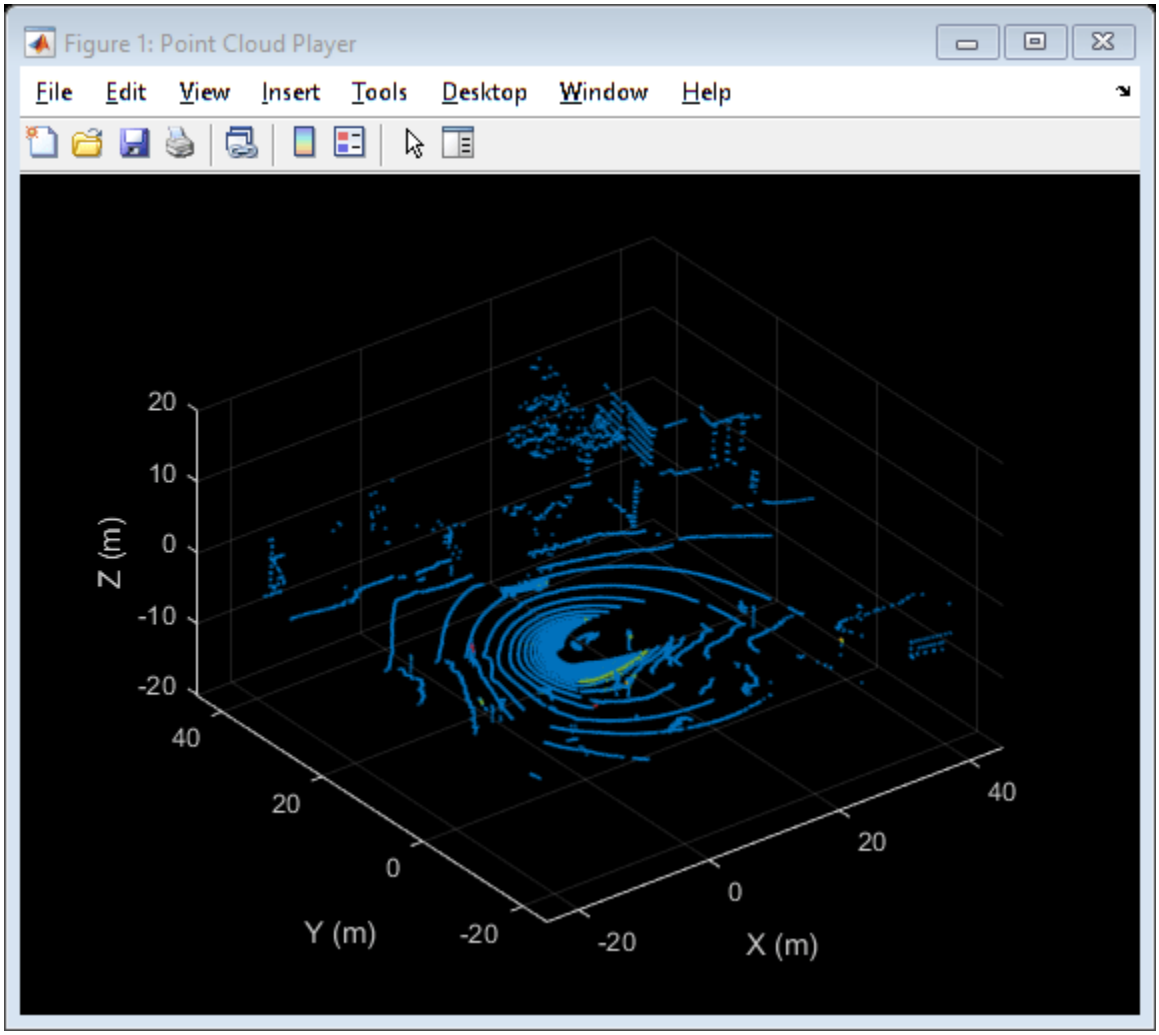
```
% Customize player axes labels
xlabel(lidarViewer.Axes, 'X (m)')
ylabel(lidarViewer.Axes, 'Y (m)')
zlabel(lidarViewer.Axes, 'Z (m)')

% Display the raw lidar scan
view(lidarViewer, ptCloud)
```



In this example, we will be segmenting points belonging to the ground plane, the ego vehicle and nearby obstacles. Set the colormap for labeling these points.

```
% Define labels to use for segmented points
colorLabels = [...
    0      0.4470 0.7410; ... % Unlabeled points, specified as [R,G,B]
    0.4660 0.6740 0.1880; ... % Ground points
    0.9290 0.6940 0.1250; ... % Ego points
    0.6350 0.0780 0.1840];    % Obstacle points

% Define indices for each label
colors.Unlabeled = 1;
```

```
colors.Ground    = 2;
colors.Ego       = 3;
colors.Obstacle  = 4;

% Set the colormap
colormap(lidarViewer.Axes, colorLabels)
```



Segment the Ego Vehicle

The lidar is mounted on top of the vehicle, and the point cloud may contain points belonging to the vehicle itself, such as on the roof or hood. Knowing the dimensions of the vehicle, we can segment out points that are closest to the vehicle.

Create a `vehicleDimensions (Automated Driving Toolbox)` object for storing dimensions of the vehicle.

```
vehicleDims = vehicleDimensions(); % Typical vehicle 4.7m by 1.8m by 1.4m
```

Specify the mounting location of the lidar in the vehicle coordinate system. The vehicle coordinate system is centered at the center of the rear-axle, on the ground, with positive X direction pointing

forward, positive Y towards the left, and positive Z upwards. In this example, the lidar is mounted on the top center of the vehicle, parallel to the ground.

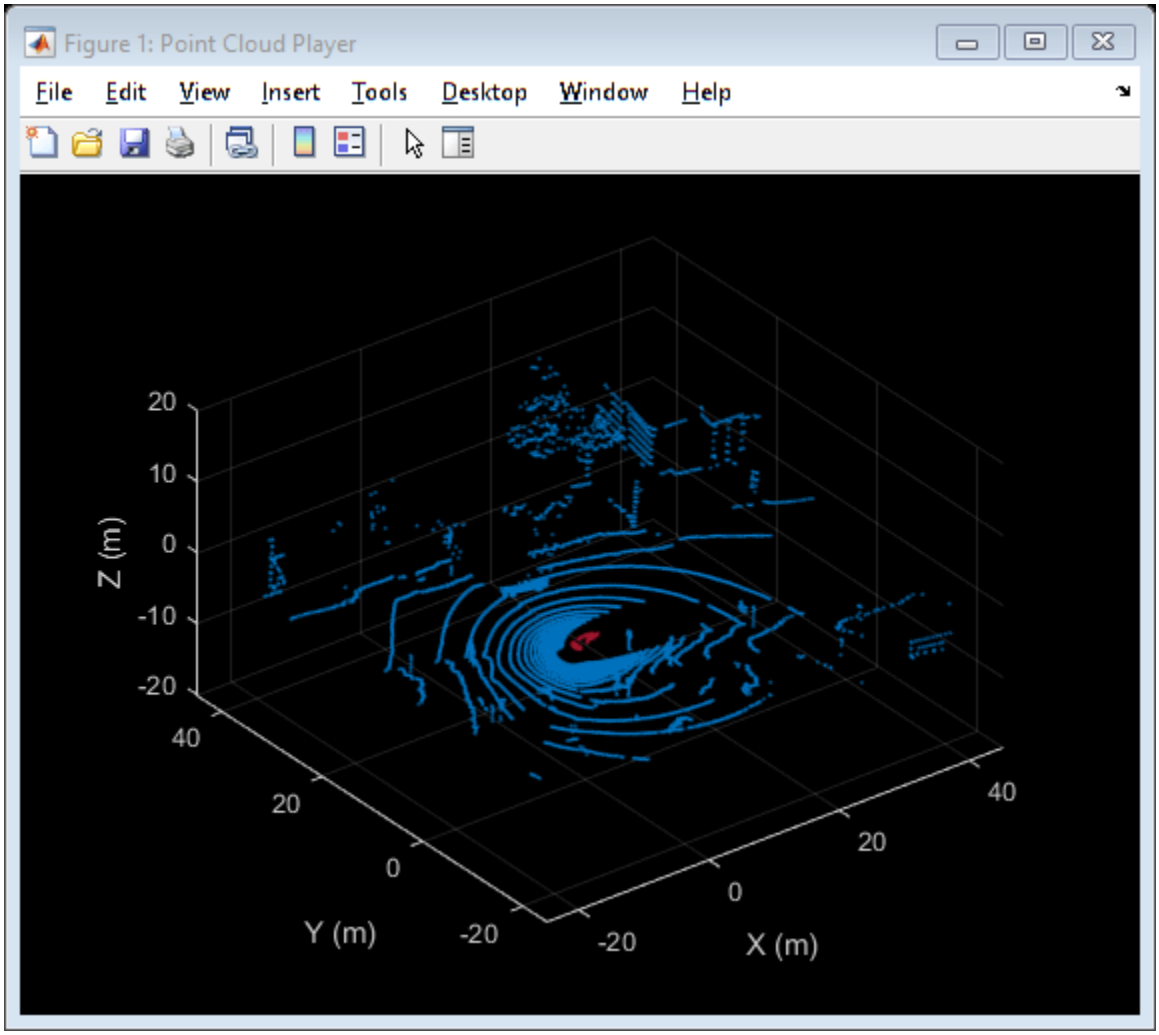
```
mountLocation = [...
    vehicleDims.Length/2 - vehicleDims.RearOverhang, ... % x
    0, ...                                               % y
    vehicleDims.Height];                                 % z
```

Segment the ego vehicle using the helper function `helperSegmentEgoFromLidarData`. This function segments all points within the cuboid defined by the ego vehicle. Store the segmented points in a struct `points`.

```
points = struct();
points.EgoPoints = helperSegmentEgoFromLidarData(ptCloud, vehicleDims, mountLocation);
```

Visualize the point cloud with segmented ego vehicle. Use the `helperUpdateView` helper function.

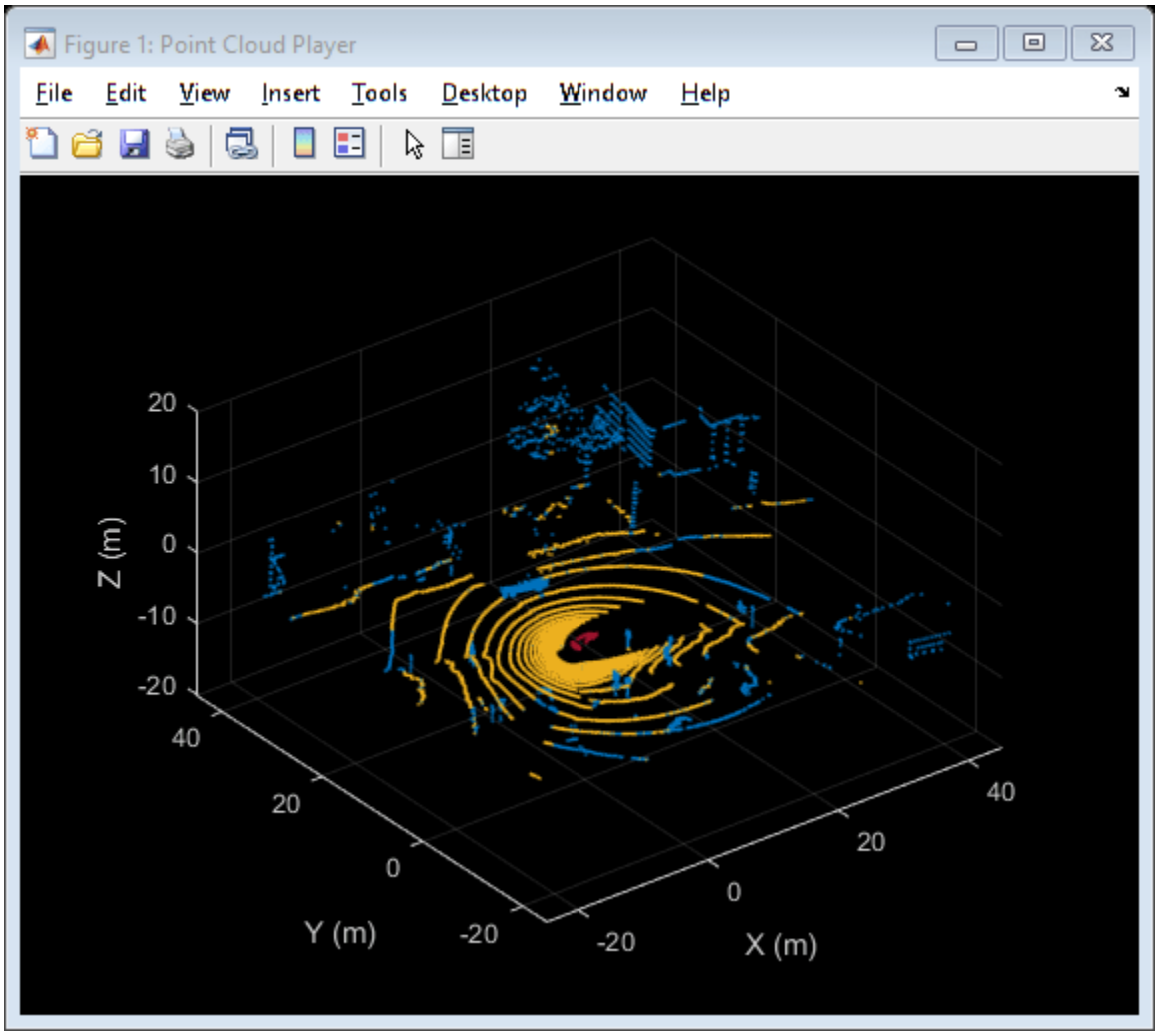
```
closePlayer = false;
helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
```



Segment Ground Plane and Nearby Obstacles

In order to identify obstacles from the lidar data, first segment the ground plane using the `segmentGroundFromLidarData` function to accomplish this. This function segments points belonging to ground from organized lidar data.

```
elevationDelta = 10;
points.GroundPoints = segmentGroundFromLidarData(ptCloud, 'ElevationAngleDelta', elevationDelta);

% Visualize the segmented ground plane.
helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
```

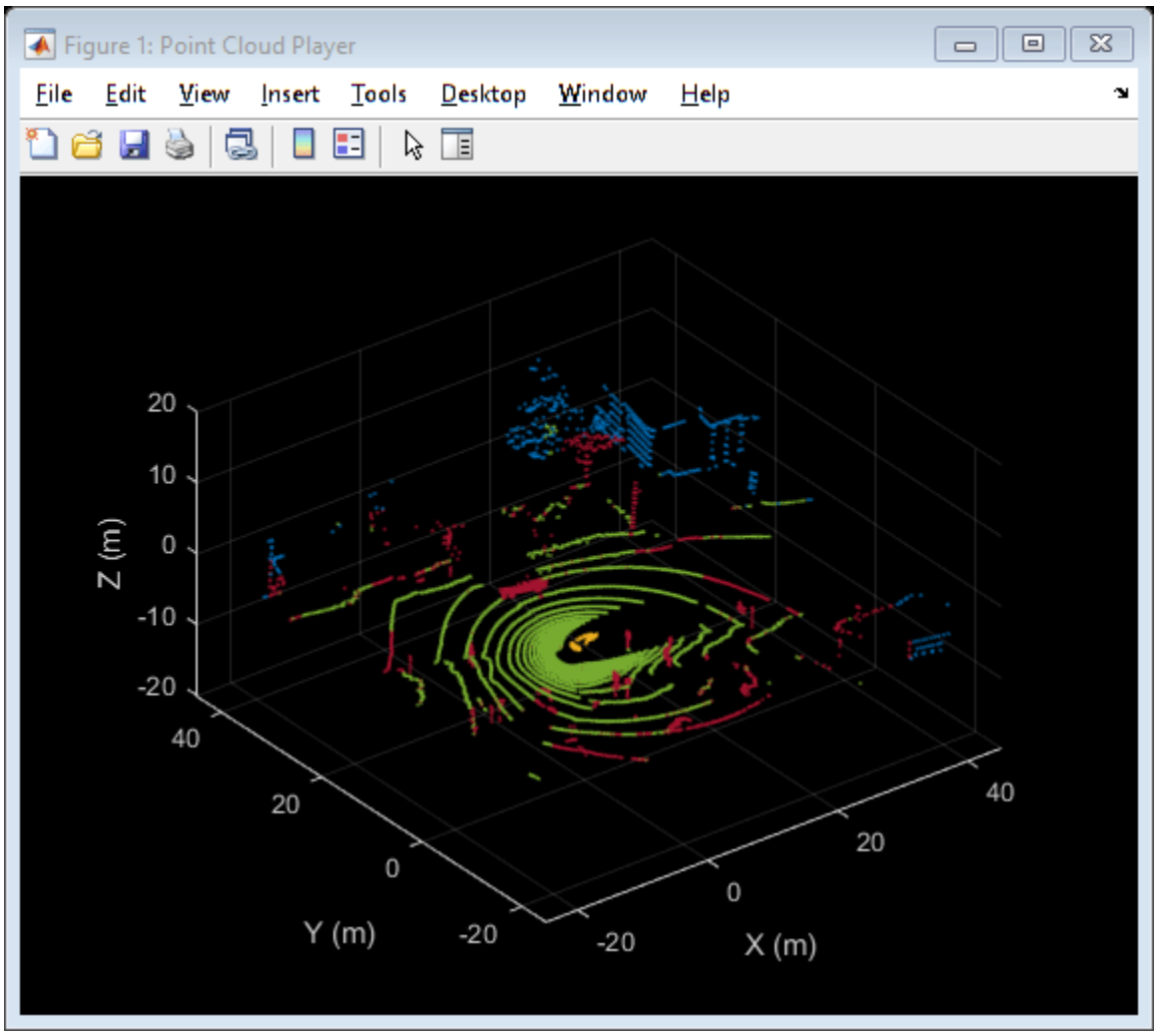


Remove points belonging to the ego vehicle and the ground plane by using the `select` function on the point cloud. Specify the `'OutputSize'` as `'full'` to retain the organized nature of the point cloud.

```
nonEgoGroundPoints = ~points.EgoPoints & ~points.GroundPoints;
ptCloudSegmented = select(ptCloud, nonEgoGroundPoints, 'OutputSize', 'full');
```

Next, segment nearby obstacles by looking for all points that are not part of the ground or ego vehicle within some radius from the ego vehicle. This radius can be determined based on the range of the lidar and area of interest for further processing.

```
sensorLocation  = [0, 0, 0]; % Sensor is at the center of the coordinate system
radius          = 40; % meters

points.ObstaclePoints = findNeighborsInRadius(ptCloudSegmented, ...
    sensorLocation, radius);

% Visualize the segmented obstacles
helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
```



Process Lidar Sequence

Now that the point cloud processing pipeline for a single lidar scan has been laid out, put this all together to process 30 seconds from the sequence of recorded data. The code below is shortened since the key parameters have been defined in the previous steps. Here, the parameters are used without further explanation.

```
% Rewind the |veloReader| to start from the beginning of the sequence
reset(veloReader);

% Stop processing after 30 seconds
stopTime = veloReader.StartTime + seconds(30);

isPlayerOpen = true;
while hasFrame(veloReader) && veloReader.CurrentTime < stopTime && isPlayerOpen

    % Grab the next lidar scan
    ptCloud = readFrame(veloReader);

    % Segment points belonging to the ego vehicle
    points.EgoPoints = helperSegmentEgoFromLidarData(ptCloud, vehicleDims, mountLocation);

    % Segment points belonging to the ground plane
    points.GroundPoints = segmentGroundFromLidarData(ptCloud, 'ElevationAngleDelta', elevationDe

    % Remove points belonging to the ego vehicle and ground plane
    nonEgoGroundPoints = ~points.EgoPoints & ~points.GroundPoints;
    ptCloudSegmented = select(ptCloud, nonEgoGroundPoints, 'OutputSize', 'full');

    % Segment obstacles
    points.ObstaclePoints = findNeighborsInRadius(ptCloudSegmented, sensorLocation, radius);

    closePlayer = ~hasFrame(veloReader);

    % Update lidar display
    isPlayerOpen = helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
end
snapnow
```

Supporting Functions

`helperSegmentEgoFromLidarData` segments points belonging to the ego vehicle given the dimensions of the vehicle and mounting location.

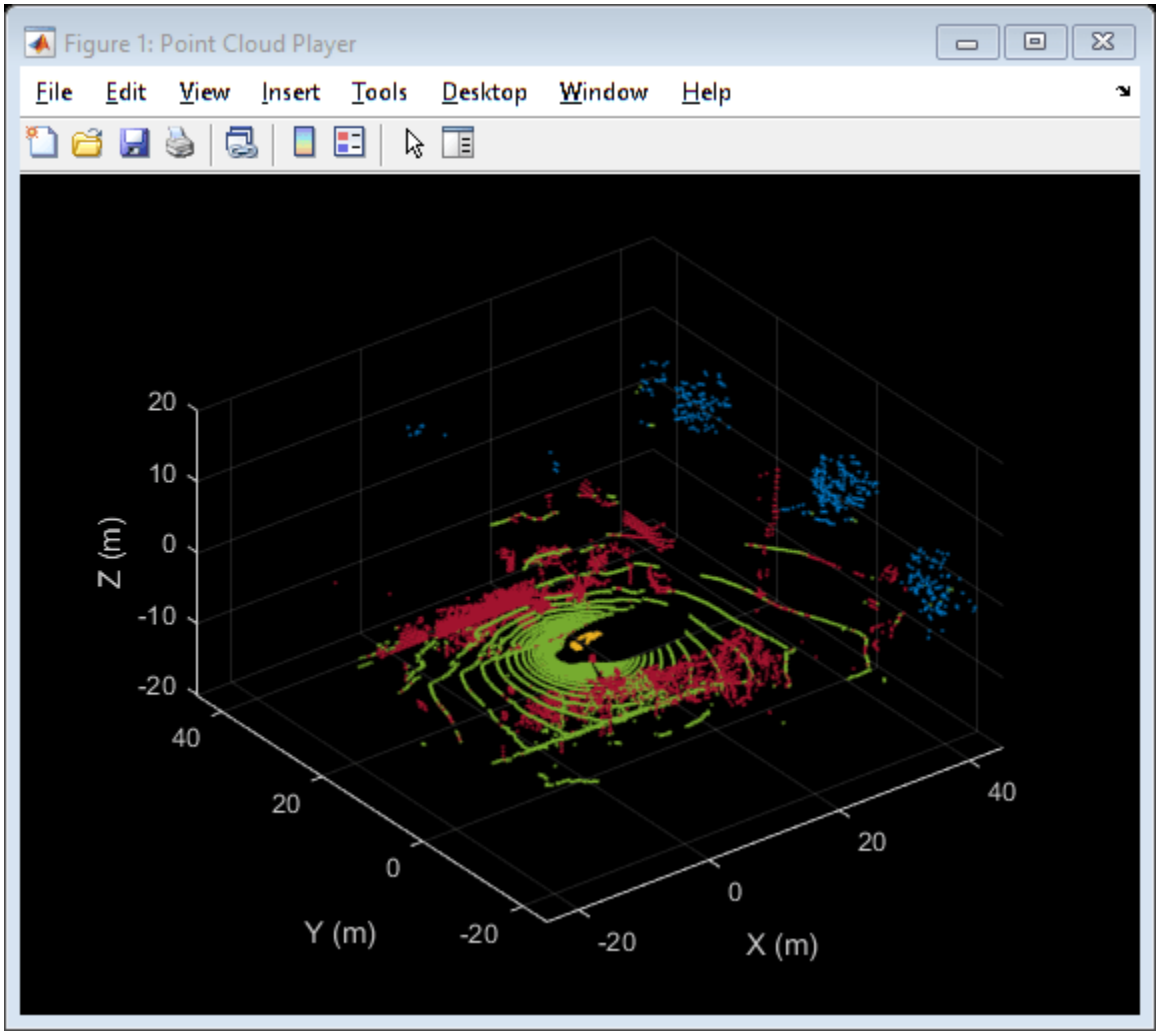
```
function egoPoints = helperSegmentEgoFromLidarData(ptCloud, vehicleDims, mountLocation)
%helperSegmentEgoFromLidarData segment ego vehicle points from lidar data
%   egoPoints = helperSegmentEgoFromLidarData(ptCloud,vehicleDims,mountLocation)
%   segments points belonging to the ego vehicle of dimensions vehicleDims
%   from the lidar scan ptCloud. The lidar is mounted at location specified
%   by mountLocation in the vehicle coordinate system. ptCloud is a
%   pointCloud object. vehicleDimensions is a vehicleDimensions object.
%   mountLocation is a 3-element vector specifying XYZ location of the
%   lidar in the vehicle coordinate system.
%
%   This function assumes that the lidar is mounted parallel to the ground
%   plane, with positive X direction pointing ahead of the vehicle,
%   positive Y direction pointing to the left of the vehicle in a
%   right-handed system.

% Buffer around ego vehicle
bufferZone = [0.1, 0.1, 0.1]; % in meters

% Define ego vehicle limits in vehicle coordinates
egoXMin = -vehicleDims.RearOverhang - bufferZone(1);
egoXMax = egoXMin + vehicleDims.Length + bufferZone(1);
```

```
egoYMin = -vehicleDims.Width/2 - bufferZone(2);
egoYMax = egoYMin + vehicleDims.Width + bufferZone(2);
egoZMin = 0 - bufferZone(3);
egoZMax = egoZMin + vehicleDims.Height + bufferZone(3);

egoXLimits = [egoXMin, egoXMax];
egoYLimits = [egoYMin, egoYMax];
egoZLimits = [egoZMin, egoZMax];

% Transform to lidar coordinates
egoXLimits = egoXLimits - mountLocation(1);
egoYLimits = egoYLimits - mountLocation(2);
egoZLimits = egoZLimits - mountLocation(3);

% Use logical indexing to select points inside ego vehicle cube
egoPoints = ptCloud.Location(:,:,1) > egoXLimits(1) ...
    & ptCloud.Location(:,:,1) < egoXLimits(2) ...
    & ptCloud.Location(:,:,2) > egoYLimits(1) ...
    & ptCloud.Location(:,:,2) < egoYLimits(2) ...
    & ptCloud.Location(:,:,3) > egoZLimits(1) ...
    & ptCloud.Location(:,:,3) < egoZLimits(2);
end
```

`helperUpdateView` updates the streaming point cloud display with the latest point cloud and associated color labels.

```
function isPlayerOpen = helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer)
%helperUpdateView update streaming point cloud display
%   isPlayerOpen = helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayers)
%   updates the pcplayer object specified in lidarViewer with a new point
%   cloud ptCloud. Points specified in the struct points are colored
%   according to the colormap of lidarViewer using the labels specified by
%   the struct colors. closePlayer is a flag indicating whether to close
%   the lidarViewer.

if closePlayer
    hide(lidarViewer);
    isPlayerOpen = false;
    return;
end

scanSize = size(ptCloud.Location);
scanSize = scanSize(1:2);

% Initialize colormap
colormapValues = ones(scanSize, 'like', ptCloud.Location) * colors.Unlabeled;

if isfield(points, 'GroundPoints')
    colormapValues(points.GroundPoints) = colors.Ground;
end

if isfield(points, 'EgoPoints')
    colormapValues(points.EgoPoints) = colors.Ego;
end

if isfield(points, 'ObstaclePoints')
    colormapValues(points.ObstaclePoints) = colors.Obstacle;
end
```

```
% Update view
view(lidarViewer, ptCloud.Location, colormapValues)

% Check if player is open
isPlayerOpen = isOpen(lidarViewer);

end
```



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.

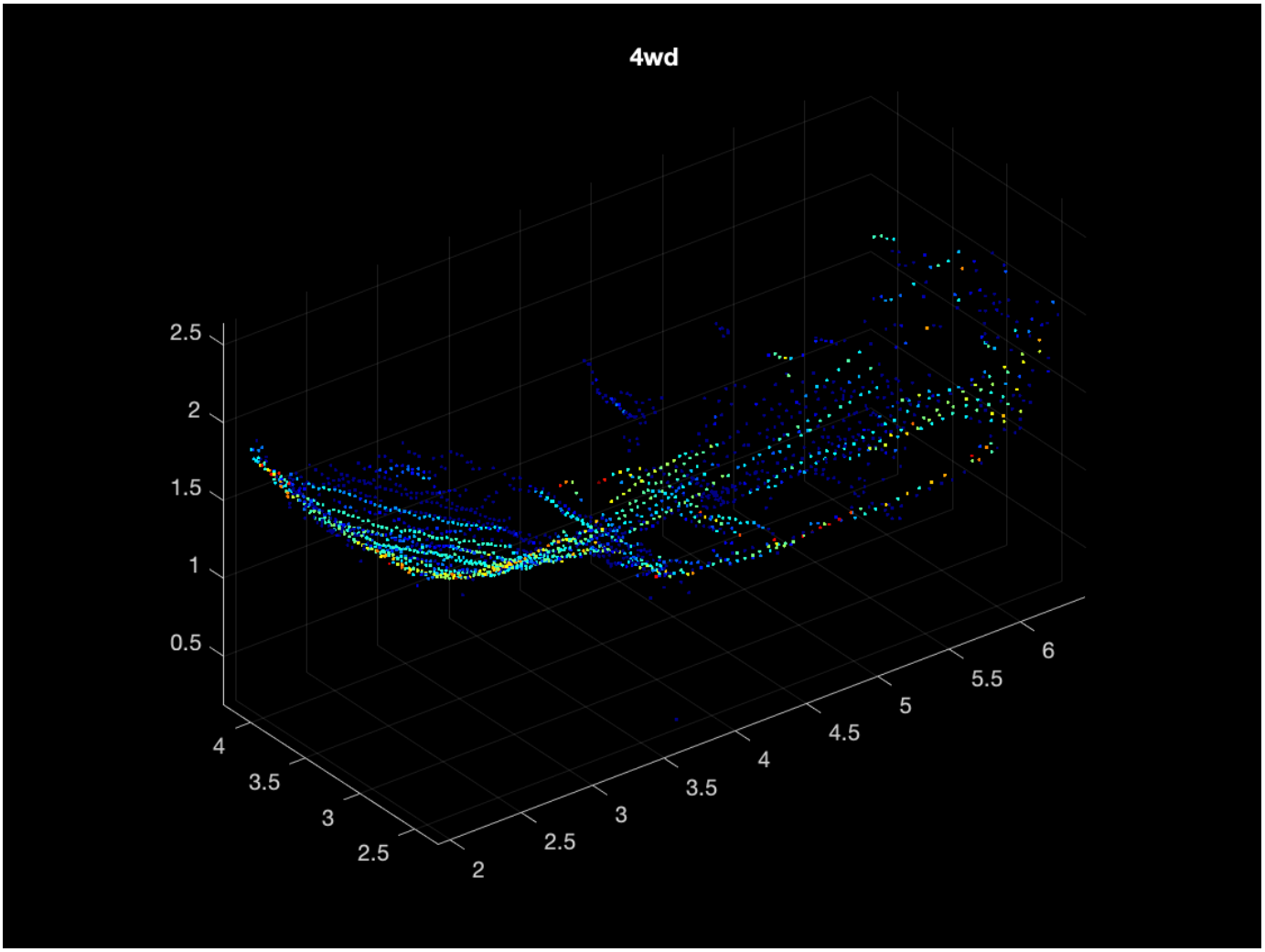
Import point cloud data

```
dataPath = downloadSydneyUrbanObjects(tempdir);
dsTrain = loadSydneyUrbanObjectsData(dataPath);
dataOut = preview(dsTrain)
```

```
dataOut=1×2 cell array
    {1×1 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);
```

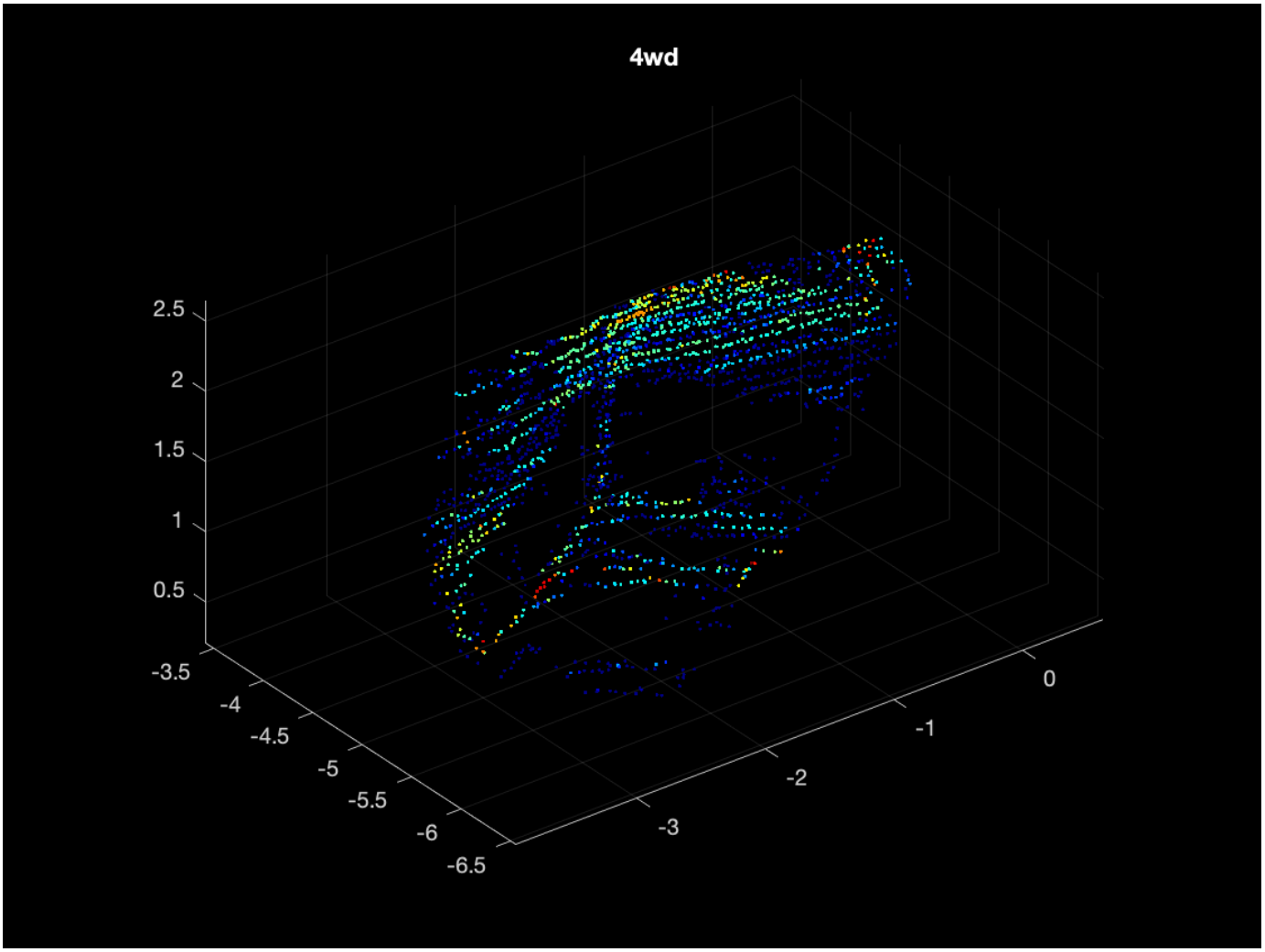
The `augmentPointCloud` function shown below, applies randomized rotation, homogenous scale, randomized reflection across the x- and y-axes, and randomized per point jitter to each observation using the `randomAffine3d` function to construct randomized affine transformations and the `pctransform` function to apply these transformations to each input point cloud.

```
dataOut = preview(dsAugmented)
```

```
dataOut=1×2 cell array
    {1×1 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(url+name,dataLoc);
end

datasetPath = dataLoc.append('sydney-urban-objects-dataset');

end

function ds = loadSydneyUrbanObjectsData(datapath,folds)
% loadSydneyUrbanObjectsData Datastore with point clouds and
% associated categorical labels for Sydney Urban Objects dataset.
%
% ds = loadSydneyUrbanObjectsData(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 = loadSydneyUrbanObjectsData(___,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

Create a datastore to load point cloud data from Sydney Urban Objects, along with associated object labels.

```
ds = loadSydneyUrbanObjectsData(sydneyUrbanObjectsPath);
```

Read and display the first observation from the Datastore.

```
data = preview(ds)
```

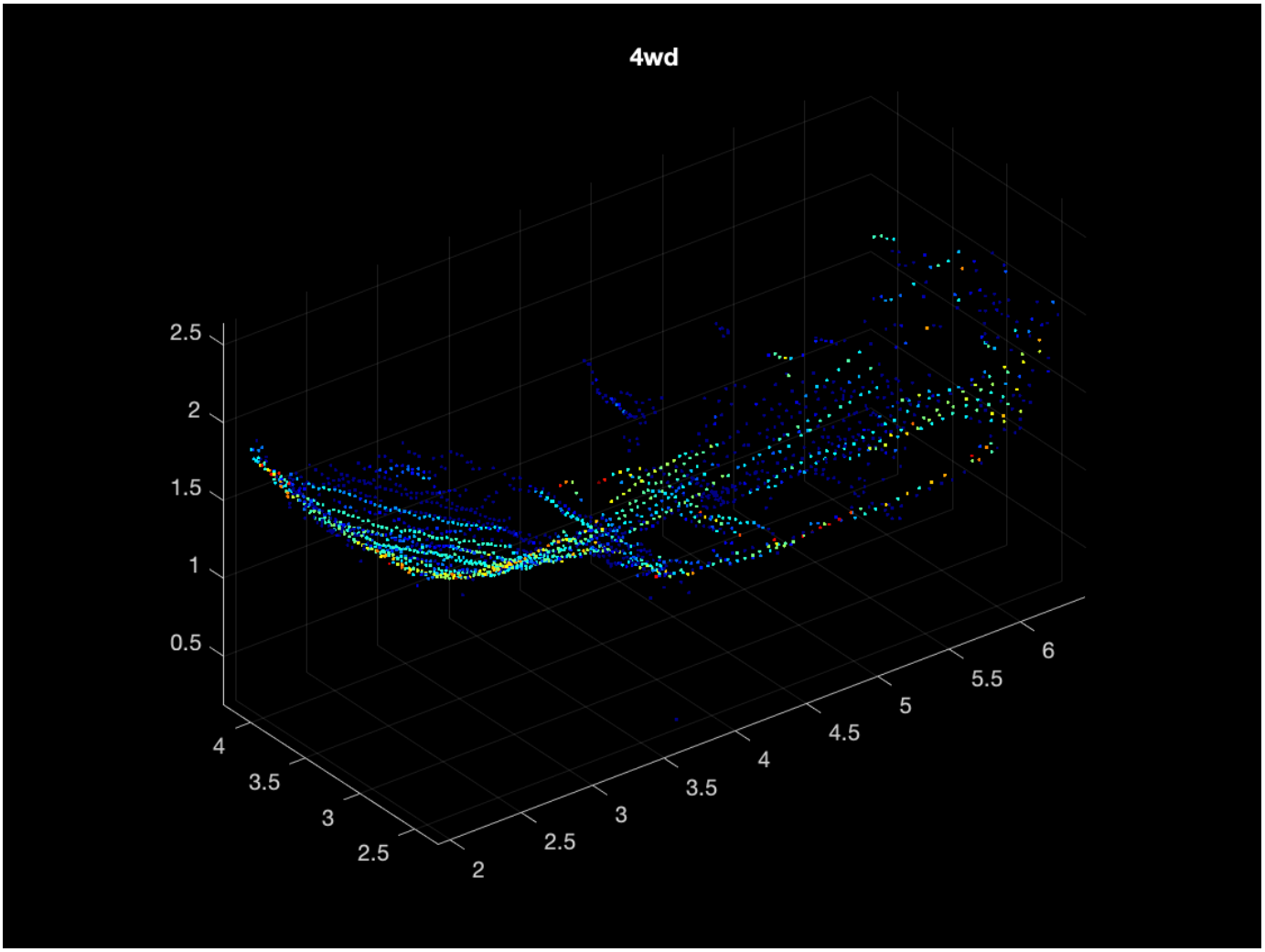
```
data=1×2 cell array
    {1×1 pointCloud}    {[4wd]}
```

```
disp(data)
```

```
    {1×1 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 Sydney 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(url+name,dataLoc);
end

datasetPath = dataLoc.append('sydney-urban-objects-dataset');

end

function ds = loadSydneyUrbanObjectsData(datapath,folds)
% loadSydneyUrbanObjectsData Create datastore with point clouds and
% associated categorical labels for Sydney Urban Objects dataset.
%
% ds = loadSydneyUrbanObjectsData(datapath) returns 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 = loadSydneyUrbanObjectsData(___,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 = loadSydneyUrbanObjectsData(dataPath);
```

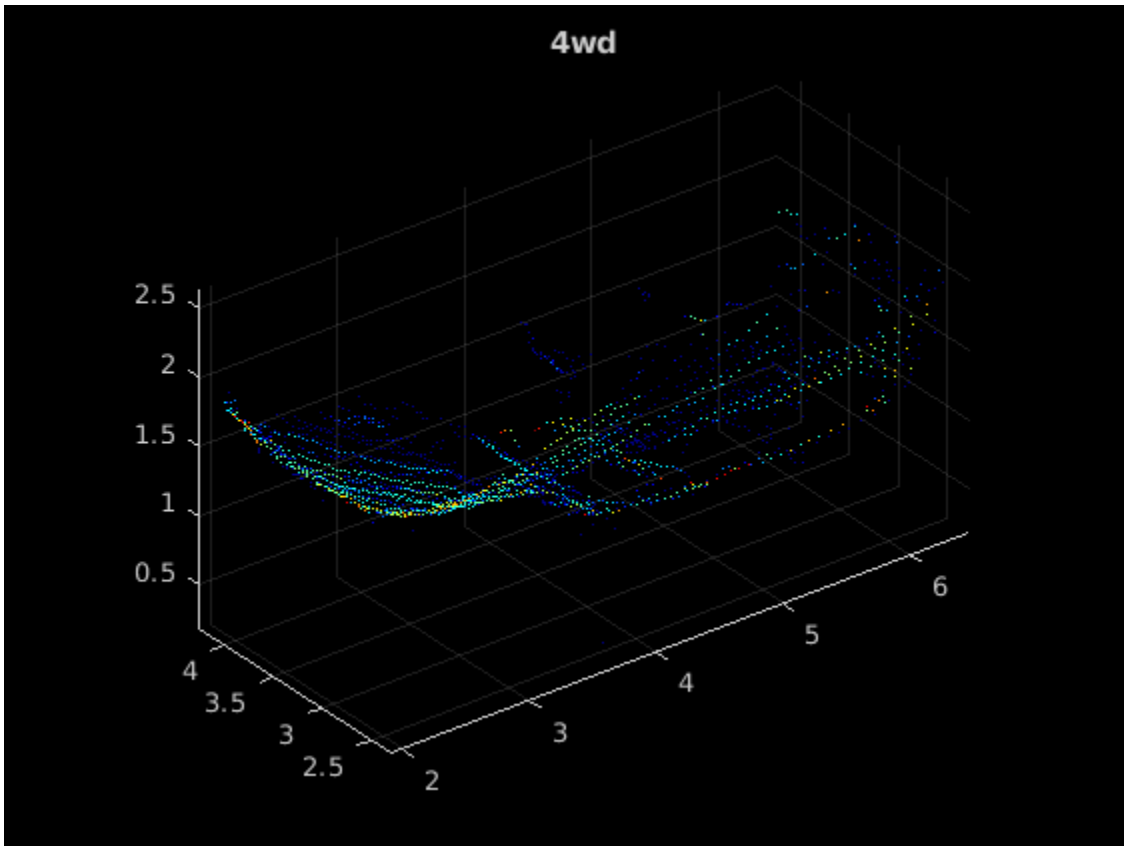
Obtain sample output data from datastore.

```
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`.

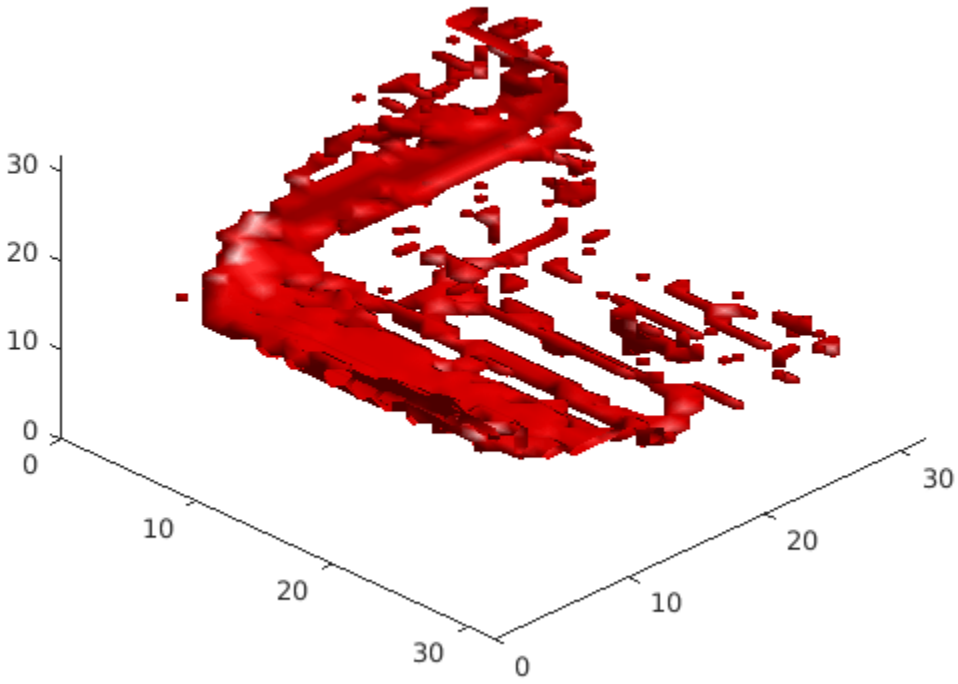
```
outputGridSize = [32,32,32];
bins = pcbin(data{1},outputGridSize);
```

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(url + name,dataLoc);
end

datasetPath = dataLoc.append('sydney-urban-objects-dataset');

end

function ds = loadSydneyUrbanObjectsData(datapath,folds)
% loadSydneyUrbanObjectsData Datastore with point clouds and
% associated categorical labels for Sydney Urban Objects dataset.
%
% ds = loadSydneyUrbanObjectsData(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 = loadSydneyUrbanObjectsData(___,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 = pcbin(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);
```

```
    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 =

  8×1 timetable

        Time          Orientation
    ______________    _____________

    23:46:11.4570    1×1 quaternion
    23:46:11.4605    1×1 quaternion
    23:46:11.4620    1×1 quaternion
    23:46:11.4655    1×1 quaternion
    23:46:11.4670    1×1 quaternion
    23:46:11.4705    1×1 quaternion
    23:46:11.4720    1×1 quaternion
    23:46:11.4755    1×1 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
```

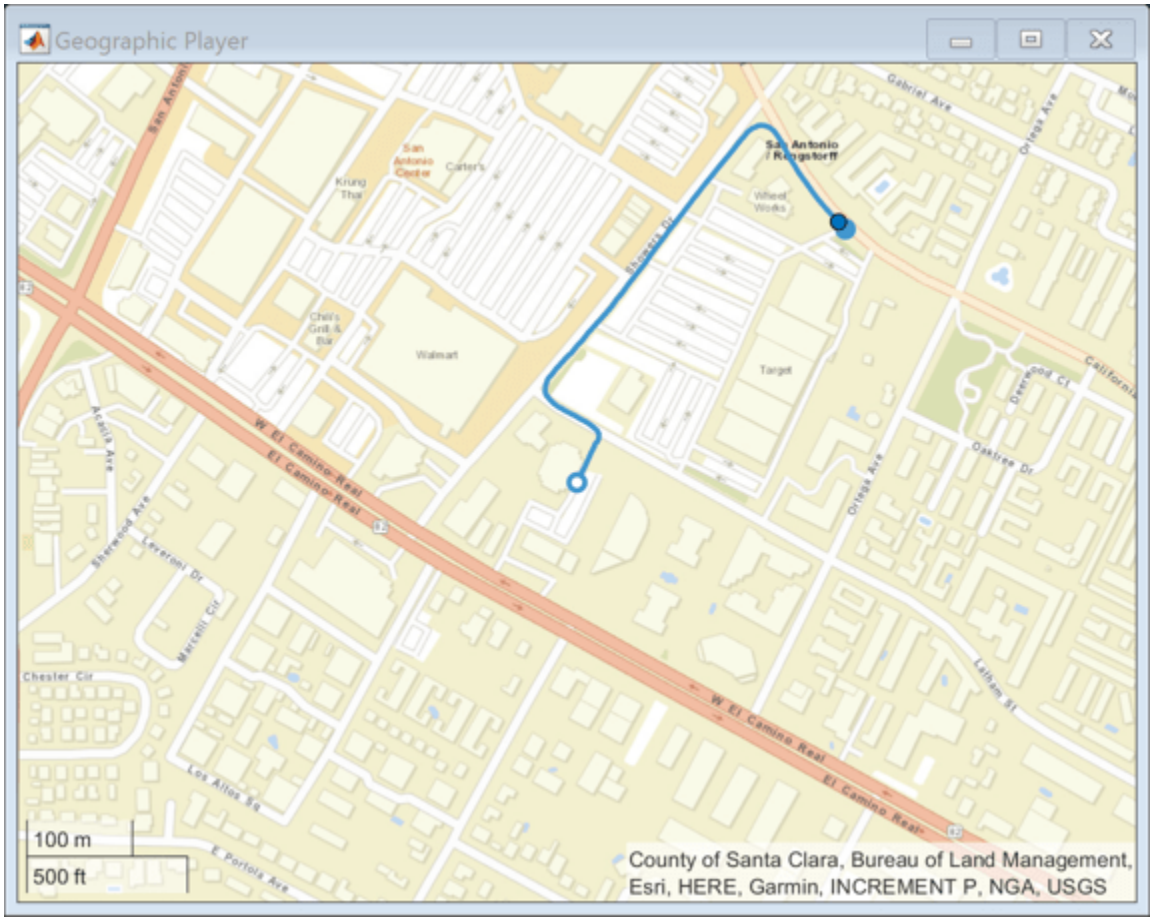
The GPS sensor is the slowest, running at a rate close to 1 Hz. The lidar is next slowest, running at a rate close to 10 Hz, followed by the IMU at a rate of almost 400 Hz.

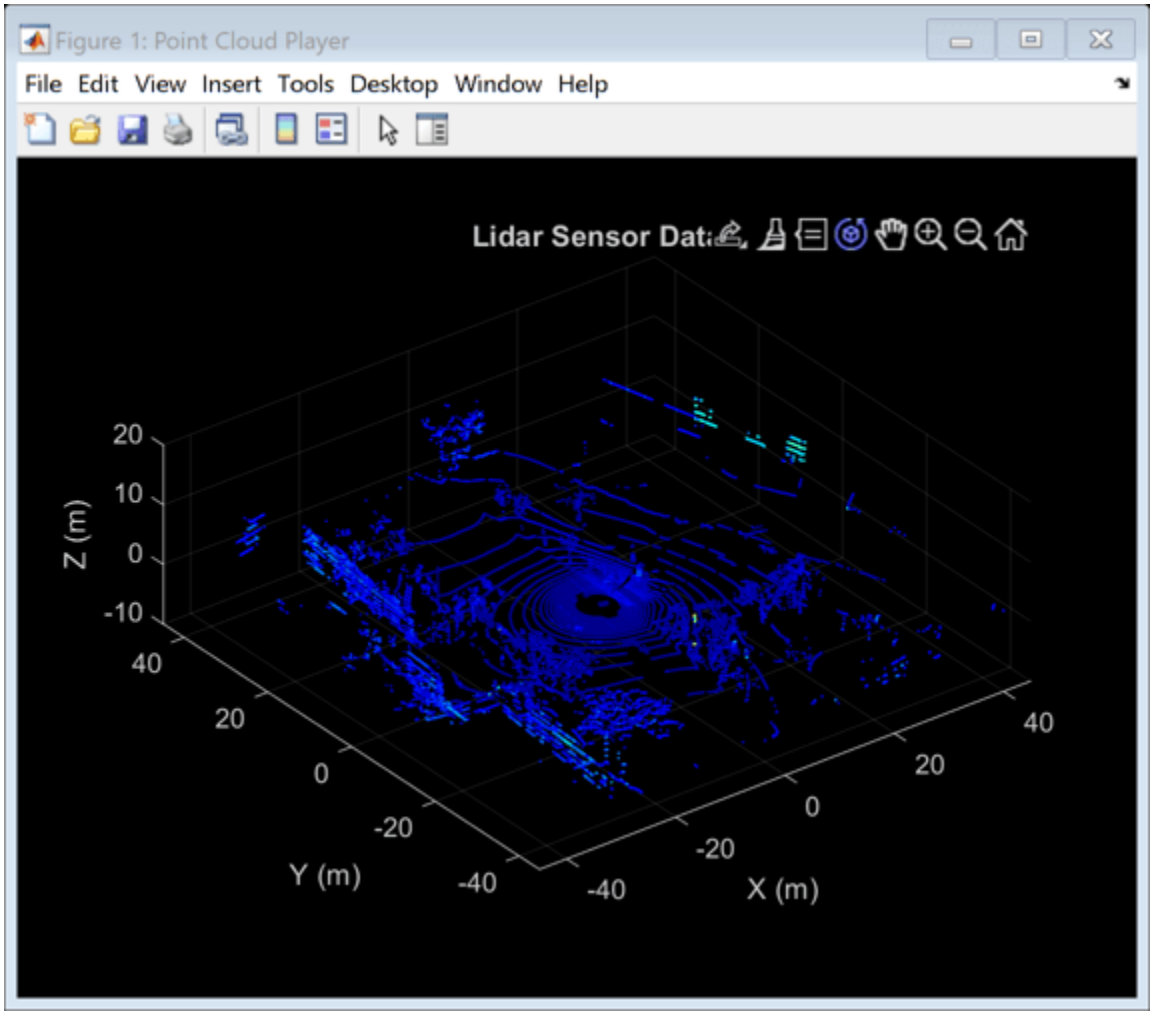
Visualize Driving Data

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
xlimits = [-45 45]; % meters
ylimits = [-45 45];
zlimits = [-10 20];

% Create a pcplayer to visualize streaming point clouds from lidar sensor
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')

% 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
```

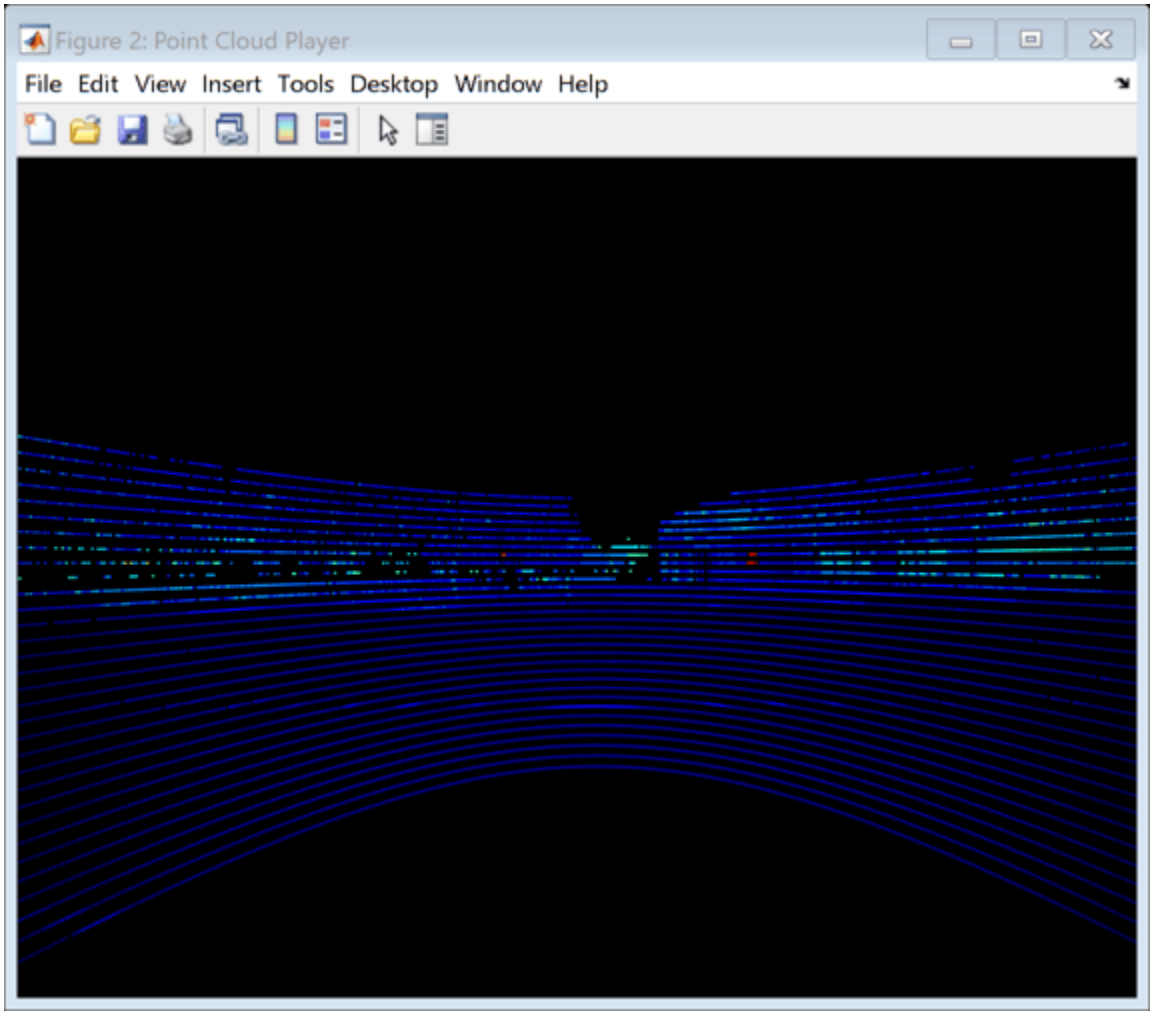




Use Recorded Lidar Data to Build a Map

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 `pcregistericp` and `pcregisterndt` for more details about each algorithm. This example uses NDT, because it is typically more accurate, especially when considering rotations. The `pcregisterndt` 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:

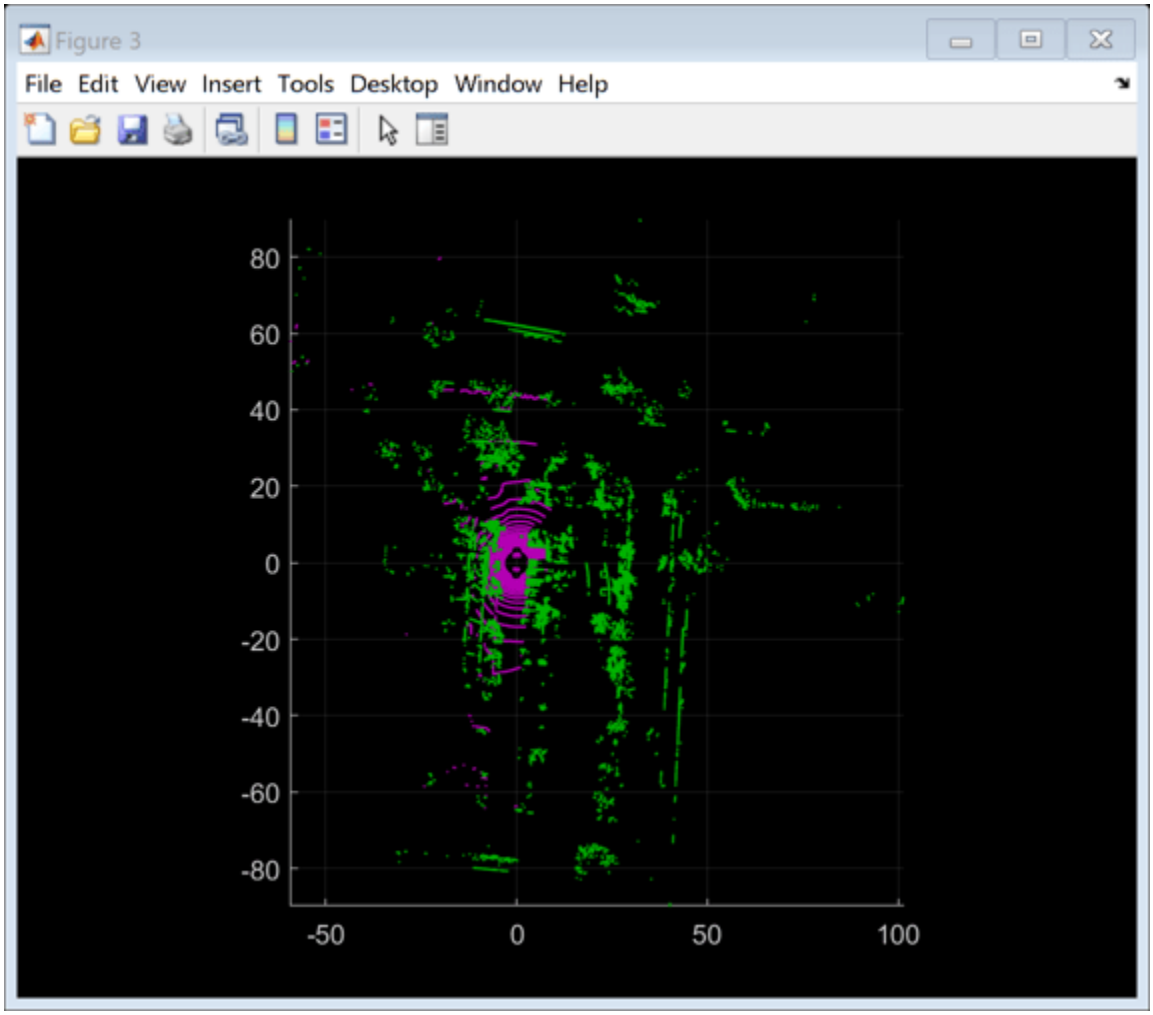
- Detect and remove the ground plane
- Detect and remove ego-vehicle

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);
```

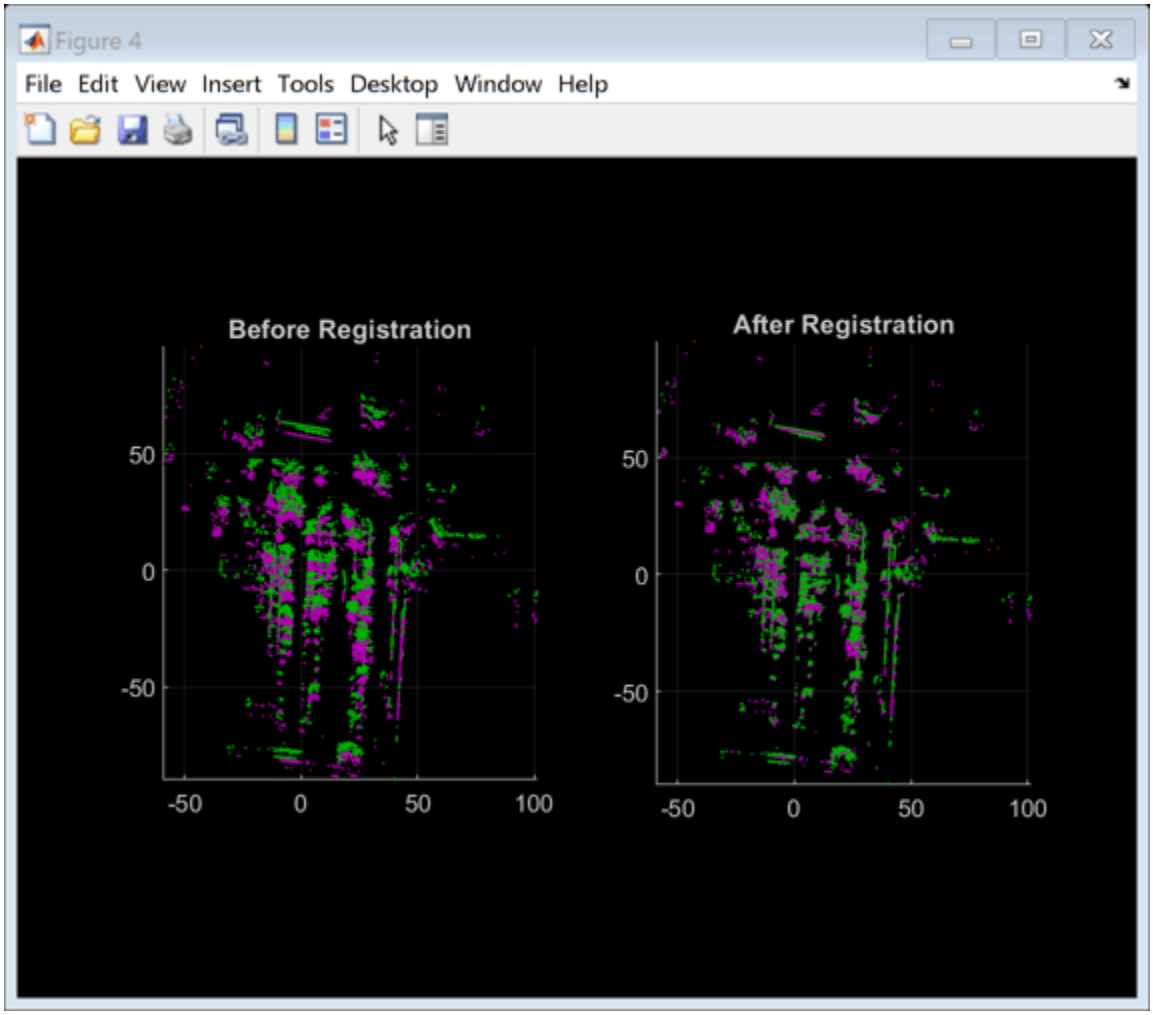
Display the raw and processed point clouds in top-view. Magenta points were removed during processing. These points correspond to the ground plane and ego vehicle.

```
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  = pcdownsample(fixedProcessed, 'random', downsamplePercent);
movingDownsampled = pcdownsample(movingProcessed, 'random', downsamplePercent);
```



After preprocessing the point clouds, register them using NDT. Visualize the alignment before and after registration.

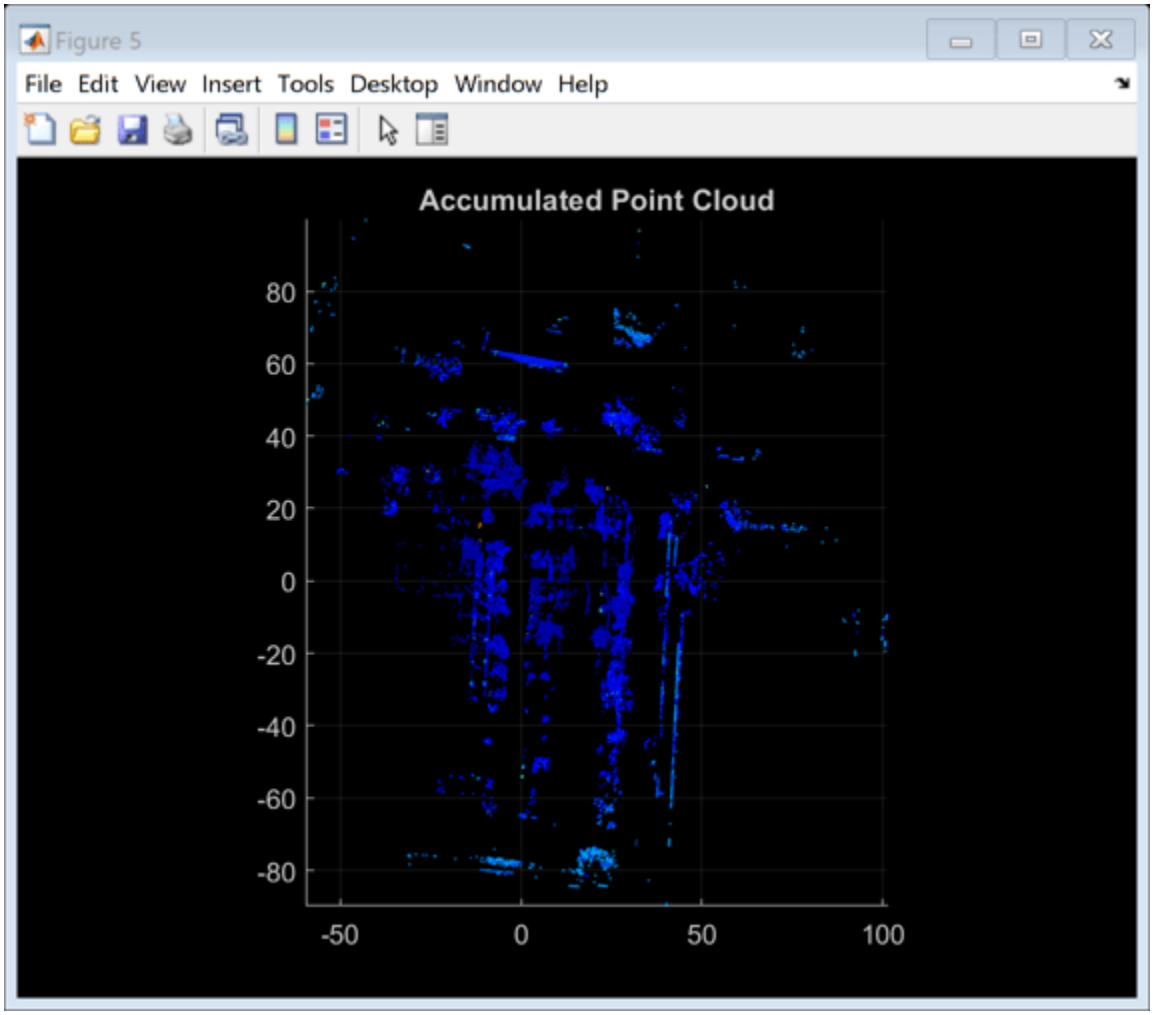
```
regGridStep = 5;
tform = pcregisterndt(movingDownsampled, fixedDownsampled, regGridStep);

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:

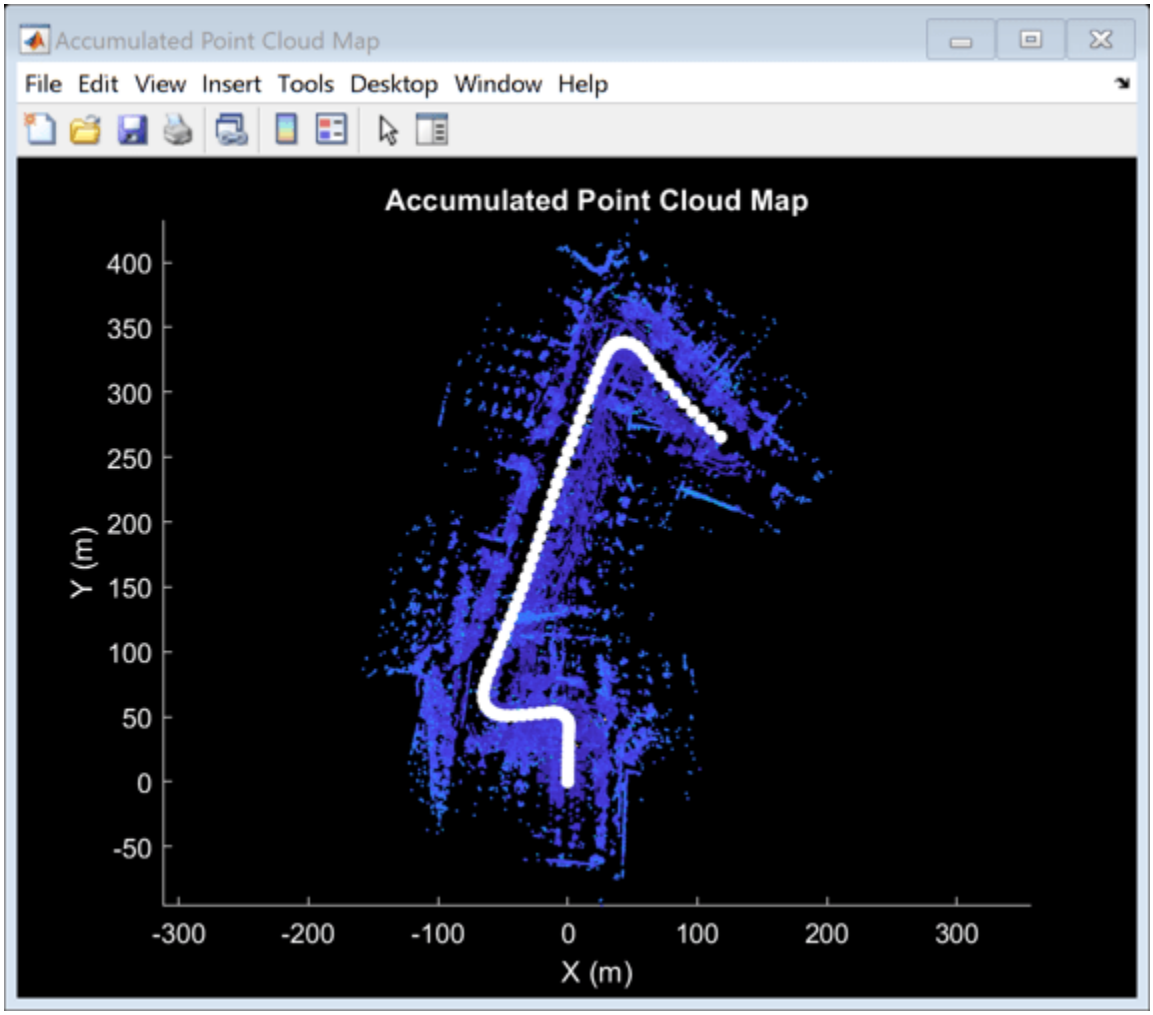
- Processing the point cloud by removing the ground plane and ego vehicle, using the `processPointCloud` method.
- Downsampling the point cloud.
- Estimating the rigid transformation required to merge the previous point cloud with the current point cloud.
- Transforming the point cloud back to the first 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:

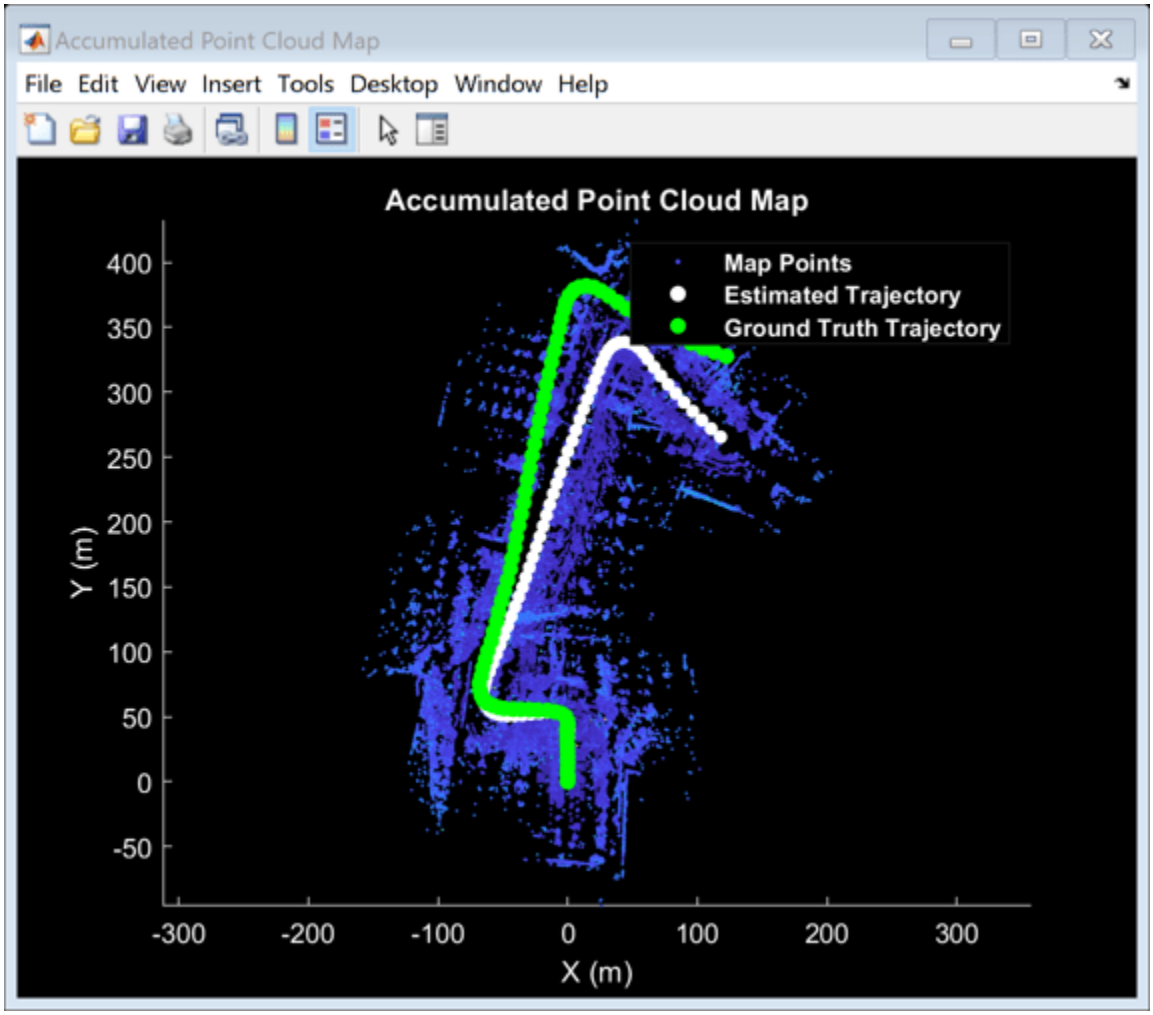
1. Convert the GPS coordinates to local Cartesian East-North-Up coordinates using the `latlon2local` (Automated Driving Toolbox) function. The GPS location from the start of the trajectory is used as the reference point and defines the origin of the local x,y,z coordinate system.
2. Rotate the Cartesian 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);
```

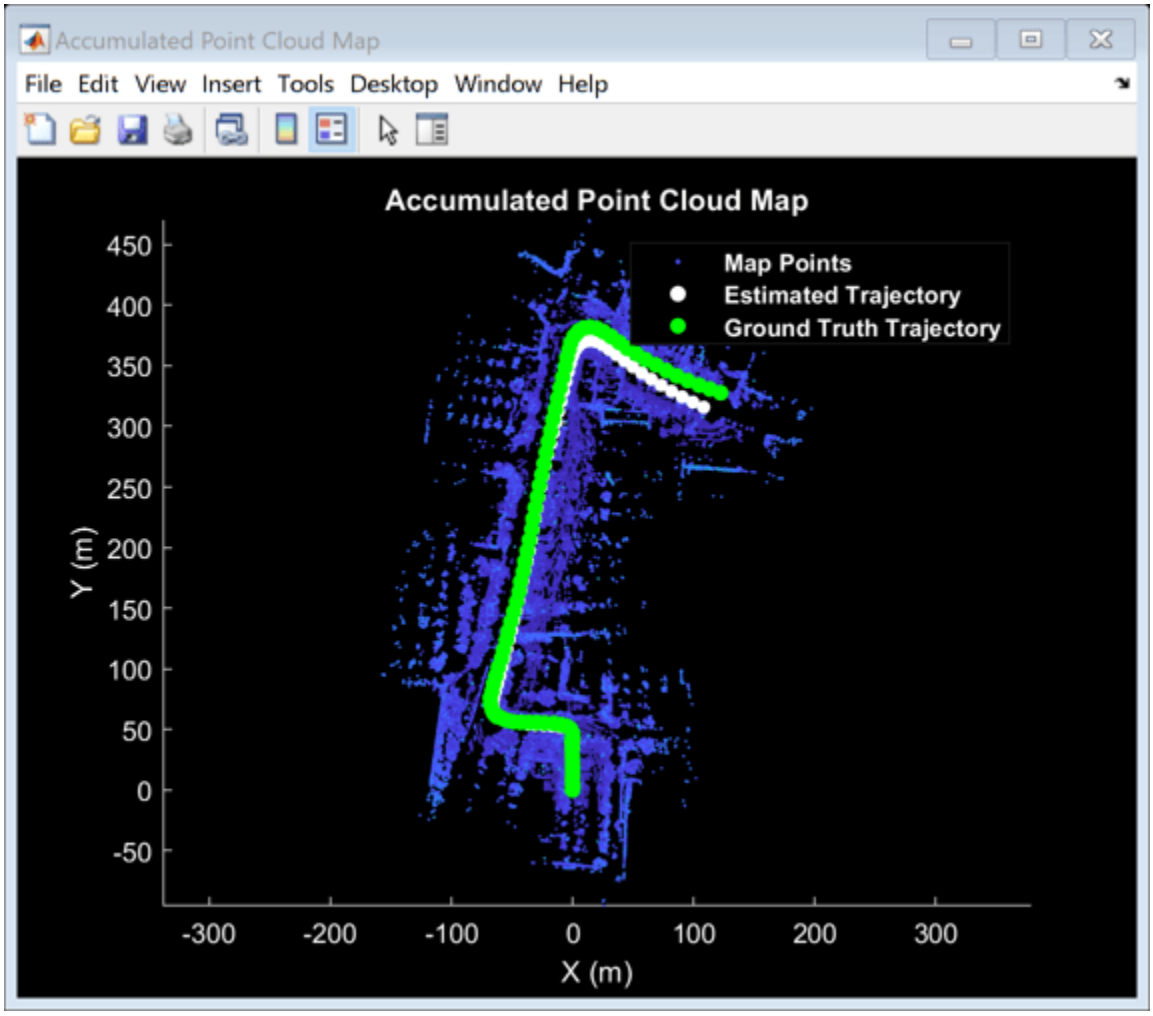
Use IMU Orientation to Improve Built Map

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);

    % 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 pcfitplane. For organized
    % point clouds, consider using segmentGroundFromLidarData instead.
    maxDistance    = 0.4;       % meters
    maxAngDistance = 5;         % degrees
    refVector      = [0, 0, 1]; % z-direction

    [~,groundIndices] = pcfitplane(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
`pcplayer` | `geoplayer` | `pointCloud`

More About

- “Build a Map from Lidar Data Using SLAM” on page 5-55
- “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";

% Get the full file path to the PNG files in the scenario1 folder.
pointCloudFilePattern = fullfile(dataFolder, 'scenario1', 'scan*.png');
numExpectedFiles = 2513;

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);

elseif folderExists && numel(dir(pointCloudFilePattern)) < numExpectedFiles
    % Redownload the data if it got reduced in the temporary directory.
    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, pointCloudFilePattern);

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: [64×870×3 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.

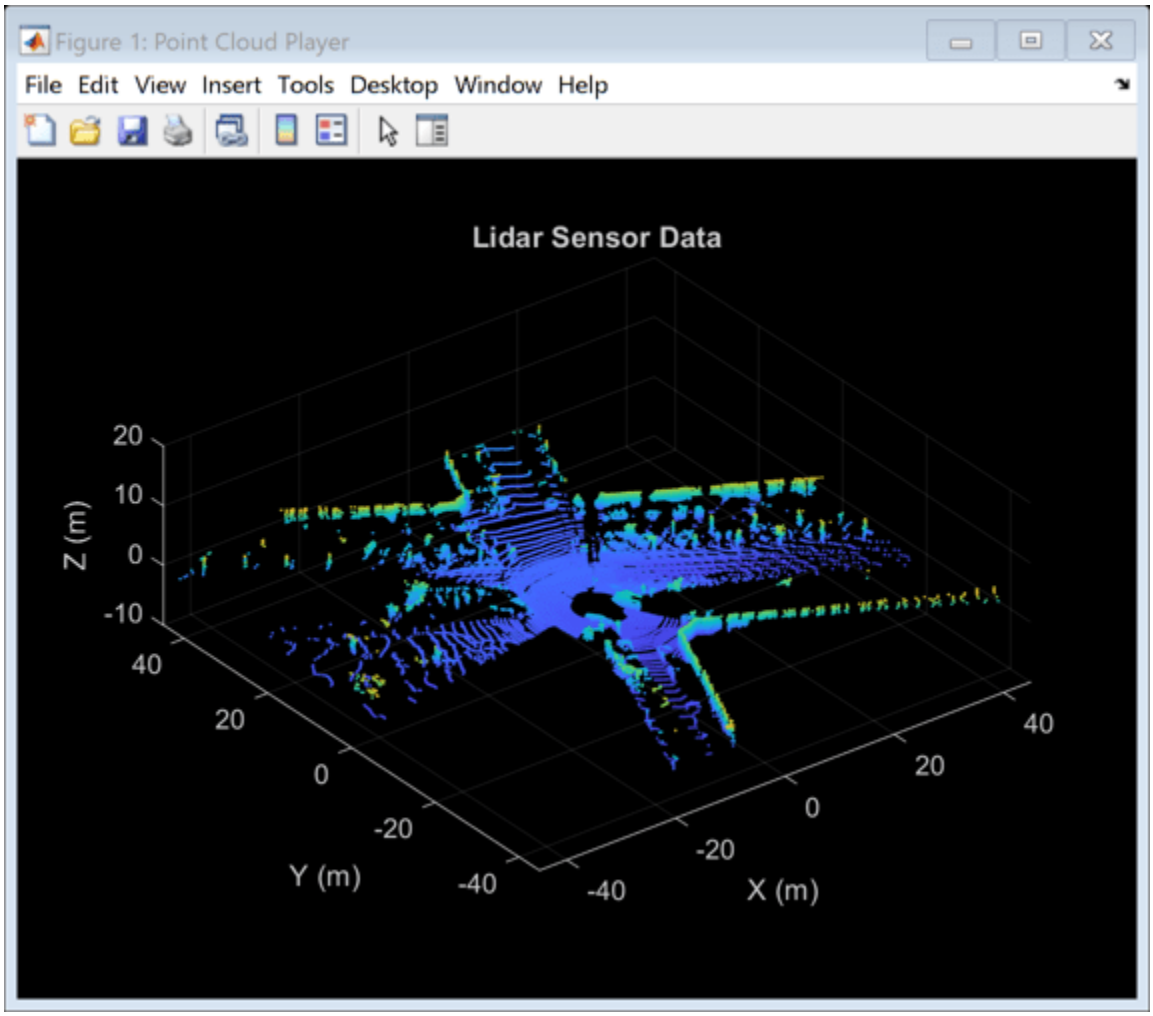
```
disp(insDataTable(1, :))
```

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 evey 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.

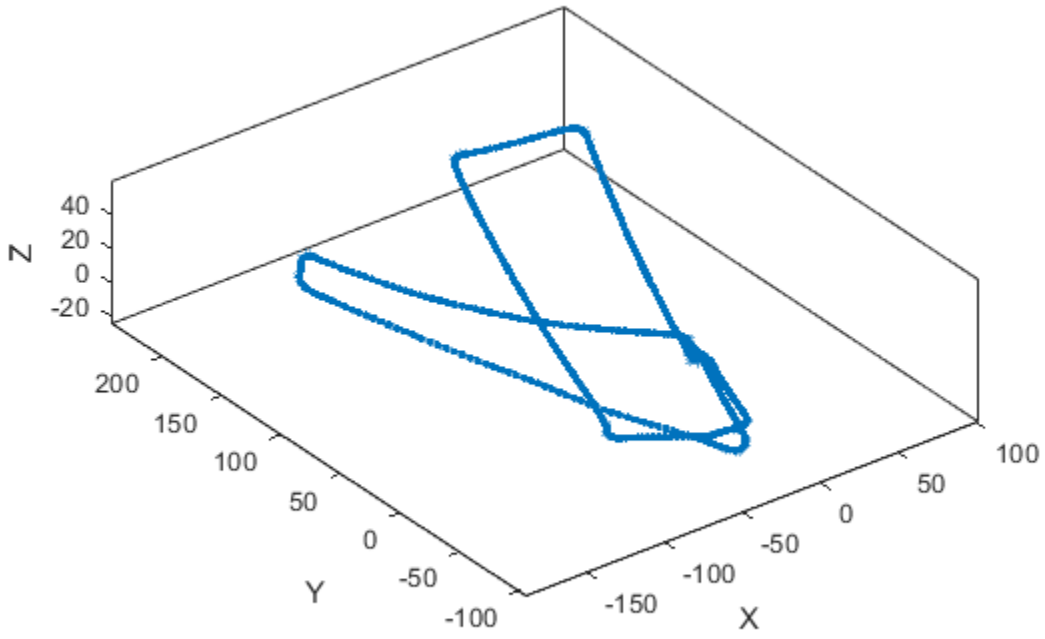
This approach of incrementally building a map and estimating the trajectory of the vehicle is called *odometry*.

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

        % 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.

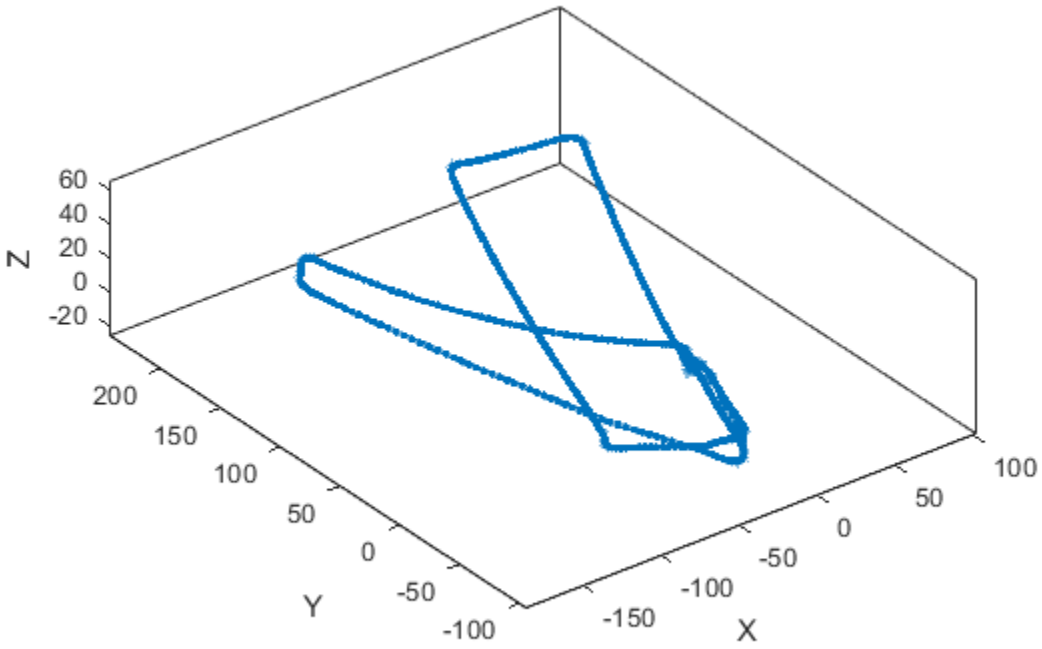
A key aspect contributing to the effectiveness of graph SLAM in correcting drift is the accurate detection of loops, that is, places that have been previously visited. This is called *loop closure detection* or *place recognition*. Adding edges to the pose graph corresponding to loop closures provides 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 a `scanContextLoopDetector` object to manage the scan context descriptors that correspond to each view. It uses the `detectLoop` object function to detect loop closures with a two phase descriptor search algorithm. In the first phase, it computes the ring key subdescriptors to find potential loop candidates. In the second phase, it classifies views as loop closures by thresholding the scan context distance.

```
% Set random seed to ensure reproducibility
rng(0);

% Create an empty view set
vSet = pcviewset;

% Create a loop closure detector
loopDetector = scanContextLoopDetector;

% 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 = 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);

        % Extract the scan context descriptor from the first point cloud
        descriptor = scanContextDescriptor(ptCloudOrig);
```

```
    % Add the first descriptor to the loop closure detector
    addDescriptor(loopDetector, viewId, descriptor)

    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);

% Extract the scan context descriptor from the point cloud
descriptor = scanContextDescriptor(ptCloudOrig);

% Add the descriptor to the loop closure detector
addDescriptor(loopDetector, viewId, descriptor)

% Detect loop closure candidates
loopViewId = detectLoop(loopDetector);

% A loop candidate was found
if ~isempty(loopViewId)
    loopViewId = loopViewId(1);

    % Retrieve point cloud from view set
    loopView = findView(vSet, loopViewId);
    ptCloudOrig = loopView.PointCloud;

    % Process point cloud
    ptCloudOld = helperProcessPointCloud(ptCloudOrig);

    % Downsample point cloud
    ptCloudOld = pcdownsample(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

    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:

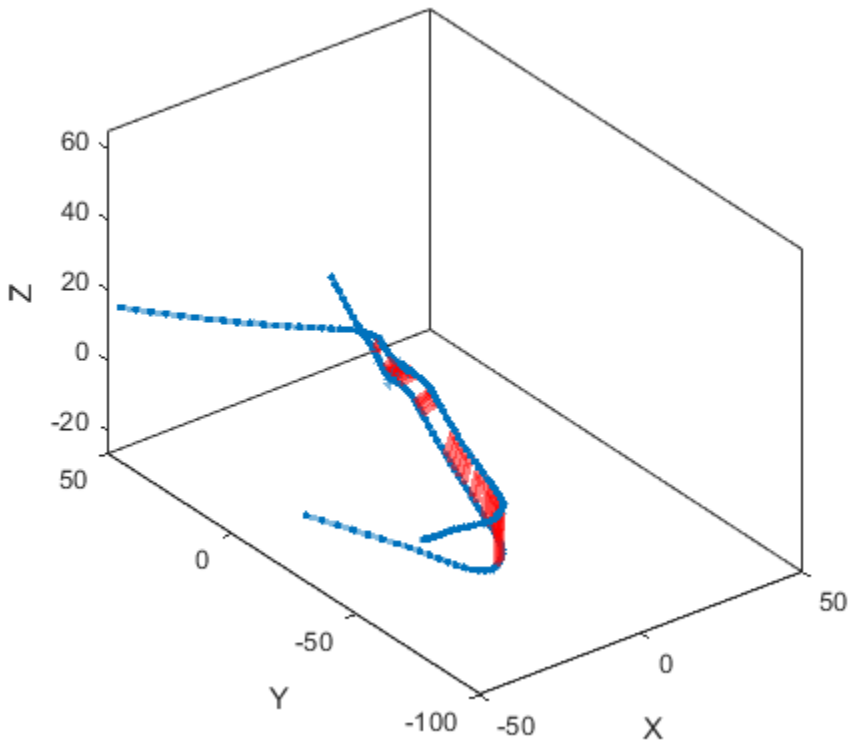
- Nodes containing the absolute pose of each view
- Edges containing the relative pose constraints of each connection

```
G = createPoseGraph(vSet);
disp(G)
```

```
  digraph with properties:

    Edges: [539×3 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)
```

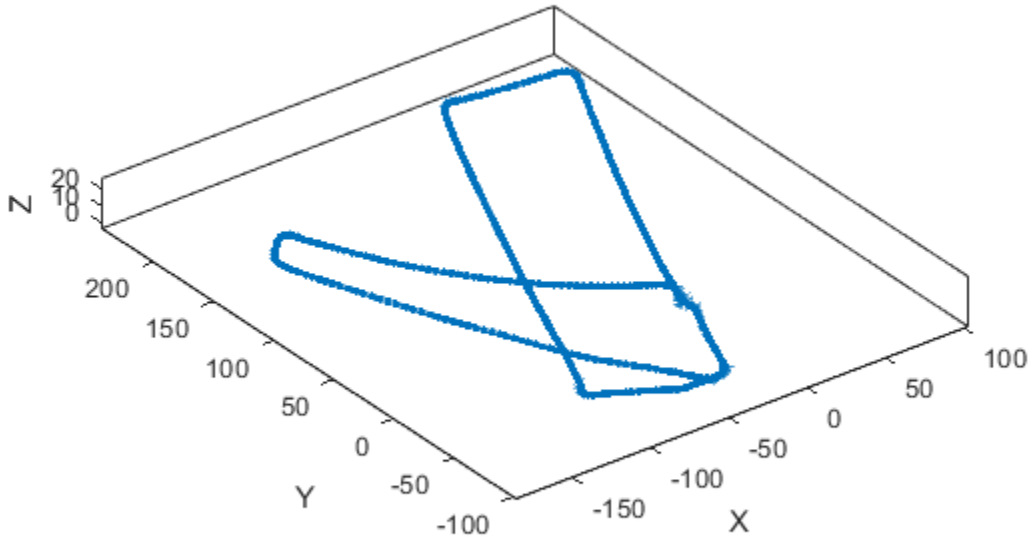


Optimize the pose graph using `optimizePoseGraph`.

```
optimG = optimizePoseGraph(G, 'g2o-levenberg-marquardt');

vSetOptim = updateView(vSet, optimG.Nodes);
```

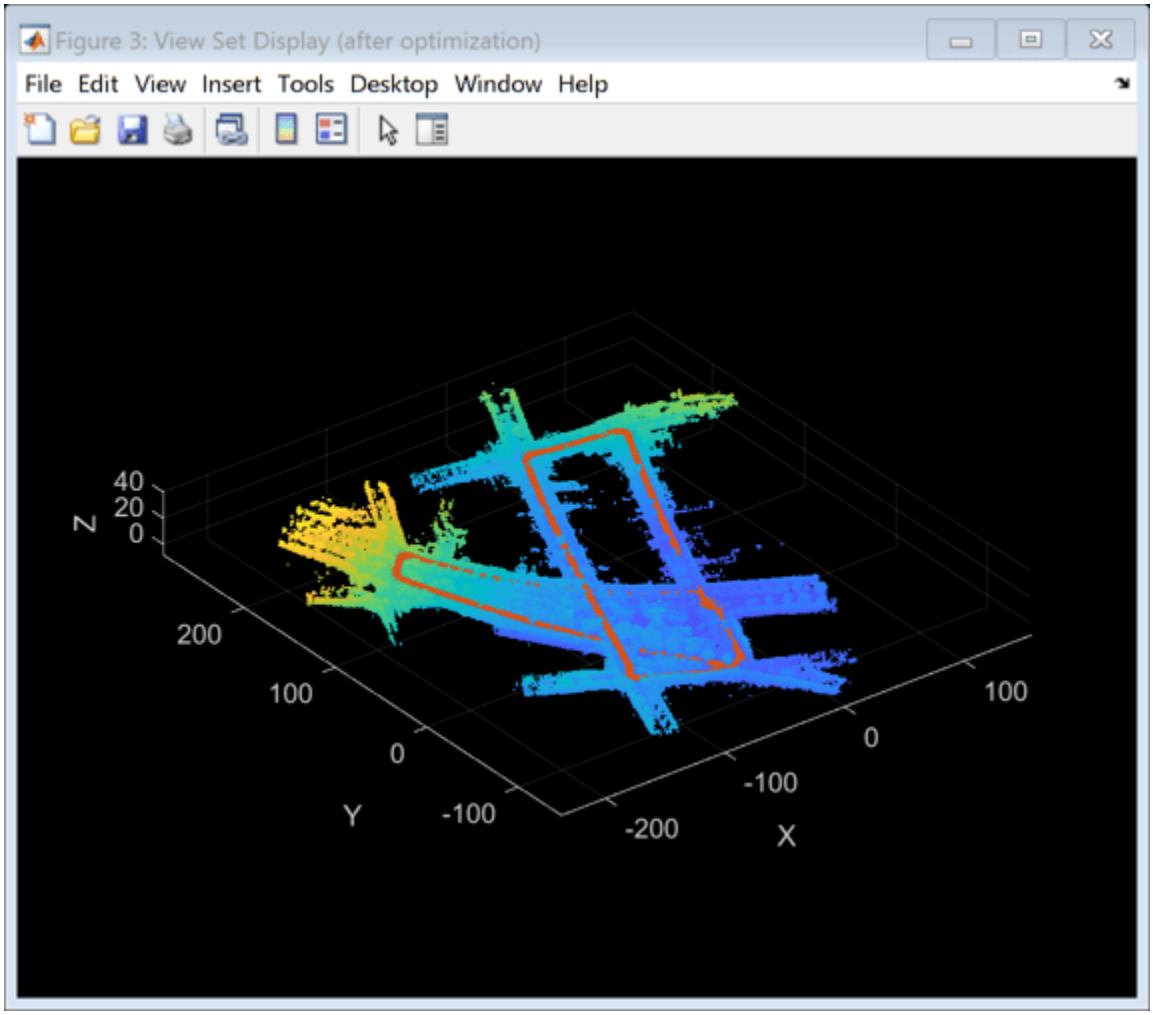
Display the view set with optimized poses. Notice that the detected loops are now merged, resulting in a more accurate trajectory.

```
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, pointCloudFilePattern)
%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.

% 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 pcfitplane, pointCloud/findNeighborsInRadius.

arguments
    ptCloudIn (1,1) pointCloud
    method          string      {mustBeMember(method, ["planefit","rangefloodfill"])} = "rangeflo
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] = pcfitplane(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
```

See Also

Functions
`pcregisterndt` | `pcshow` | `createPoseGraph` | `optimizePoses`

Objects
`pcviewset` | `rigid3d` | `pointCloud`

More About

- “Build a Map from Lidar Data” on page 5-35
- “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 = pcdownsample(ptCloudRef, 'gridAverage', gridSize);
moving = pcdownsample(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.

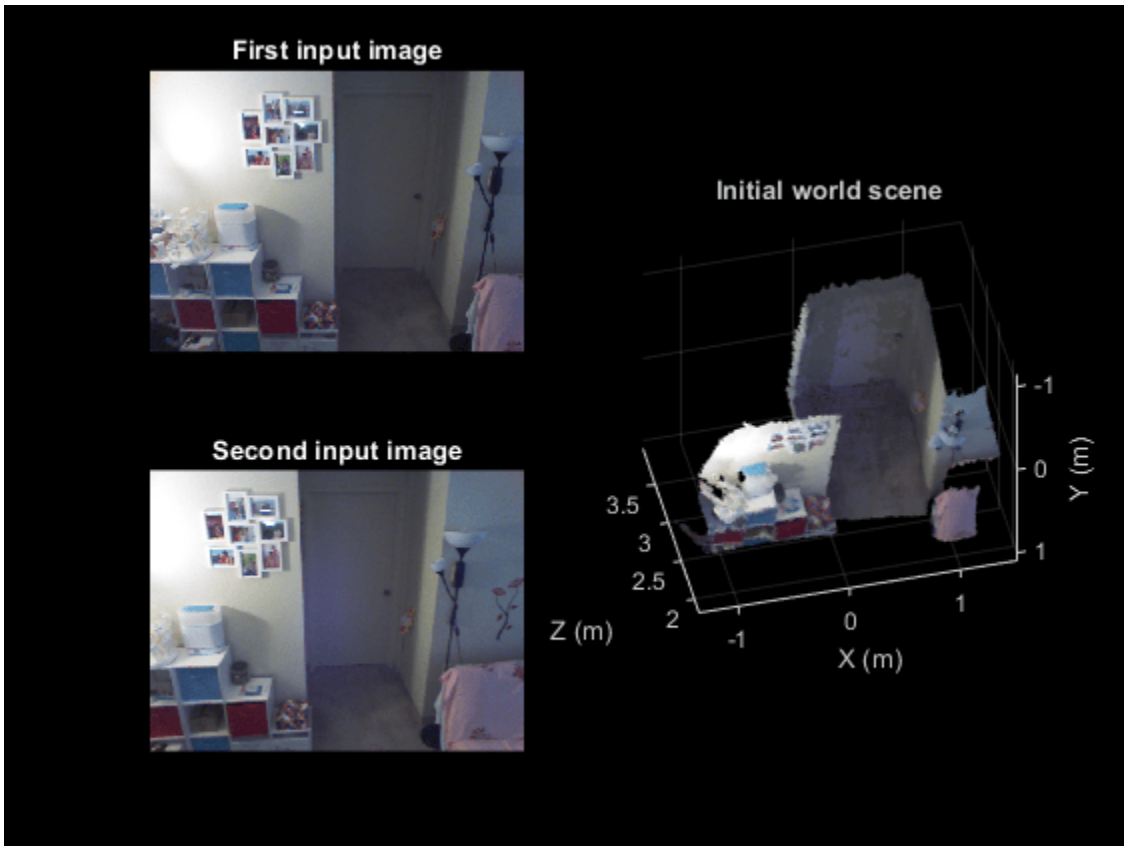
Begin by finding the rigid transformation for aligning the second point cloud with the first point cloud. Use it to transform the second point cloud to the reference coordinate system defined by the first point cloud.

```
tform = pcregistericp(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)')
```

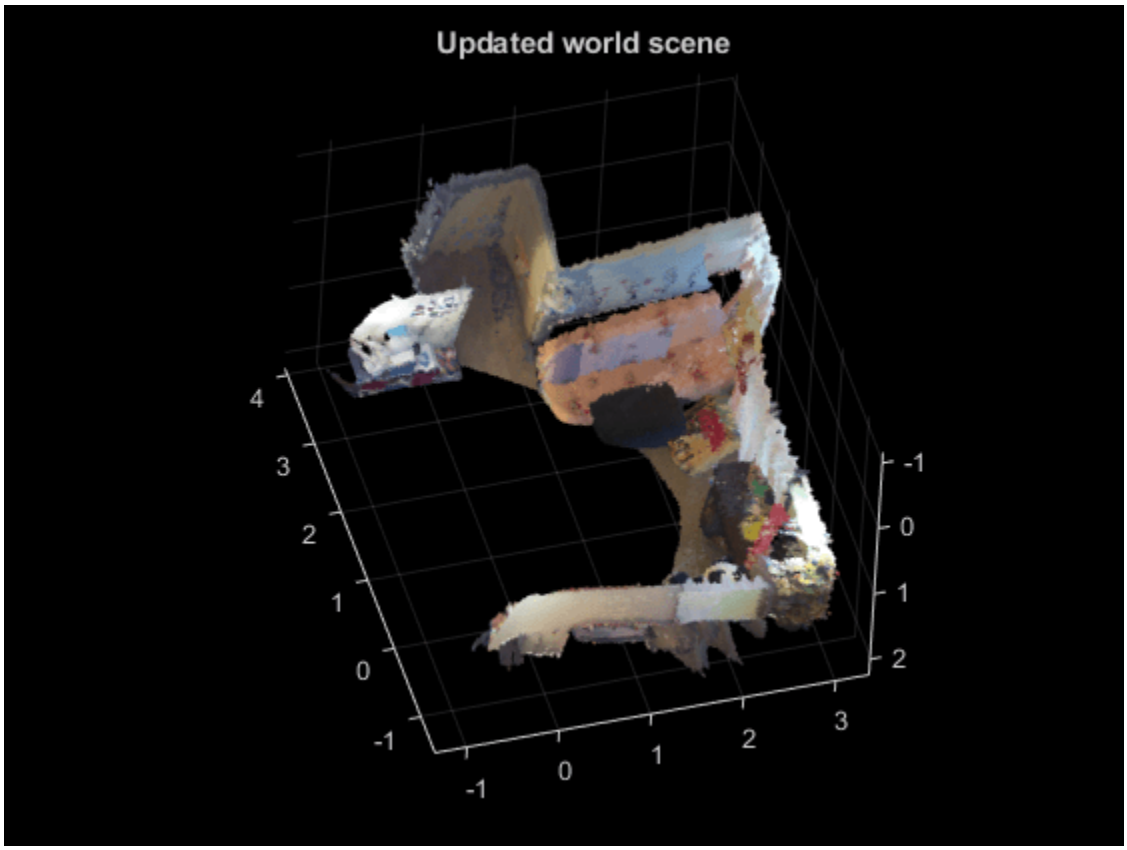



```
drawnow
```

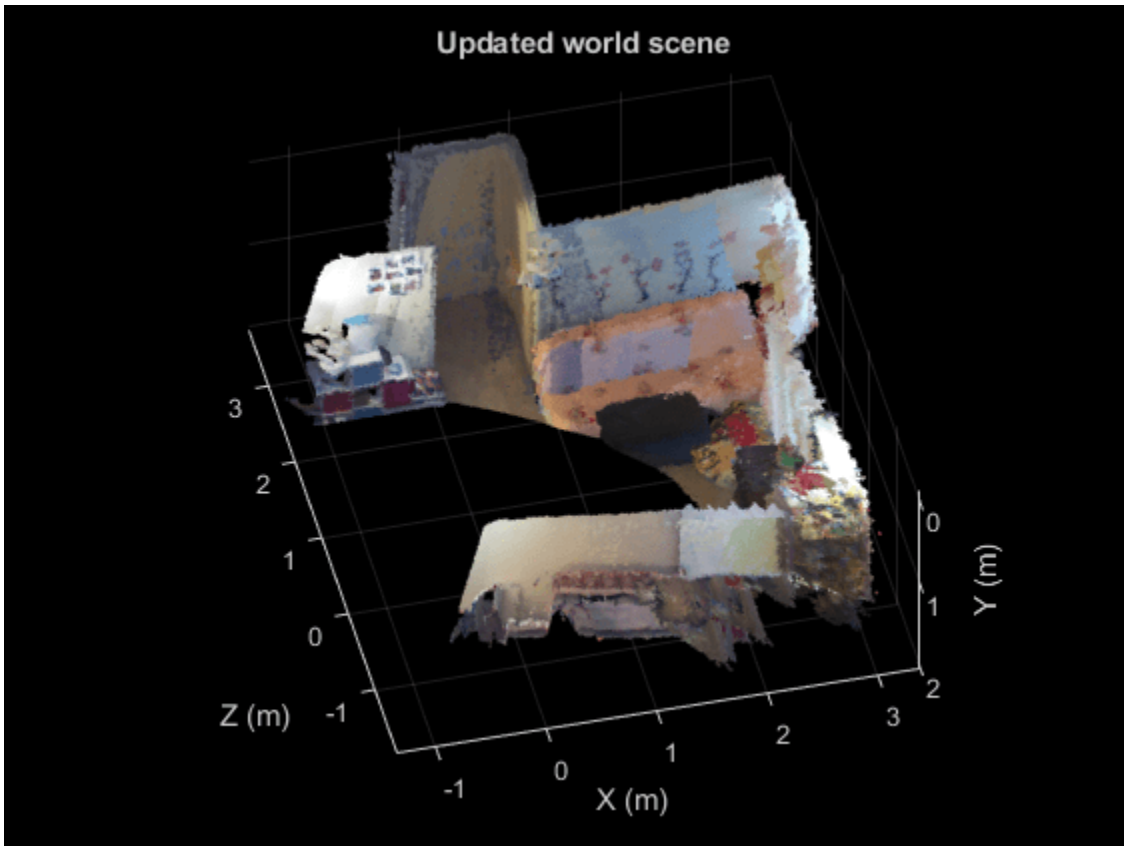
Stitch a Sequence of Point Clouds

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 = pcdownsample(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)')
```


Updated world scene
3
2
1
0
Z (m)
-1
-1
0
1
2
3
X (m)
0
1
2
Y (m)

Computer Vision with Simulink Examples

- “Multicore Simulation of Video Processing System” on page 6-2
- “Concentricity Inspection” on page 6-6
- “Object Counting” on page 6-8
- “Video Focus Assessment” on page 6-10
- “Video Compression” on page 6-12
- “Motion Detection” on page 6-14
- “Pattern Matching” on page 6-16
- “Scene Change Detection” on page 6-19
- “Surveillance Recording” on page 6-21
- “Traffic Warning Sign Recognition” on page 6-23
- “Abandoned Object Detection” on page 6-26
- “Color-based Road Tracking” on page 6-29
- “Detect and Track Face” on page 6-33
- “Lane Departure Warning System” on page 6-40
- “Tracking Cars Using Foreground Detection” on page 6-44
- “Tracking Cars Using Optical Flow” on page 6-47
- “Tracking Based on Color” on page 6-49
- “Video Mosaicking” on page 6-51
- “Video Stabilization” on page 6-56
- “Periodic Noise Reduction” on page 6-58
- “Rotation Correction” on page 6-60
- “Barcode Recognition Using Live Video Acquisition” on page 6-63
- “Edge Detection Using Live Video Acquisition” on page 6-65
- “Noise Removal and Image Sharpening” on page 6-70
- “Track Marker Using Simulink Images” on page 6-76

Multicore Simulation of Video Processing System

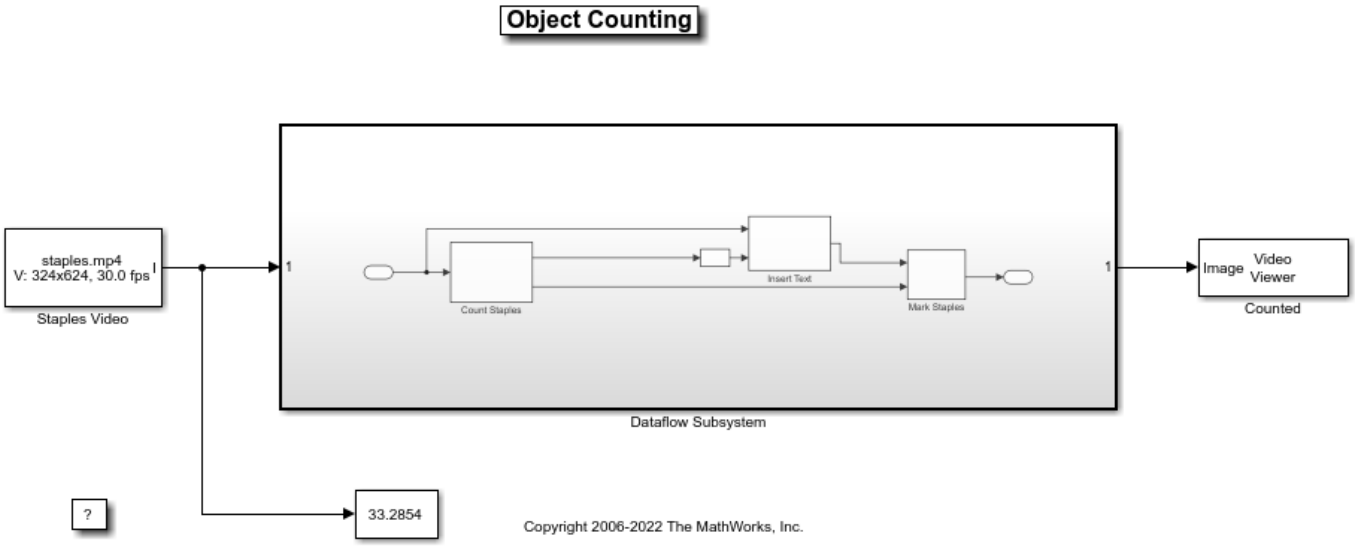
This example shows how to run a video processing system on multiple cores using dataflow execution domain in Simulink®.

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).

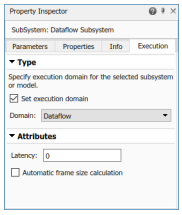
Object Counting in Video

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 accessing Property Inspector. To access Property Inspector, in the Simulink Toolstrip, on the Modeling tab, in the Design gallery select Property Inspector or on the Simulation tab, Prepare gallery, select Property Inspector.

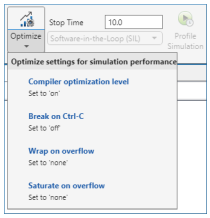


Dataflow domains automatically partition your model into multiple threads for better performance. Once you set the **Domain** parameter to `Dataflow`, you can use the **Multicore** tab analysis to analyze your model to get better performance. The **Multicore** tab is available in the toolstrip when there is a dataflow domain in the model. To learn more about the **Multicore** tab, see "Perform Multicore Analysis for Dataflow" (DSP System Toolbox).

Analyzing Concurrency in Dataflow Subsystem

For this example the **Multicore** tab mode is set to `Simulation Profiling` for simulation performance analysis.

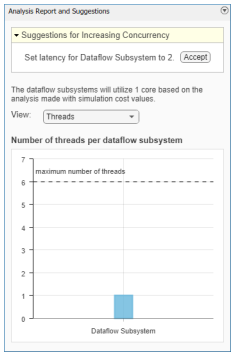
It is recommended to optimize model settings for optimal simulation performance. To accept the proposed model settings, on the **Multicore** tab, click **Optimize**. Alternatively, you can use the drop menu below the **Optimize** button to change the settings individually. In this example the model settings are already optimal.



On the **Multicore** tab, click the **Run Analysis** button to start the analysis of the dataflow domain for simulation performance. Once the analysis is finished, the Analysis Report and Suggestions window shows how many threads the dataflow subsystem uses during simulation.

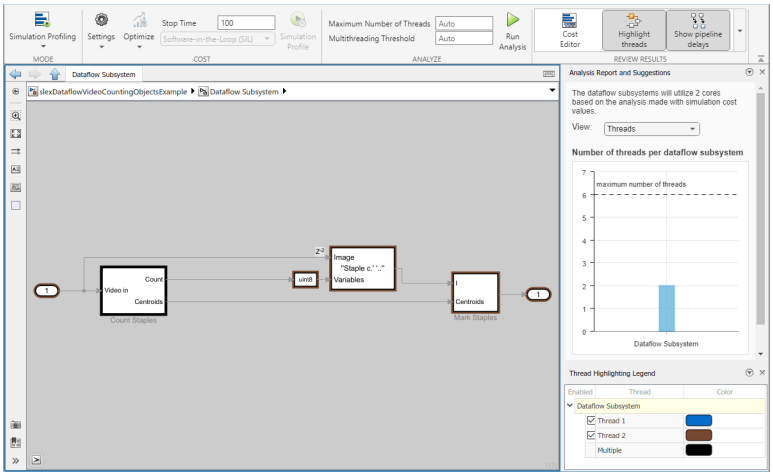
After analyzing the model, the Analysis Report and Suggestions window shows one thread because the data dependency between the blocks in the model prevents blocks from being executed concurrently. By pipelining the data dependent blocks, the dataflow subsystem can increase concurrency for higher data throughput. The Analysis Report and Suggestions window shows the recommended number of pipeline delays as Suggested for Increasing Concurrency. The suggested latency value is computed to give the best performance.

The following diagram shows the Analysis Report and Suggestions window where the suggested latency is 2 for the dataflow subsystem.



Click the **Accept** button 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.

The Analysis Report and Suggestions window now shows the number of threads as 2 meaning that the blocks inside the dataflow subsystem simulate in parallel using 2 threads. **Highlight threads** highlights the blocks with colors based on their thread allocation as shown in the **Thread Highlighting Legend**. **Show pipeline delays** shows where pipelining delays were inserted within the dataflow subsystem using Z^{-n} tags.



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 = 6.86s
Simulation execution time for single-threaded model = 13.34s
Actual speedup with dataflow: 1.9x
```

Summary

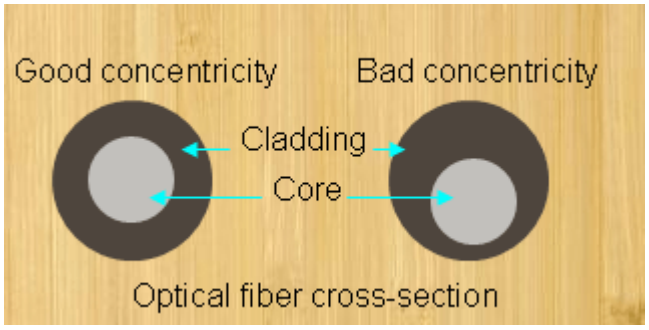
This example shows how multithreading using dataflow domain can improve performance in a video processing model using multiple cores on the desktop.

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.

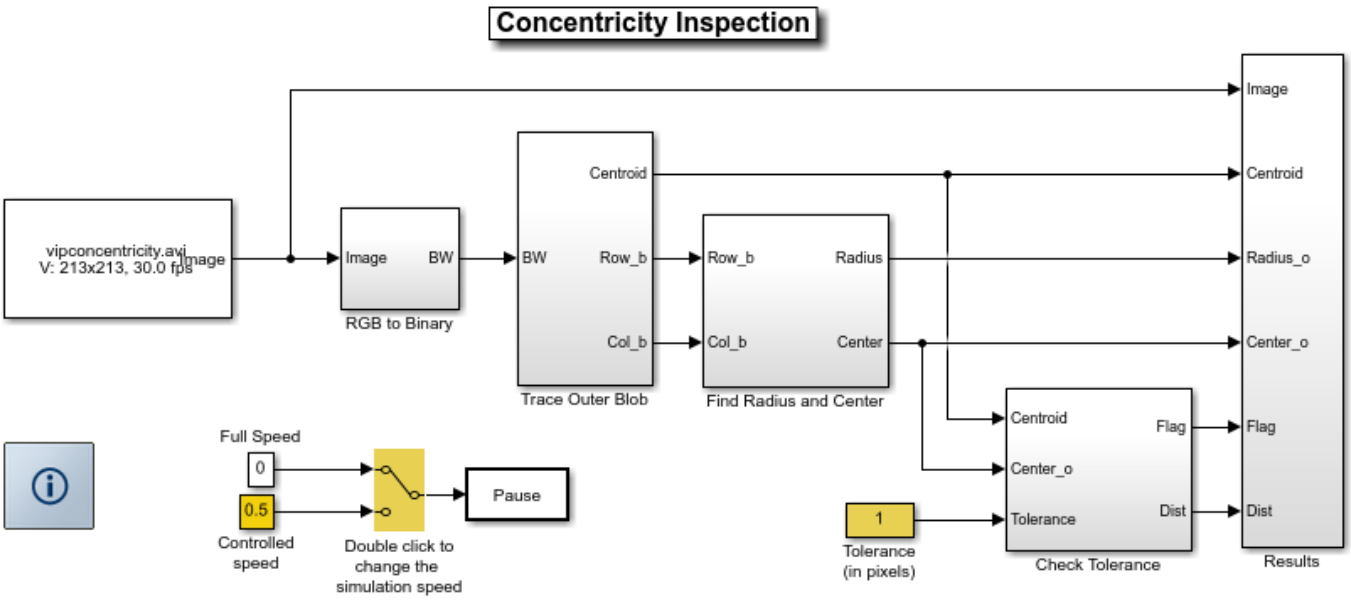
The following figure shows examples of optical fibers with good and bad concentricity:



Example Model

The following figure shows the Concentricity Inspection example model:

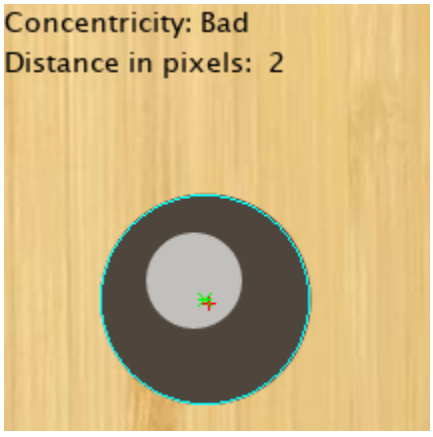
```
open_system('vipconcentricity');
```



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');
```



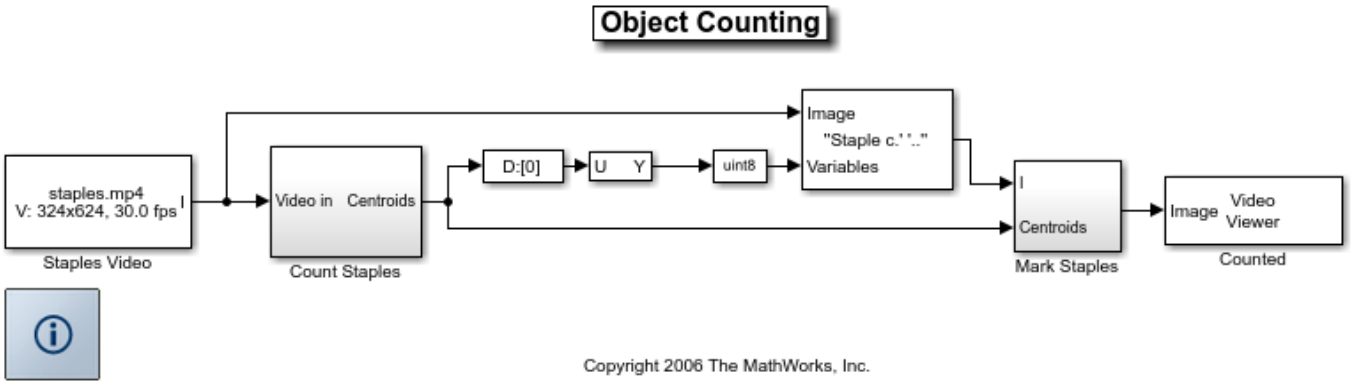

```
close_system('vipconcentricity', 0);
```

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.

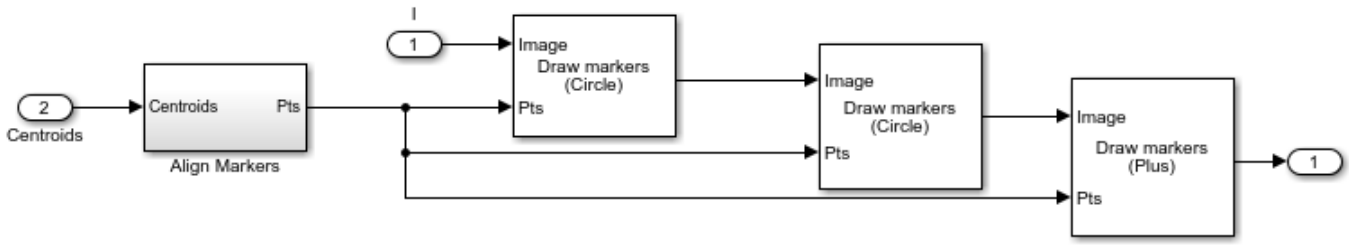


Count Staples Subsystem

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.

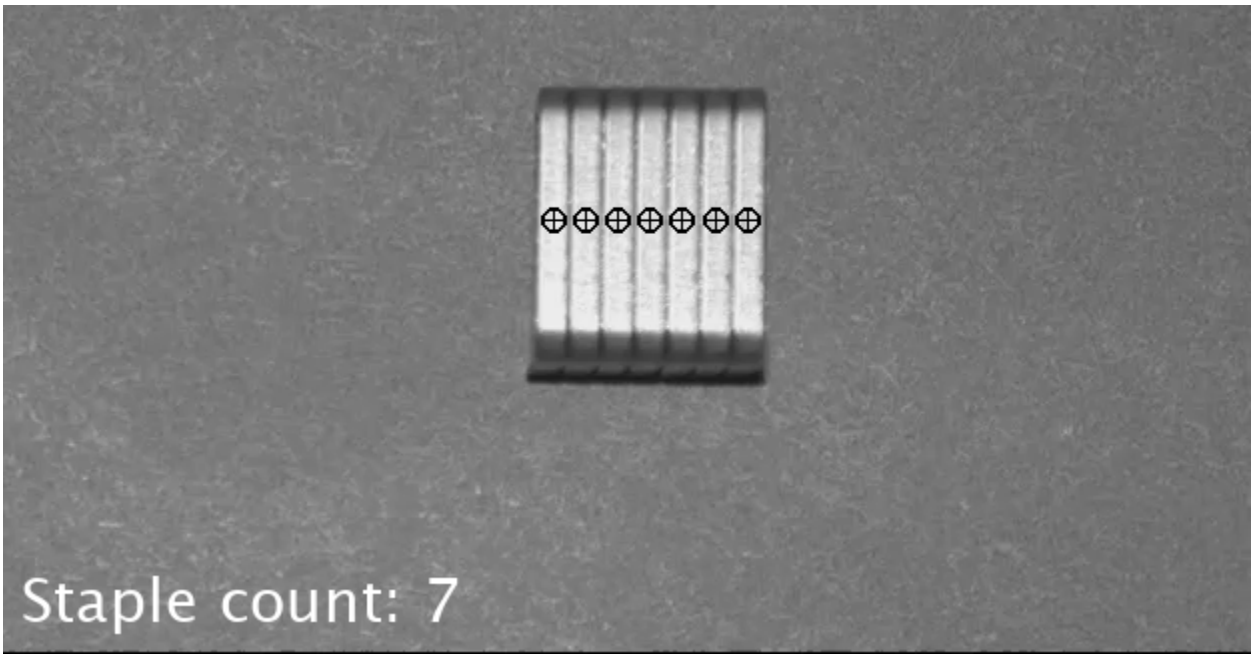
Mark Staples Subsystem

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.

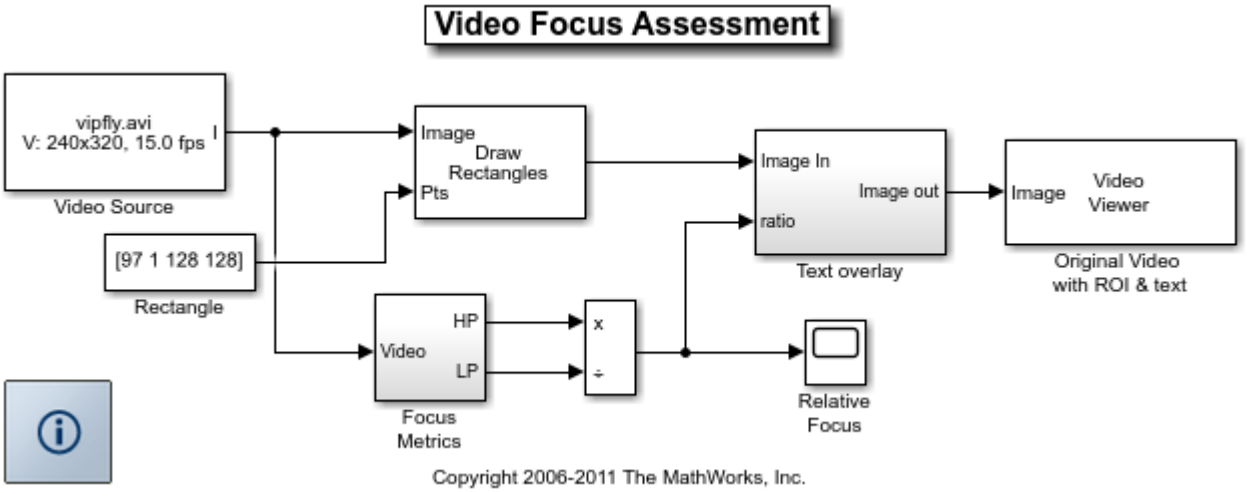


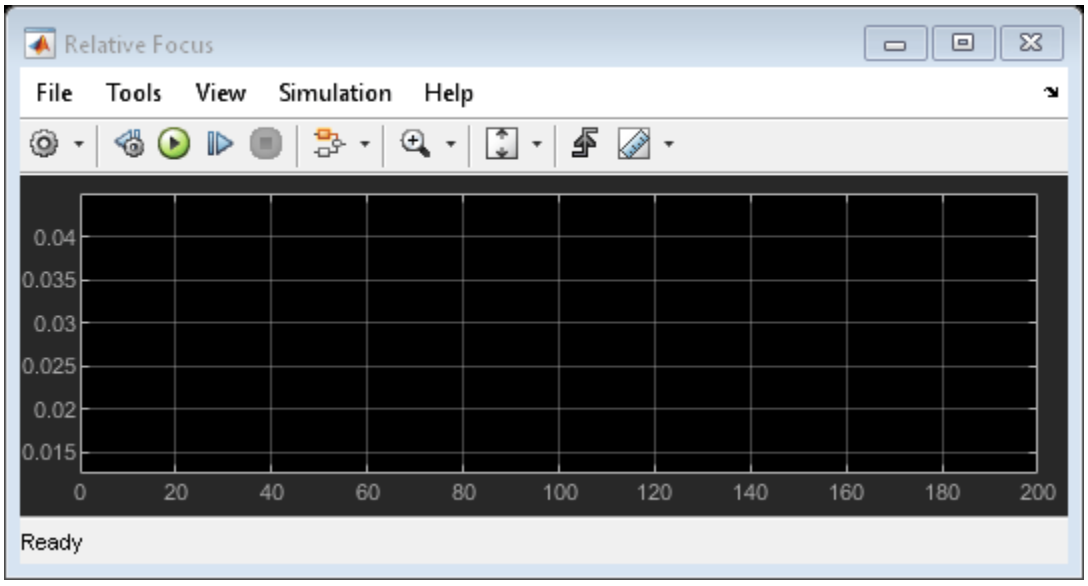
Video Focus Assessment

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:



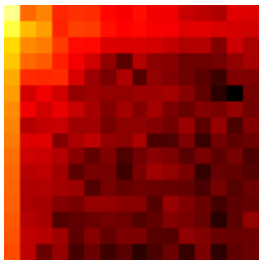


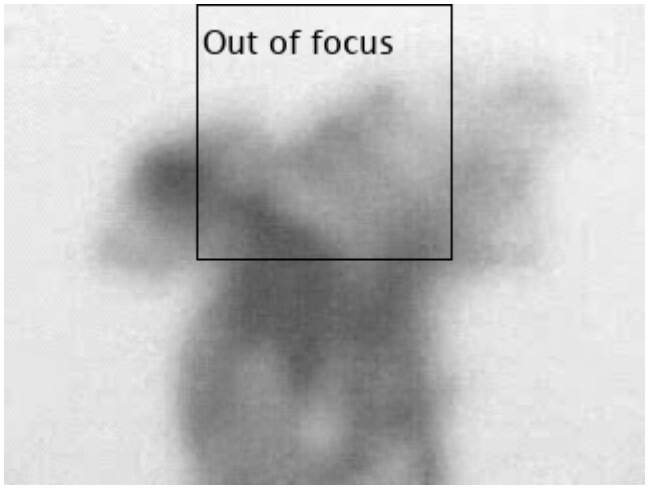
Video Focus Assessment Results

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.



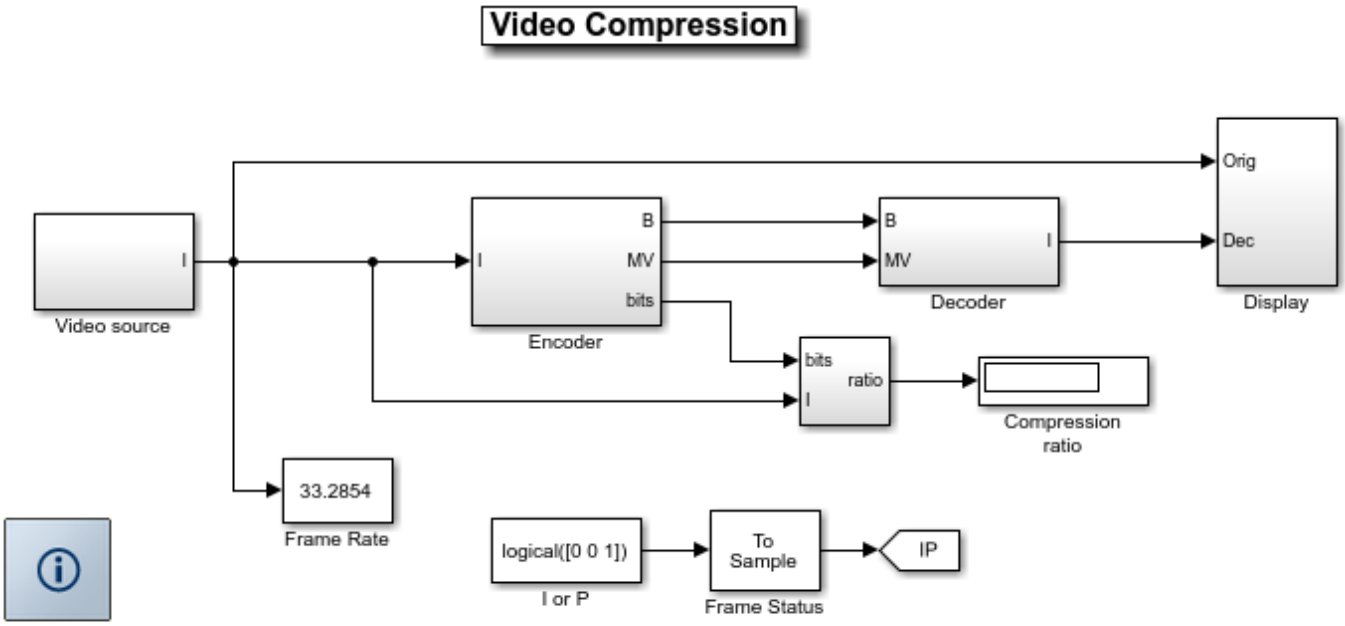


Video Compression

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.

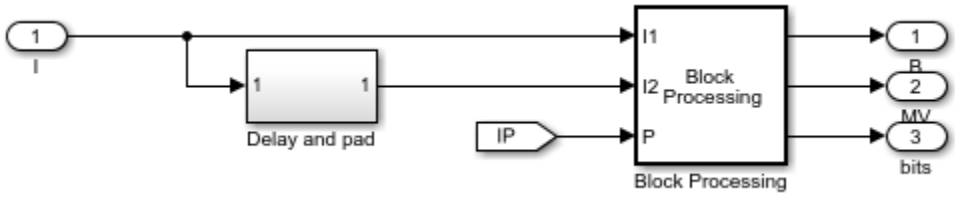
Example Model

The following figure shows the Video Compression model:



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:

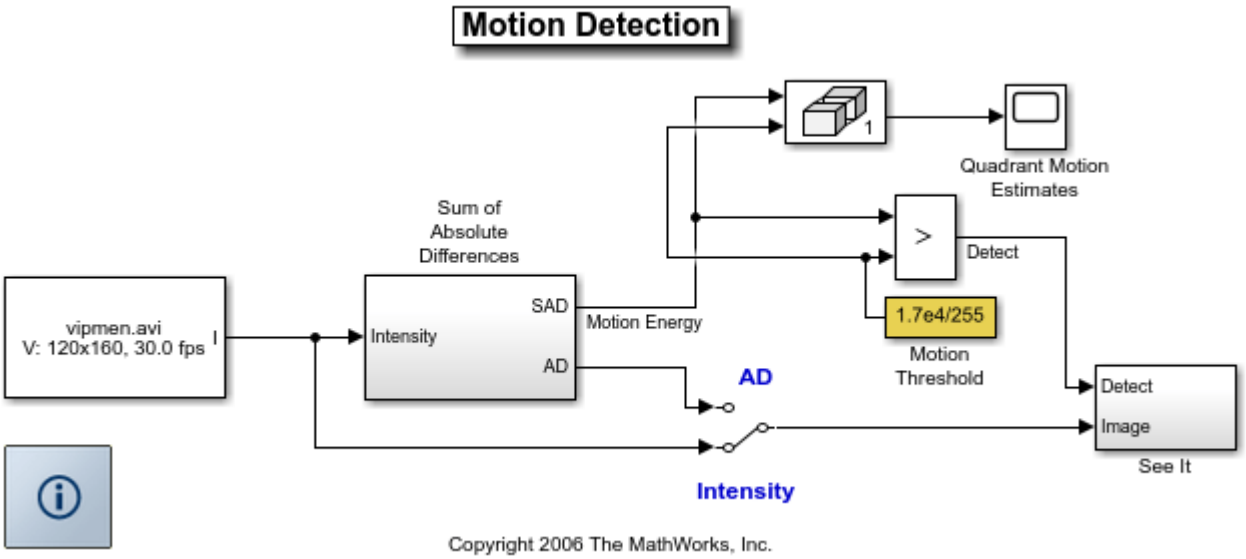
vipcodec_color.slx

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

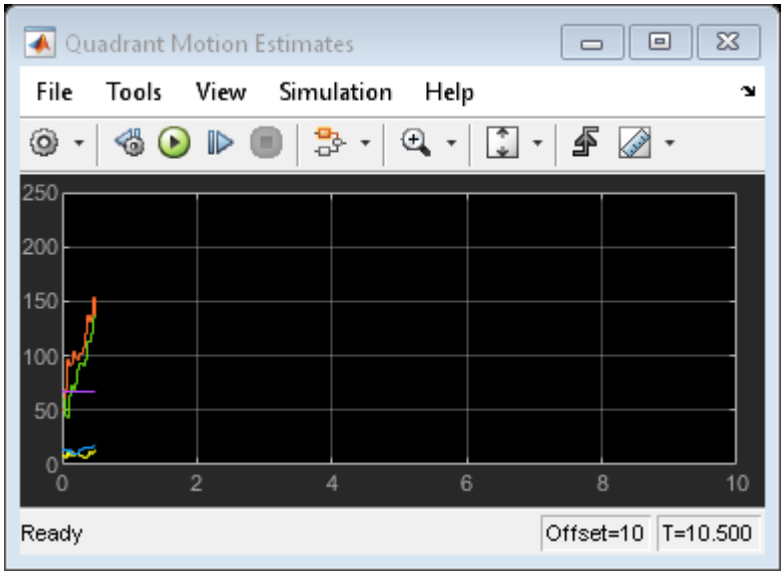
The following figure shows the Motion Detection example model:

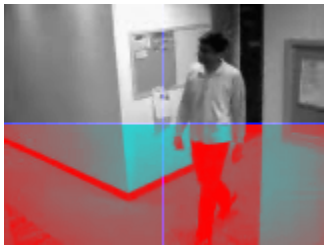


Motion Detection Results

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.


Quadrant Motion Estimates
File
Tools
View
Simulation
Help
250
200
150
100
50
0
0
2
4
6
8
10
Ready
Offset=10
T=10.500



Pattern Matching

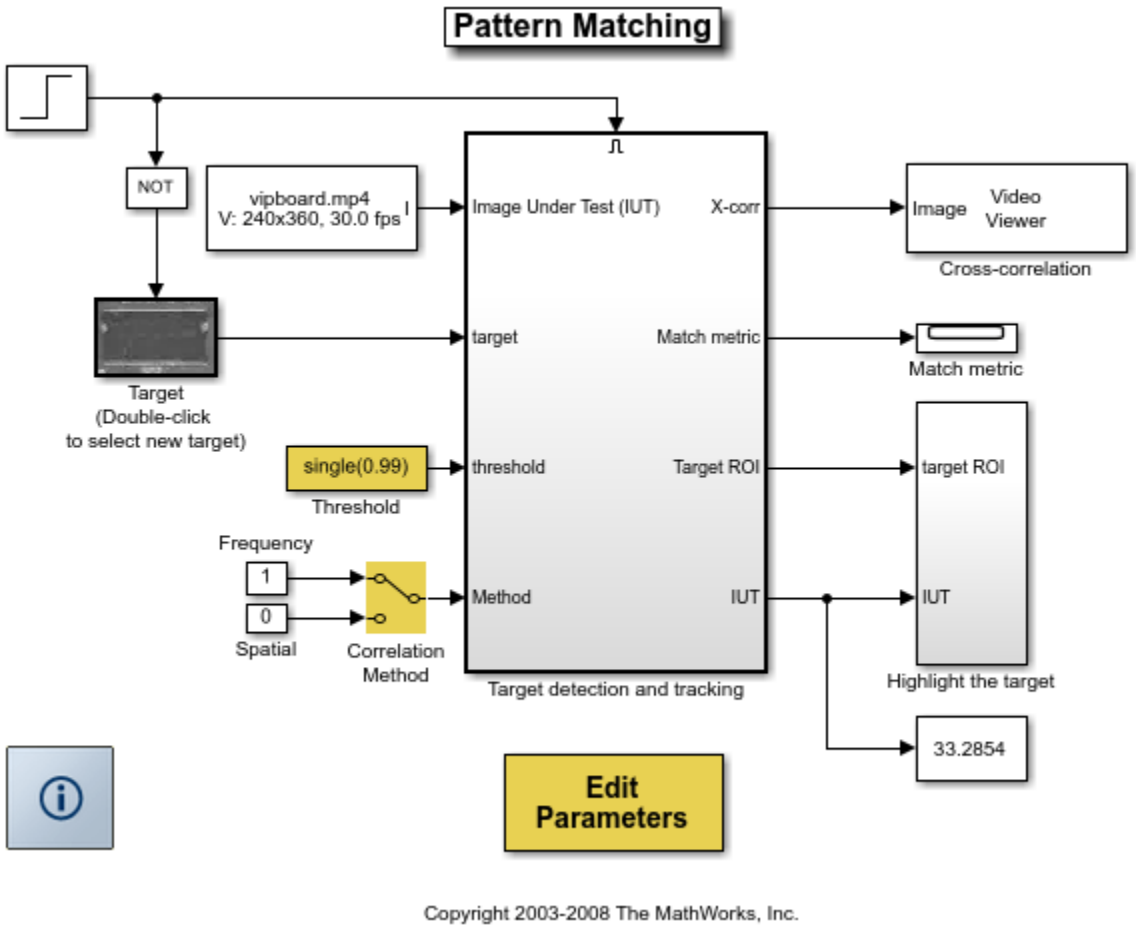
This example shows how to use the 2-D normalized cross-correlation for pattern matching and target tracking.

Double-click the Edit Parameters block to select the number of similar targets to detect. You can also change the pyramiding factor. By increasing it, you can match the target template to each video frame more quickly. Changing the pyramiding factor might require you to change the Threshold value.

Additionally, you can double-click the Correlation Method switch to specify the domain in which to perform the cross-correlation. The relative size of the target to the input video frame and the pyramiding factor determine which domain computation is faster.

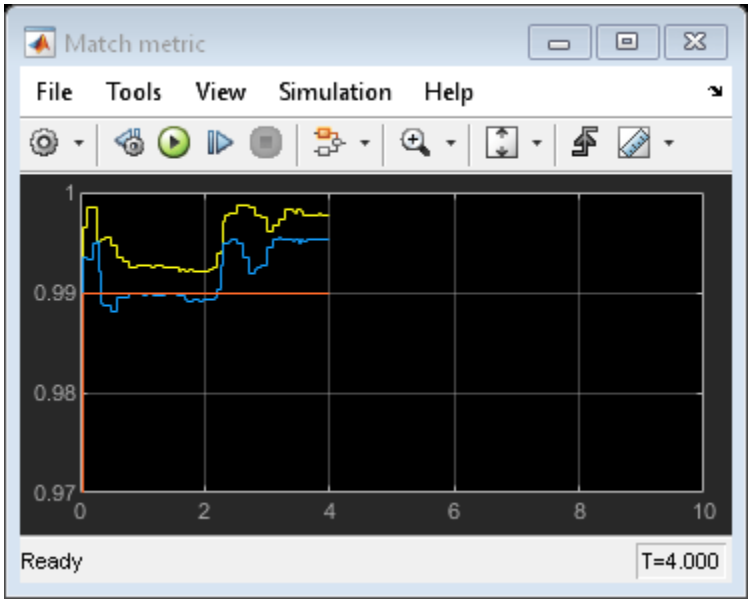
Example Model

The following figure shows the Pattern Matching model:

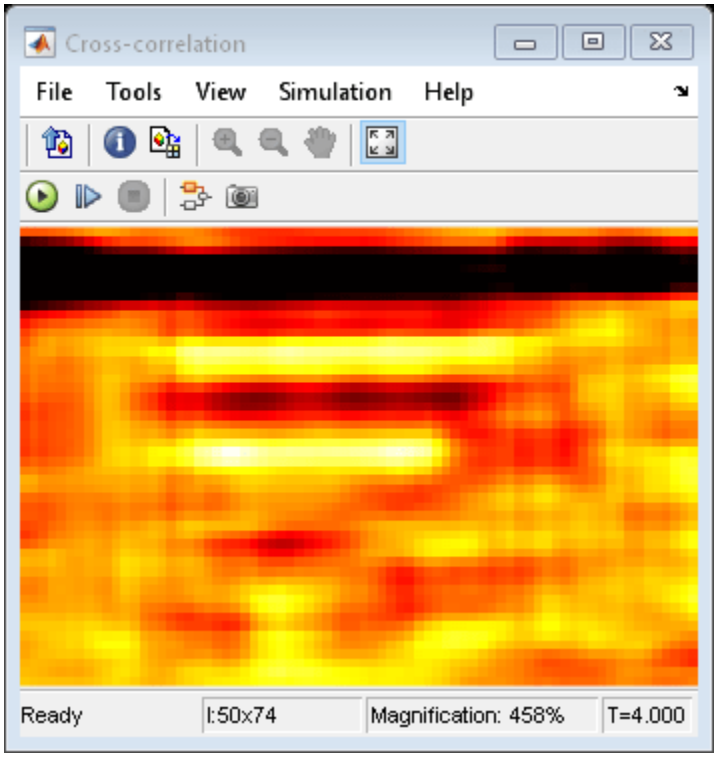


Pattern Matching Results

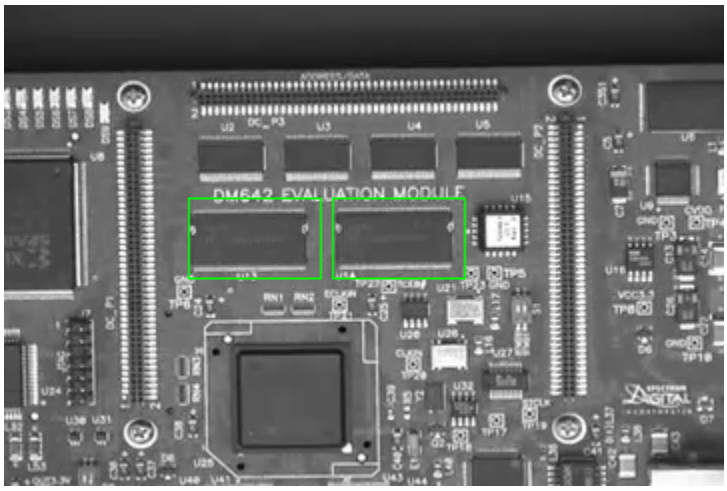
The Match metric window shows the variation of the target match metrics. The model determines that the target template is present in a video frame when the match metric exceeds a threshold (cyan line).



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.

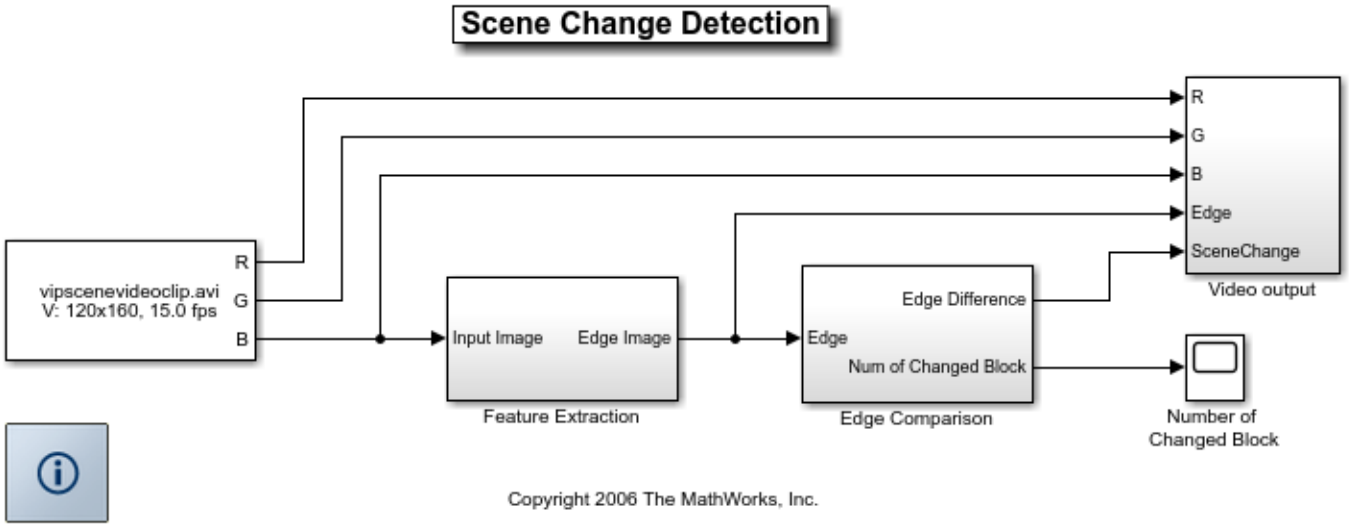


Scene Change Detection

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.

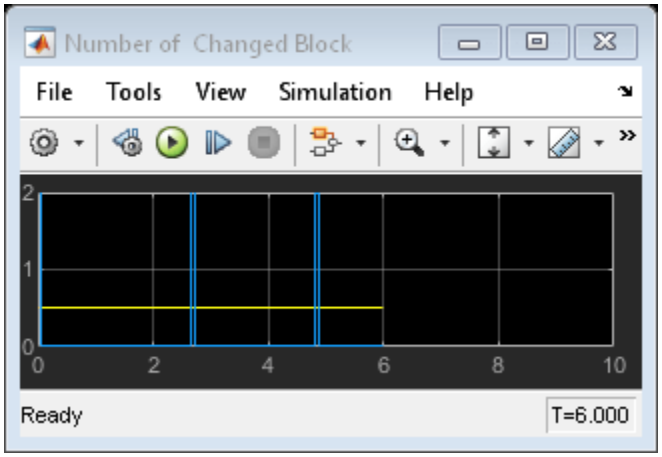
Example Model

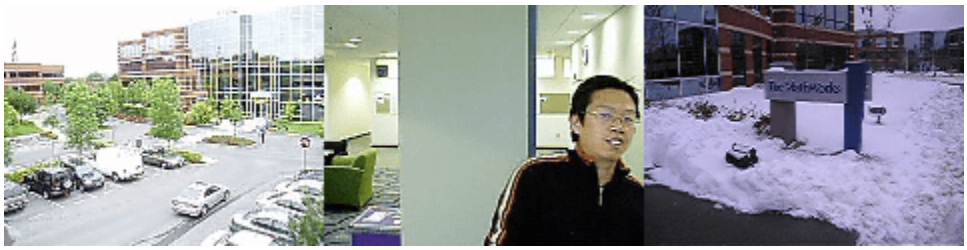
The following figure shows the Scene Change Detection example model:



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.






Frame 90 Shot 3

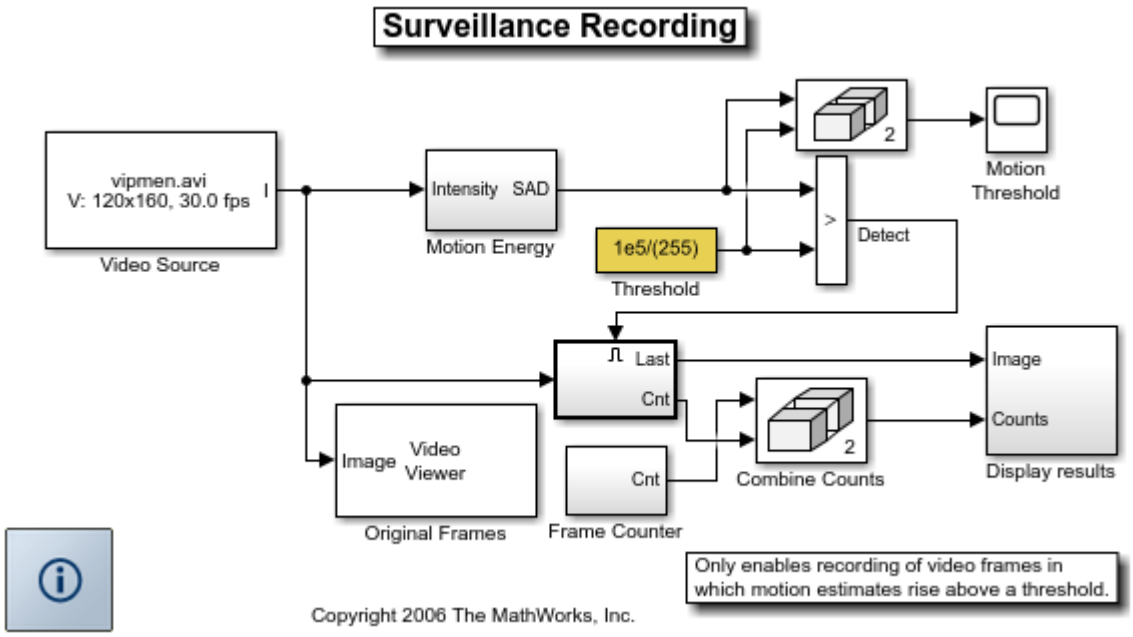
Surveillance Recording

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.

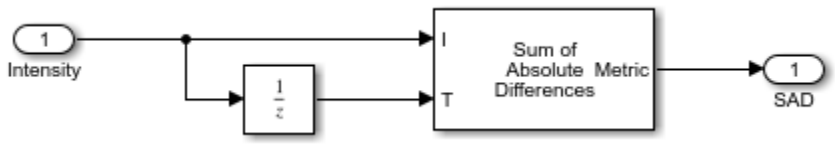
Example Model

The following figure shows the Surveillance Recording model:



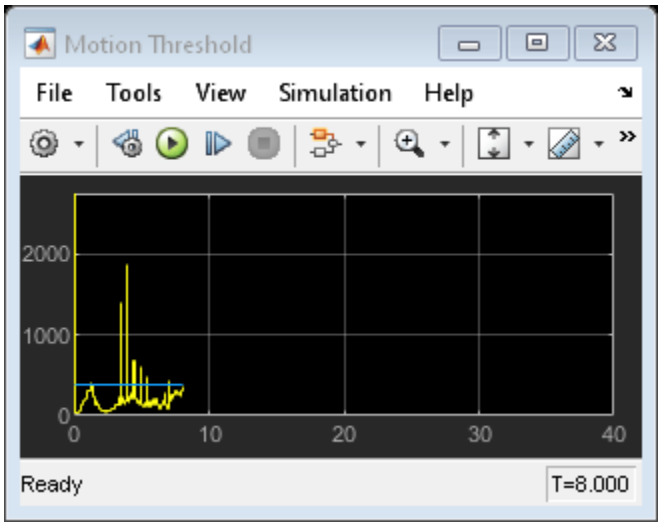
Motion Energy Subsystem

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

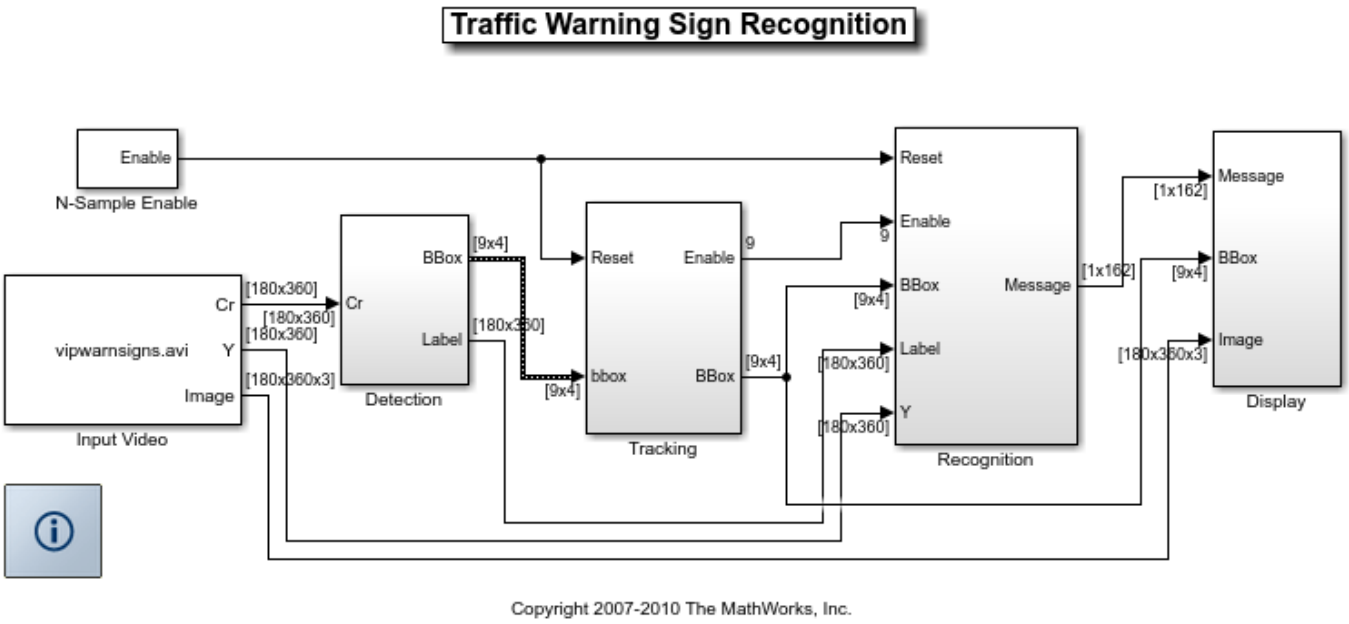
Traffic Warning Sign Recognition

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:



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.

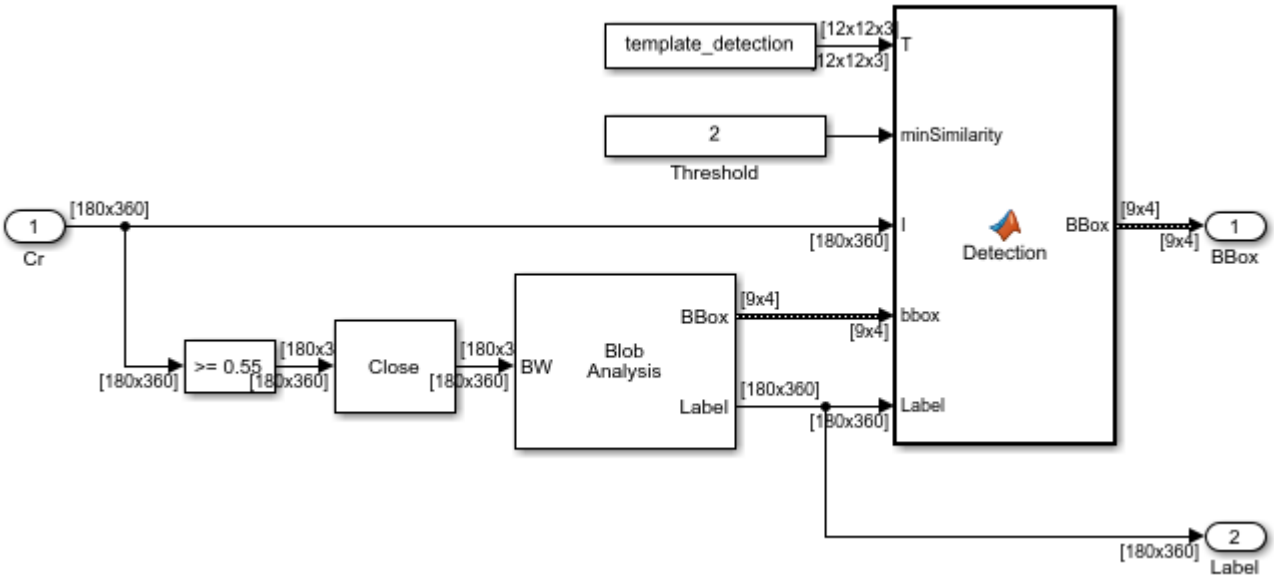
The Detection Templates window shows the traffic warning sign detection templates.

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

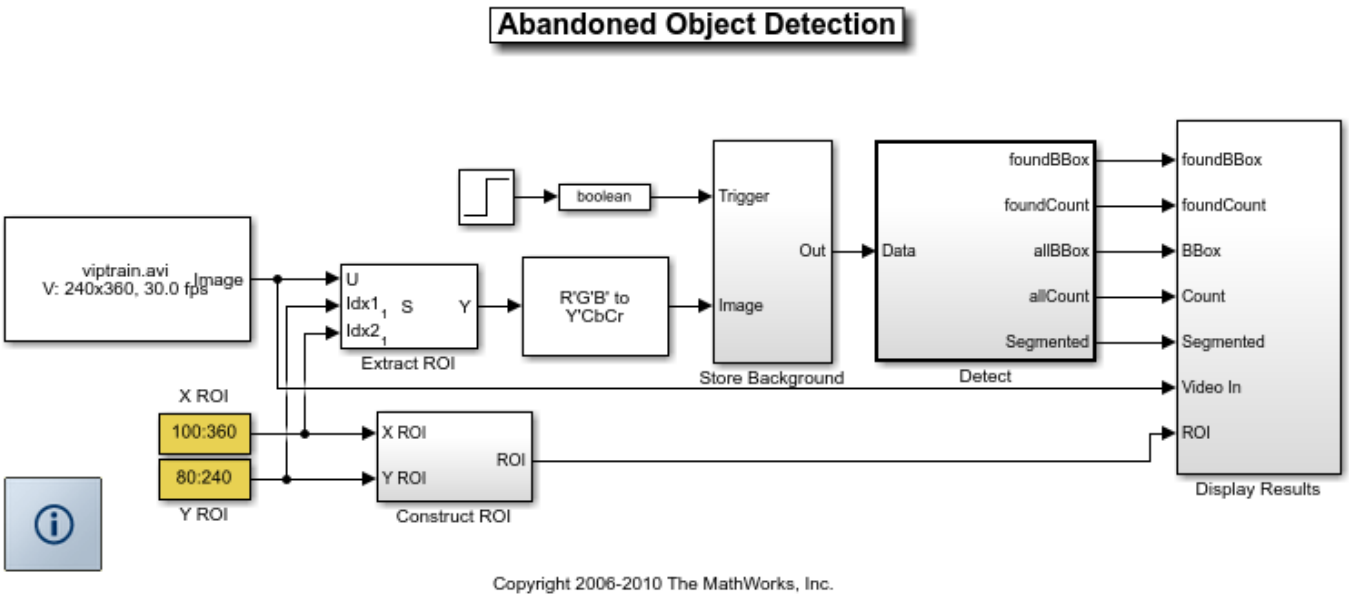
This example shows how to track objects at a train station and to determine which ones remain stationary. Abandoned objects in public areas concern authorities since they might pose a security risk. Algorithms, such as the one used in this example, can be used to assist security officers monitoring live 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

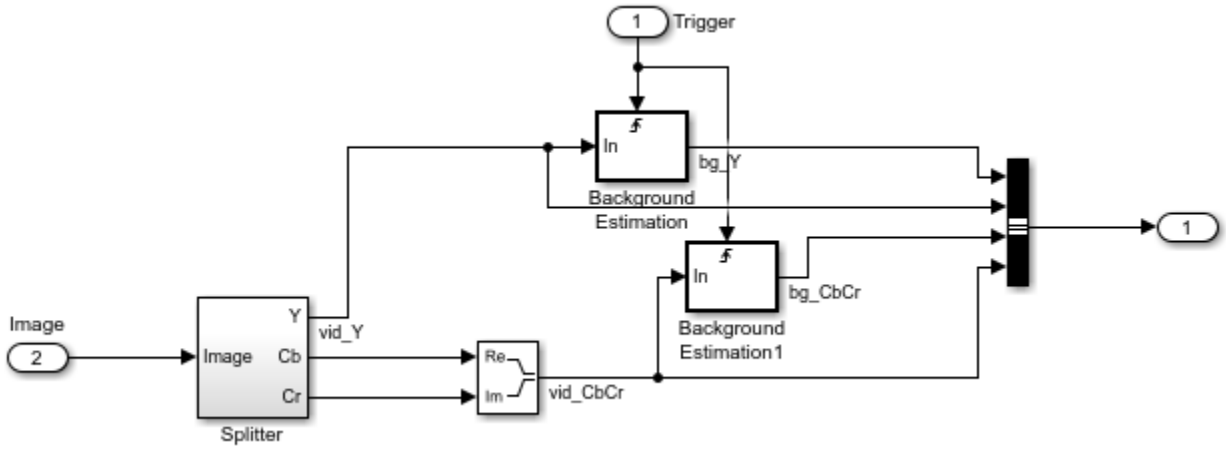
The following figure shows the Abandoned Object Detection example model.



Store Background Subsystem

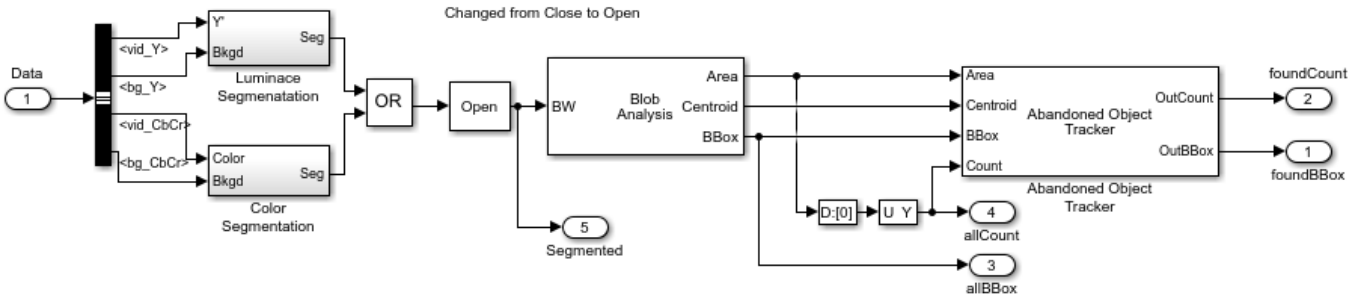
This example uses the first frame of the video as the background. To improve accuracy, the example uses both intensity and color information for the background subtraction operation. During this operation, Cb and Cr color channels are stored in a complex array.

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

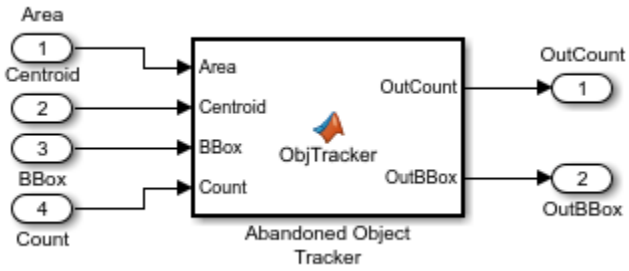


Detect Subsystem

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.

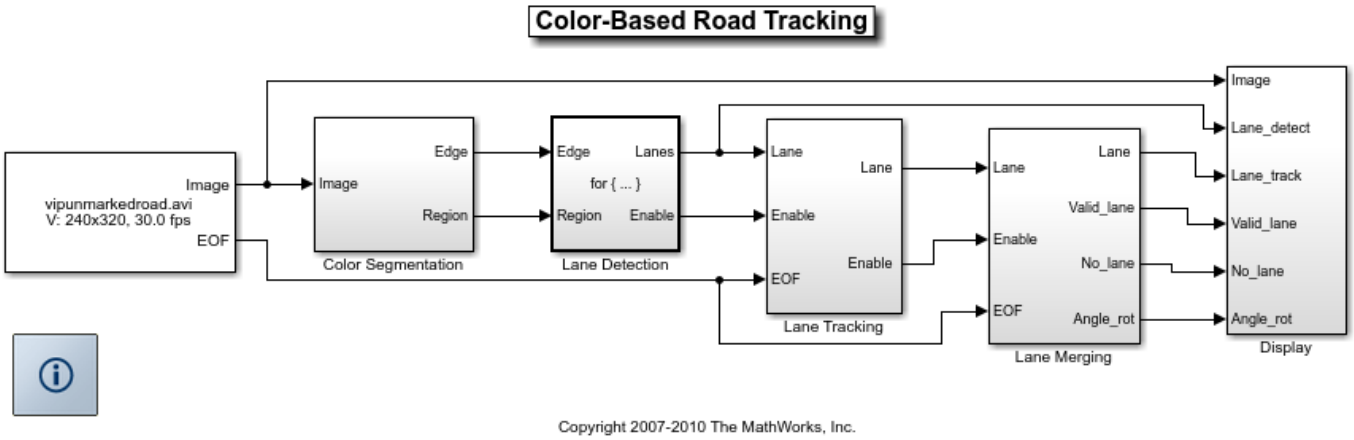
The Abandoned Objects window highlights the abandoned objects with a red box.

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.

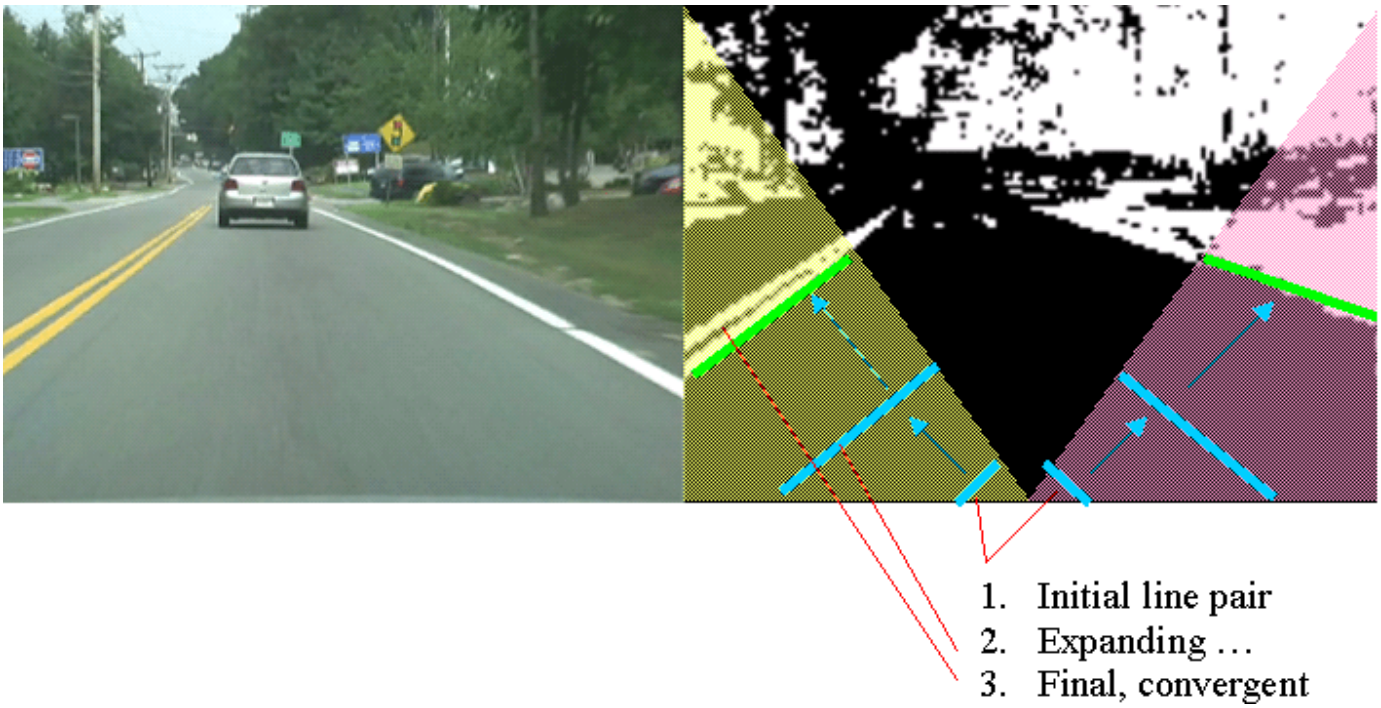
Example Model

The following figure shows the Color-based Road Tracking model:



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.



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

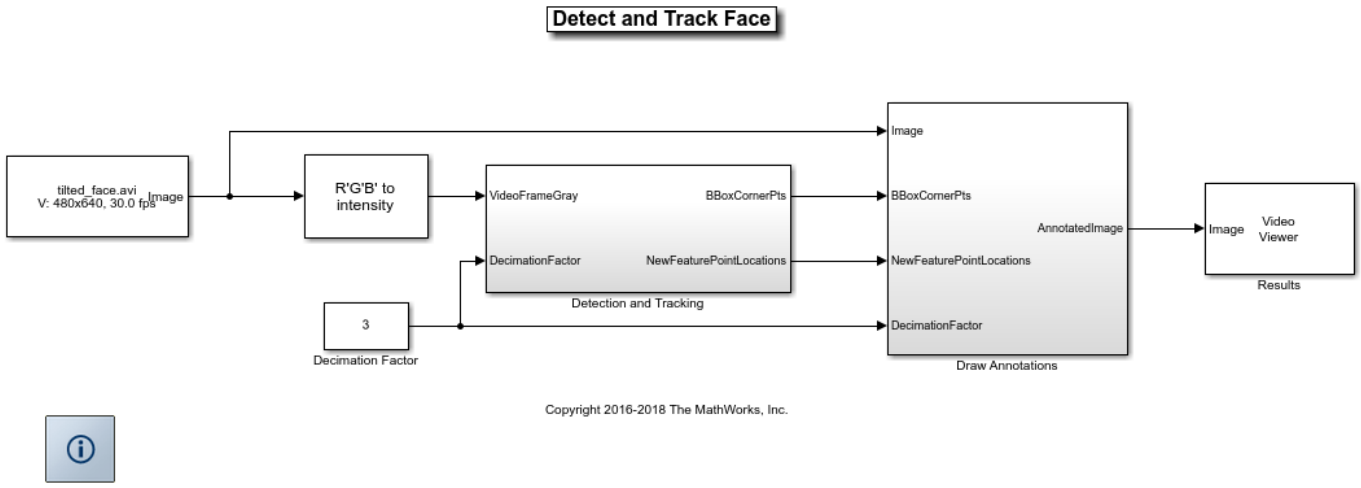
This example shows how to implement a face detection and tracking algorithm in Simulink® by using a MATLAB® Function block. It closely follows the “Face Detection and Tracking Using the KLT Algorithm” on page 8-20 MATLAB® example.

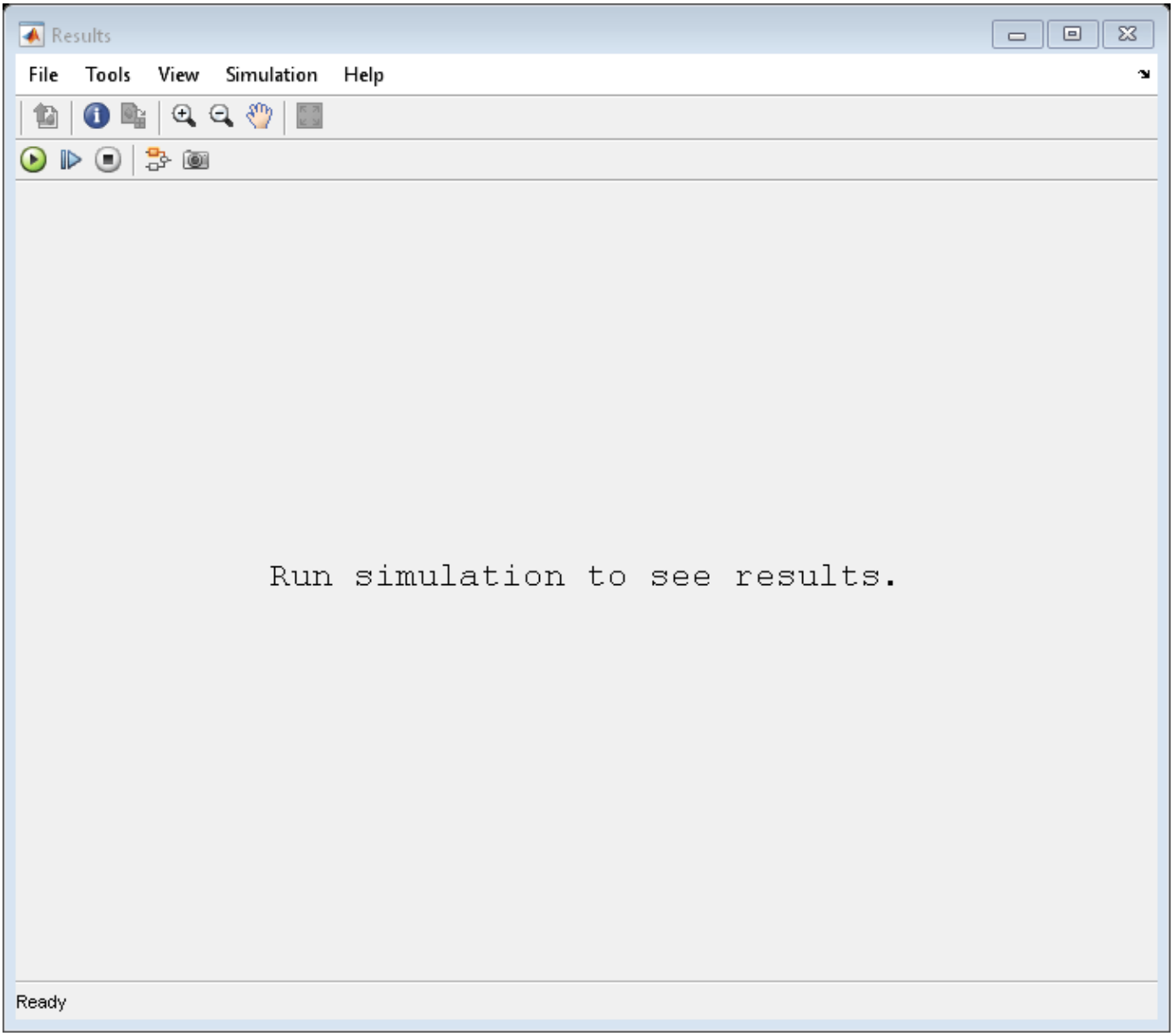
Introduction

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');
```





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

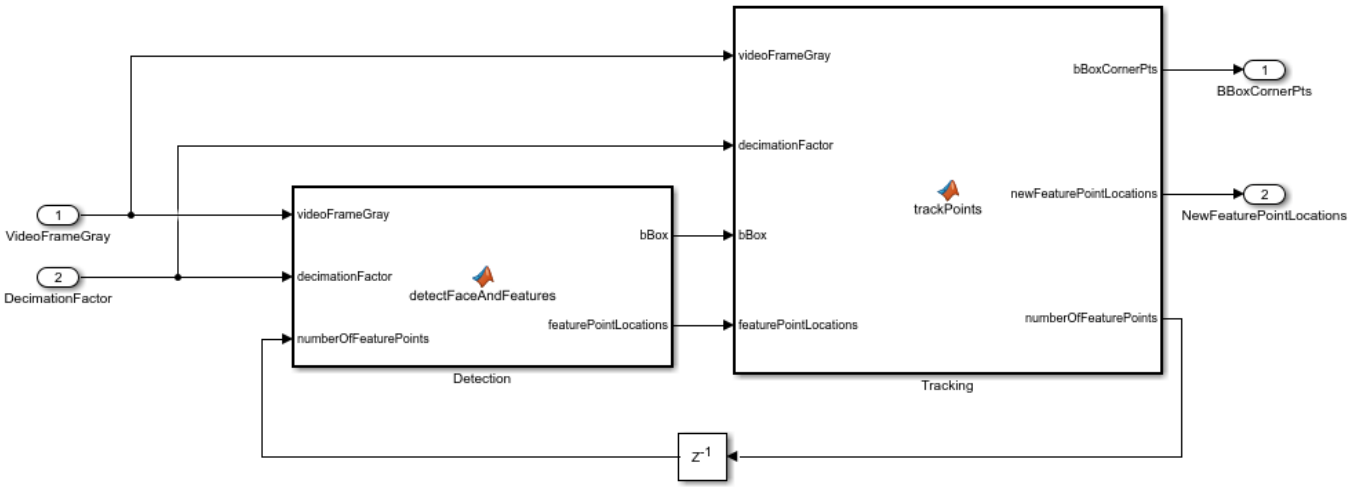
In this example, the `vision.CascadeObjectDetector` System object™ detects the location of the face in the current video frame. The cascade object detector uses the Viola-Jones detection algorithm and a trained classification model for detection. After the face is detected, facial feature points are 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.

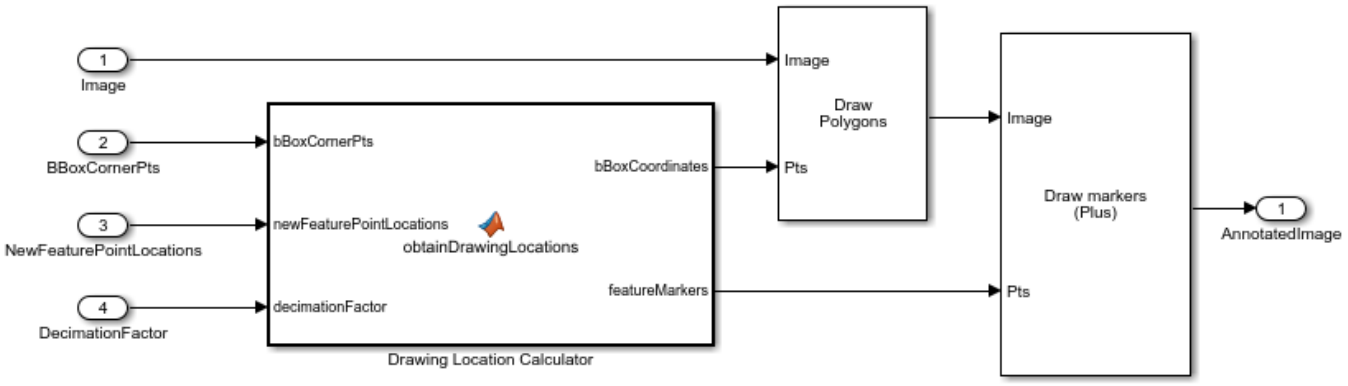
```
open_system('DetectAndTrackFace/Detection and Tracking')
```



Draw Annotations

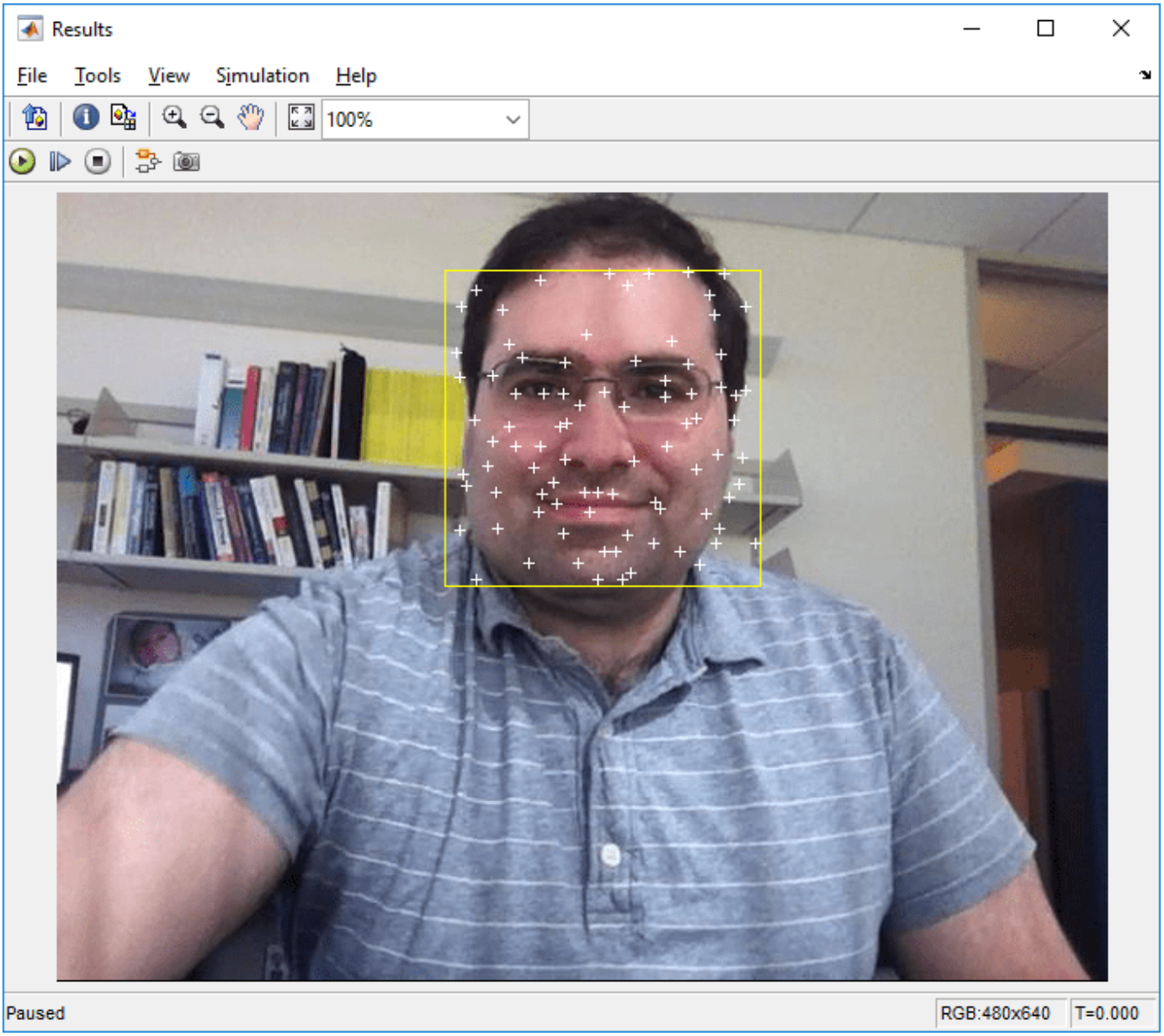
The bounding box corner points and feature point locations are used to draw on the output video frame. The Draw Shapes block draws the bounding box. The feature points are drawn using the Draw Markers block.

```
open_system('DetectAndTrackFace/Draw Annotations')
```

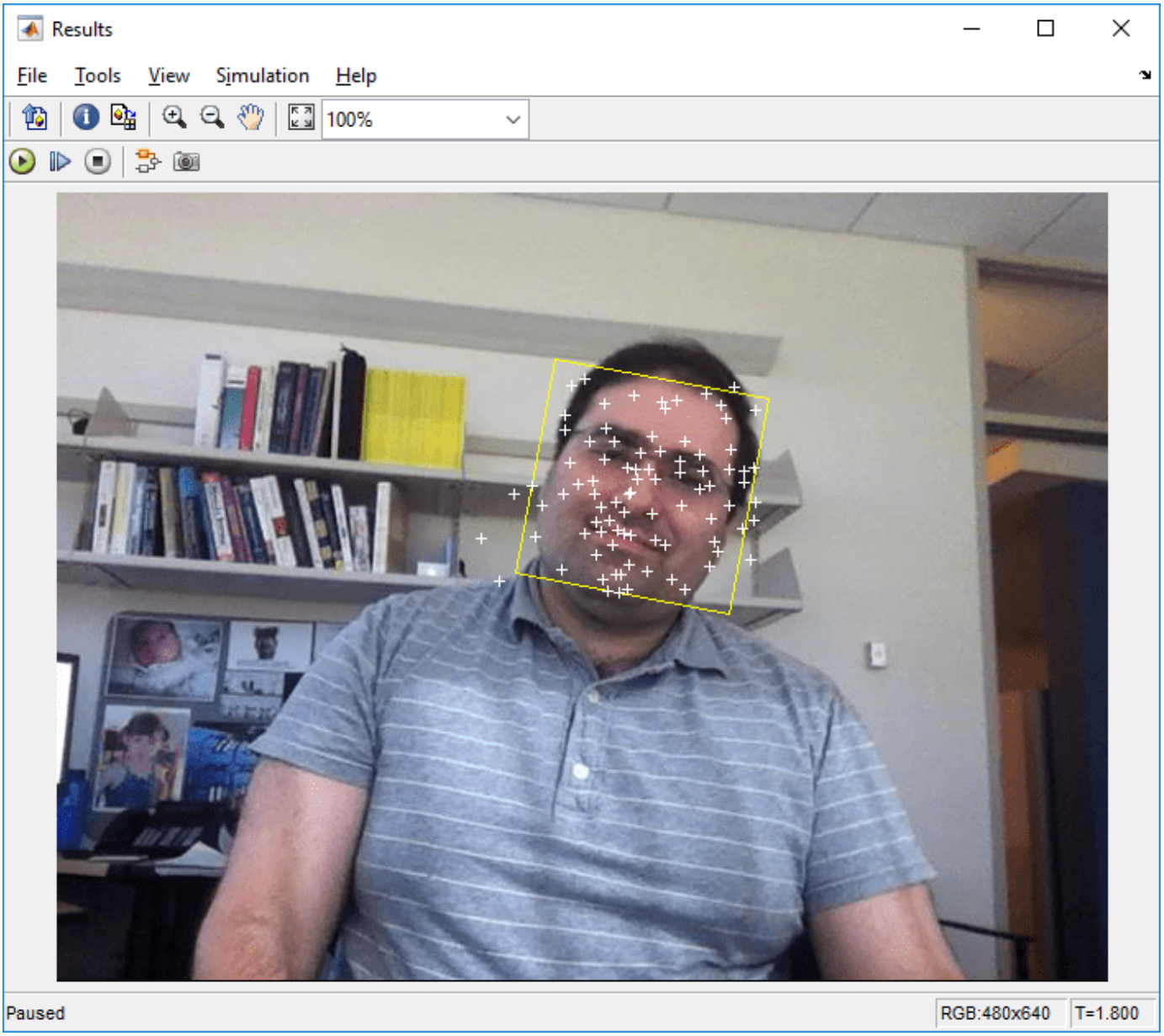


Results

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.

ZKalal, 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.

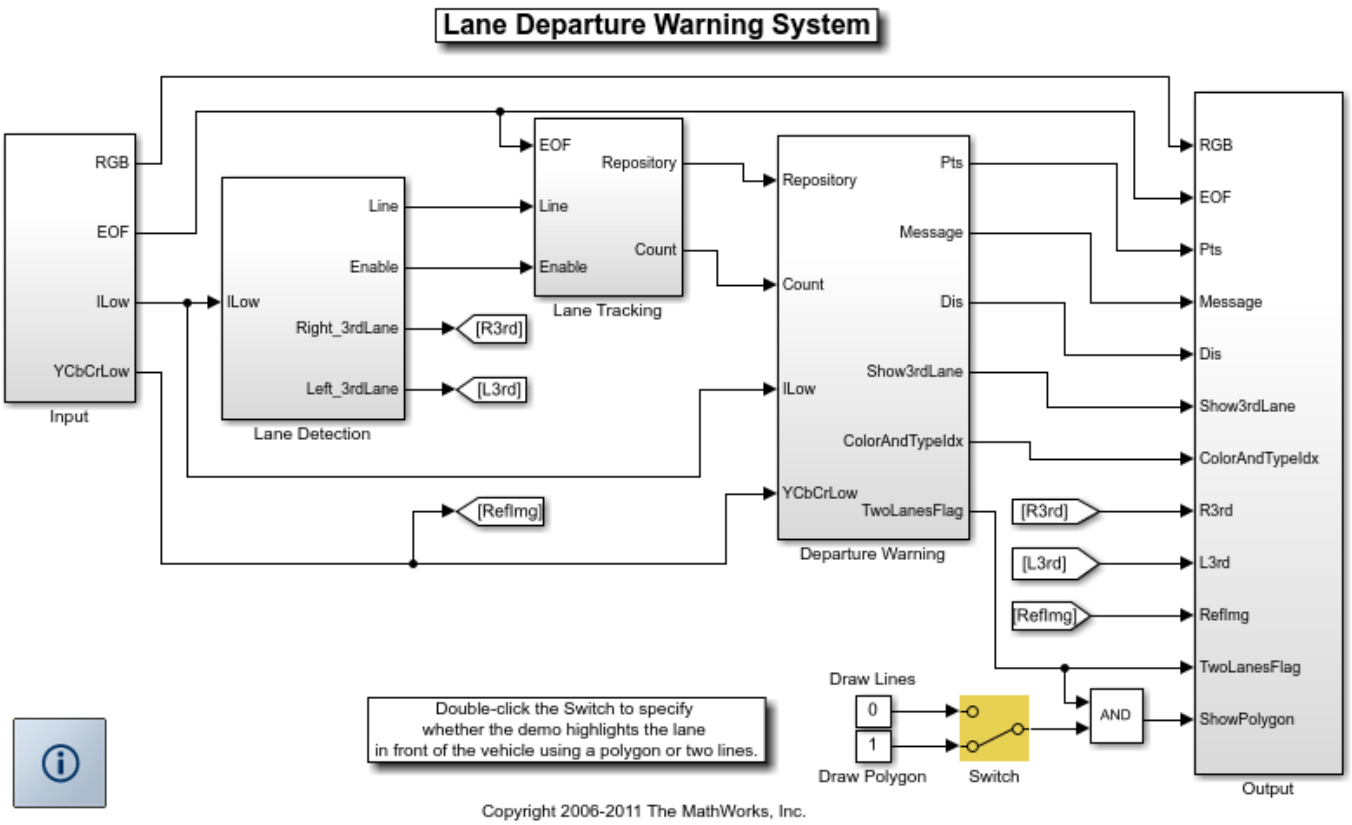
To process low quality video sequences, where lane markers might be difficult to see or are hidden behind objects, the example waits for a lane marker to appear in multiple frames before it considers the marker to be valid. The example uses the same process to decide when to begin to ignore a lane marker.

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.

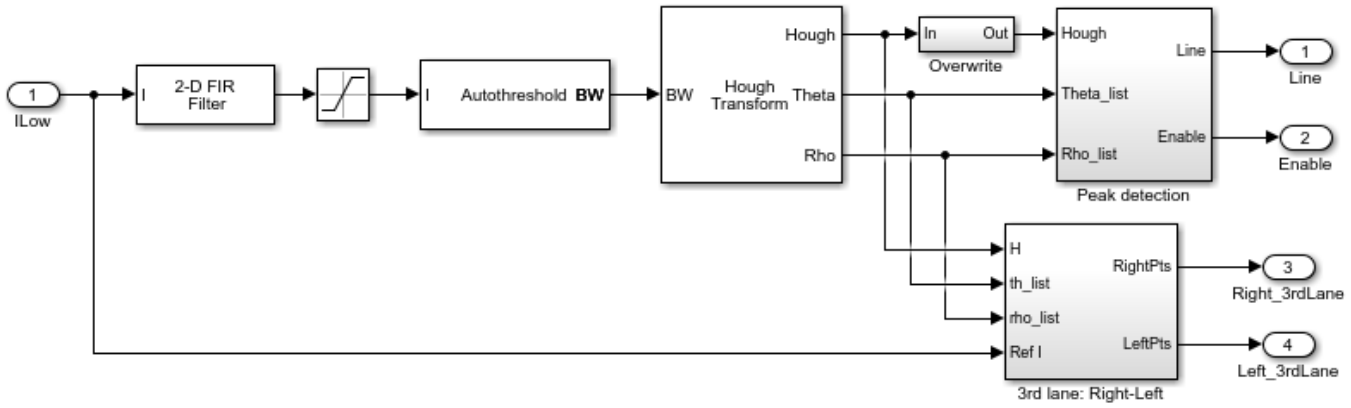
Example Model

The following figure shows the Lane Departure Warning System example model:



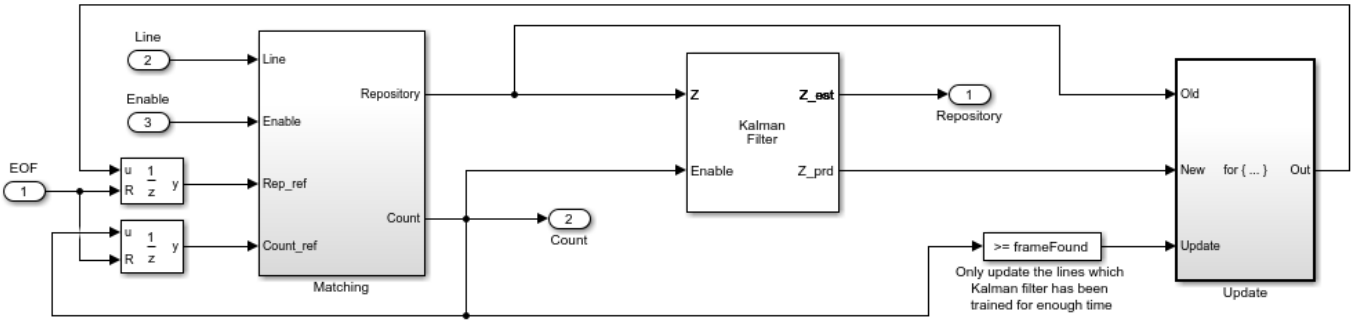
Lane Detection Subsystem

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.



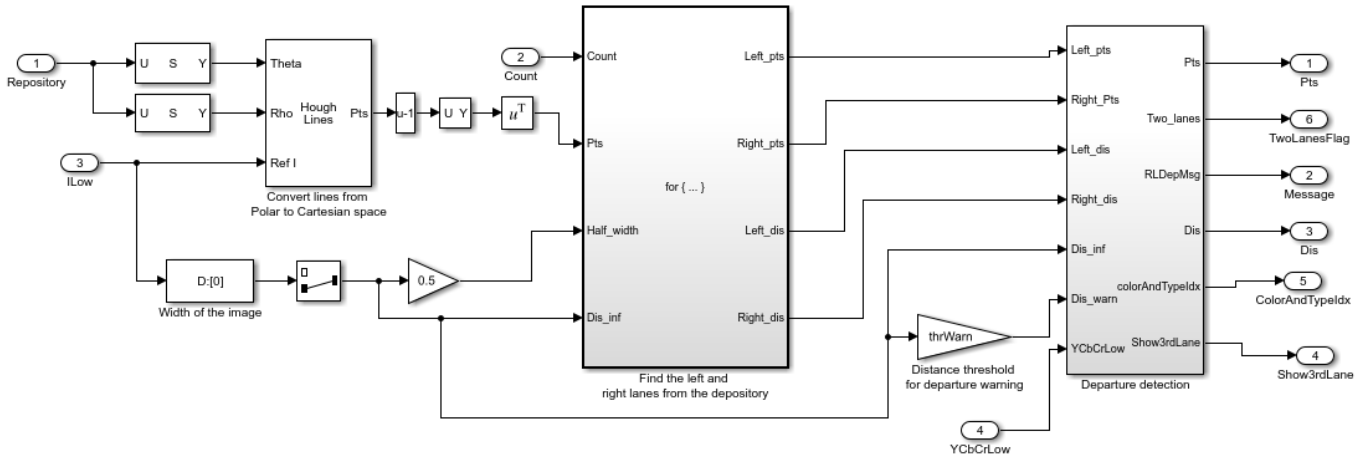
Lane Tracking Subsystem

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.



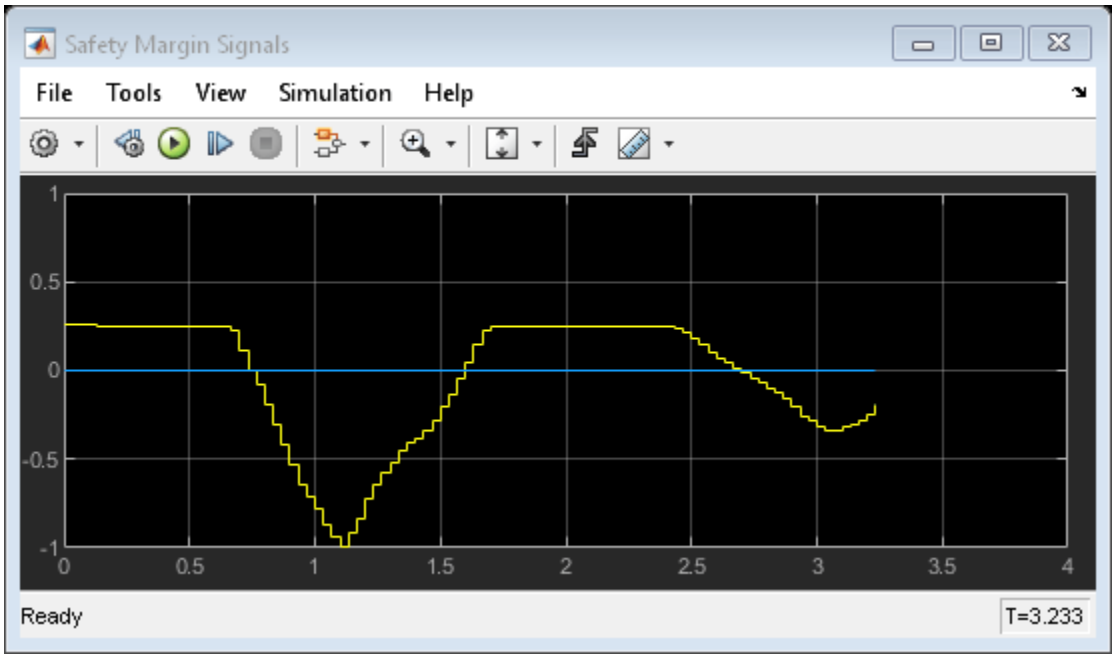
Departure Warning Subsystem

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.

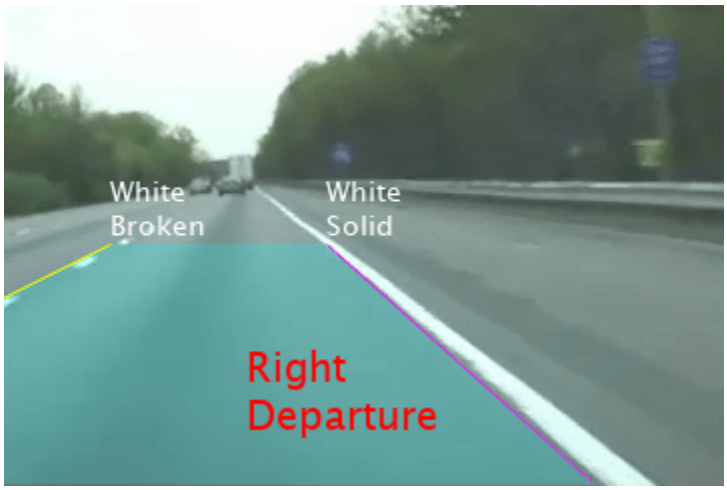


Lane Departure Warning System Results

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.

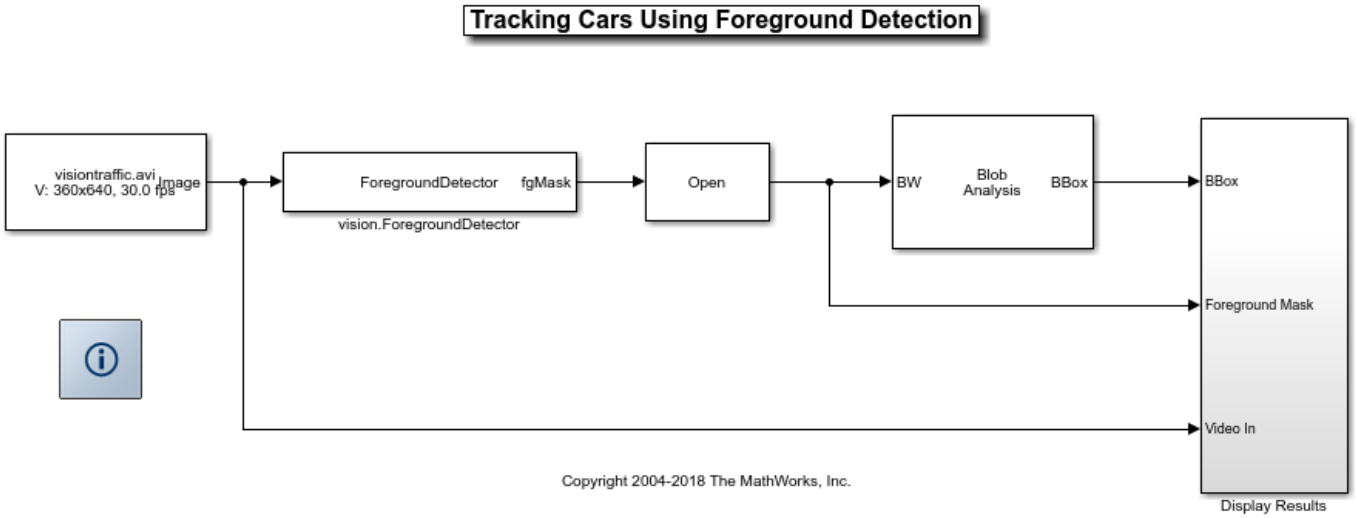

White
Broken
White
Solid
Right
Departure

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:



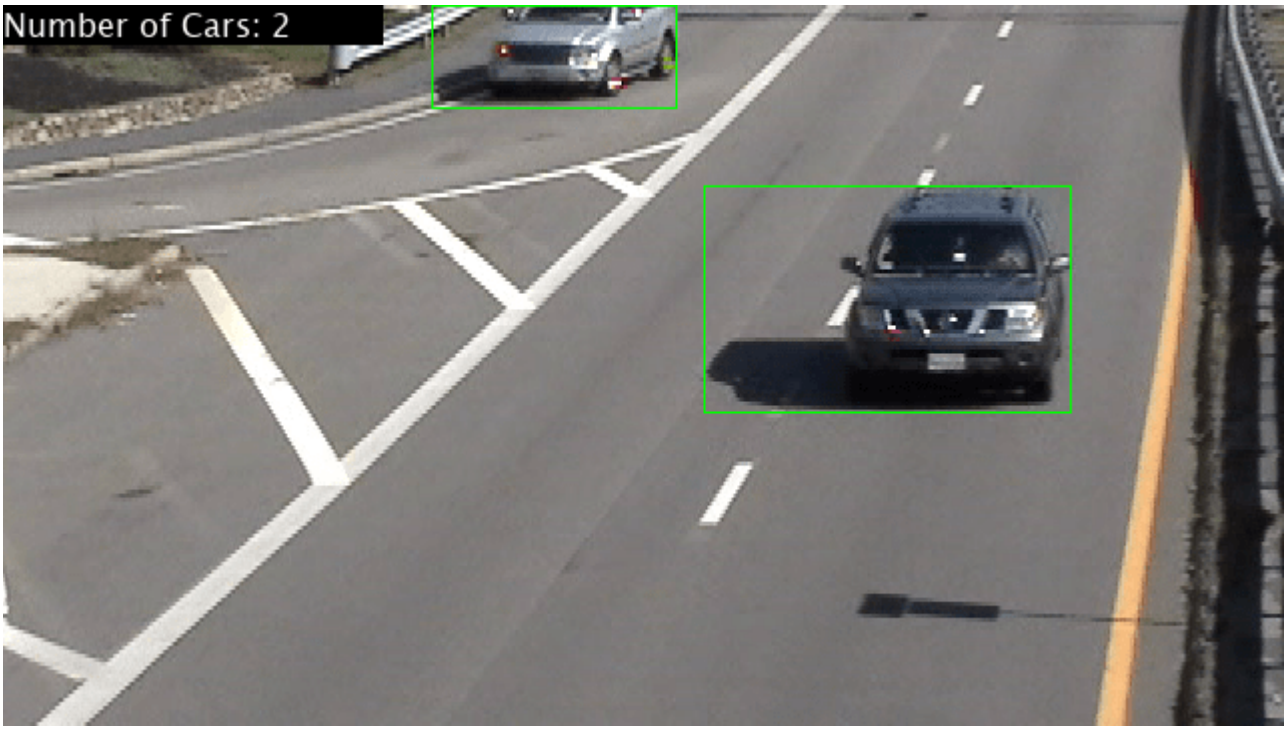
Detection and Tracking Results

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

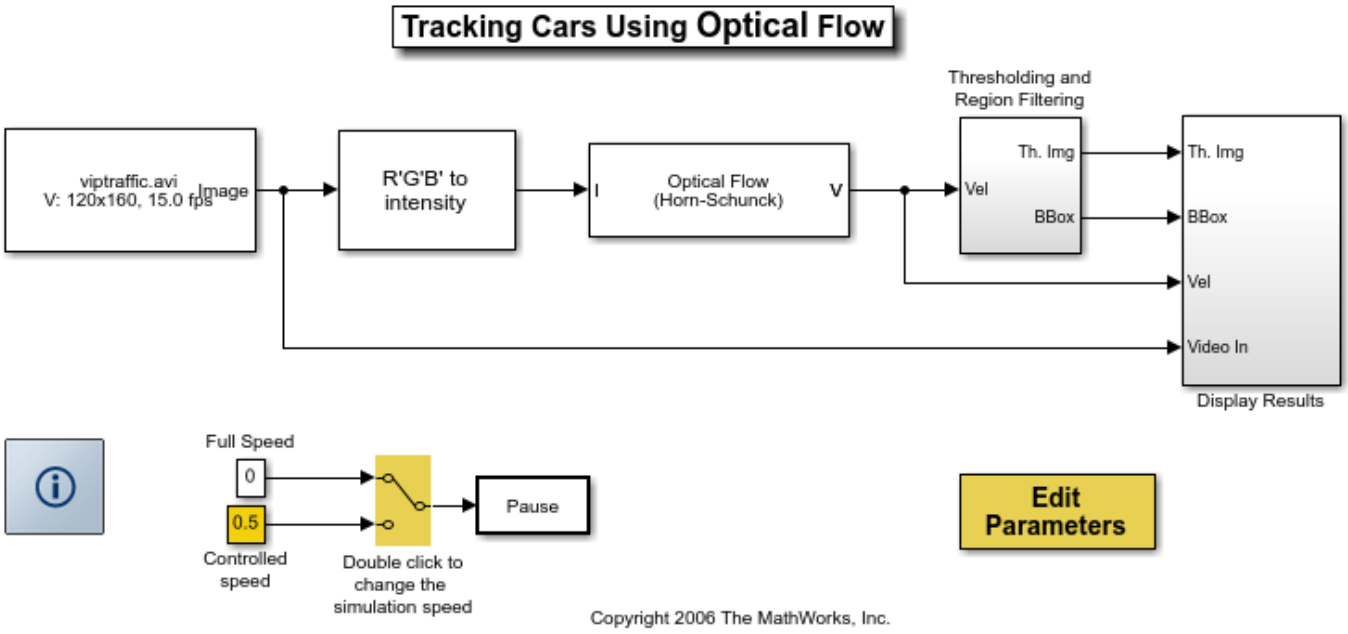
The algorithm in this example is suitable for an embedded software implementation. You can deploy it to an ARM™ processor using a Xilinx™ Zynq™ video processing reference design. See “Tracking Cars with Zynq-Based Hardware” (Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware).

Tracking Cars Using Optical Flow

This example shows how to detect and track cars in a video sequence using optical flow estimation.

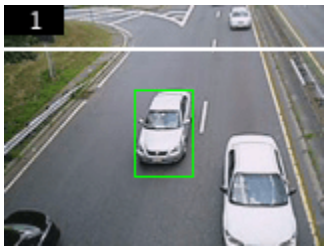
Example Model

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 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.

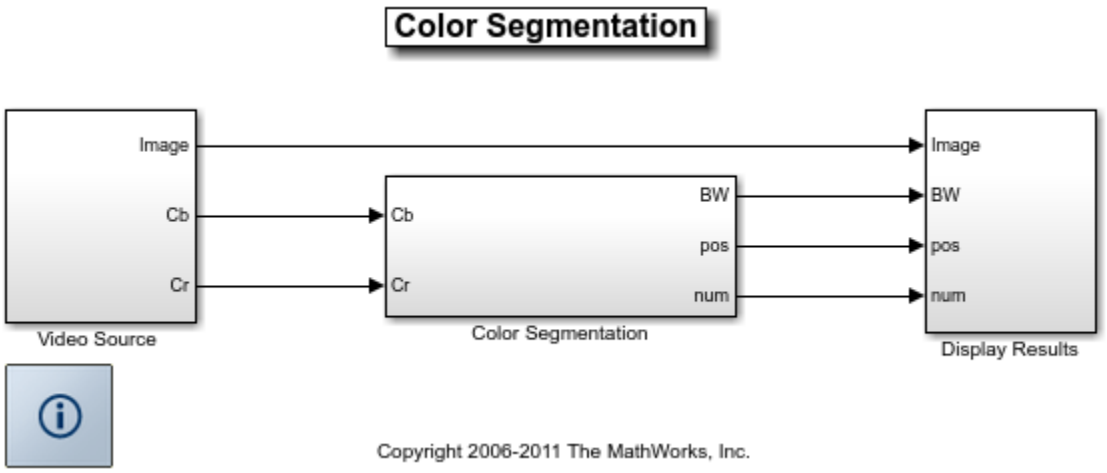

1

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

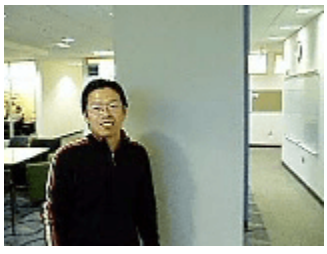
To create an accurate color model for the example, many images containing skin color samples were processed to compute the mean (m) and covariance (C) of the Cb and Cr color channels. Using this color model, the Color Segmentation/Color Classifier subsystem classifies each pixel as either skin or nonskin by computing the square of the Mahalanobis distance and comparing it to a threshold. The equation for the Mahalanobis distance is shown below:

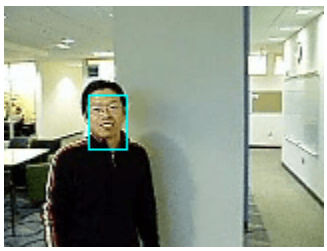
SquaredDistance(Cb,Cr) = (x-m)'*inv(C)*(x-m), where x=[Cb; Cr]

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.





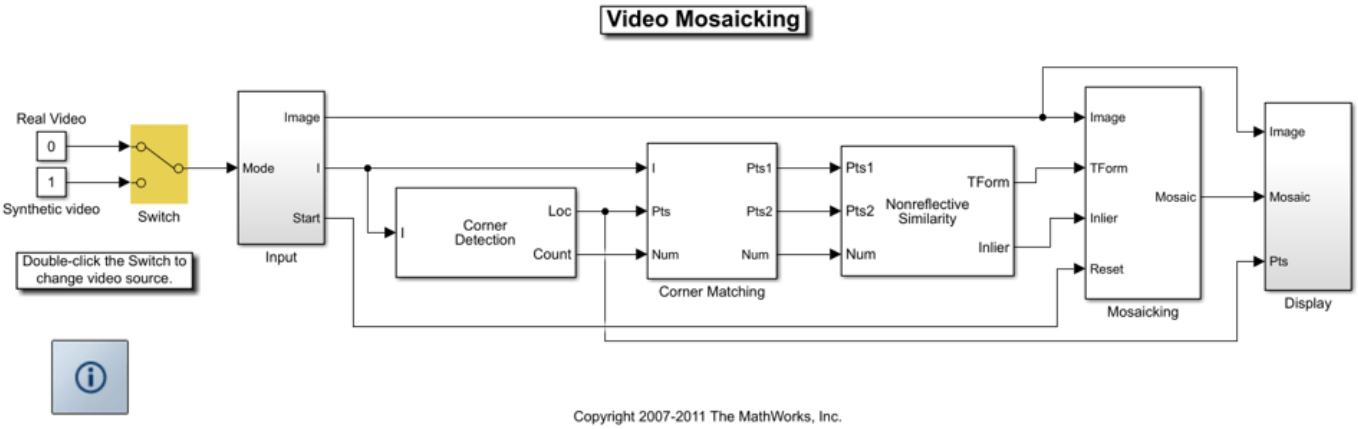
Video Mosaicking

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

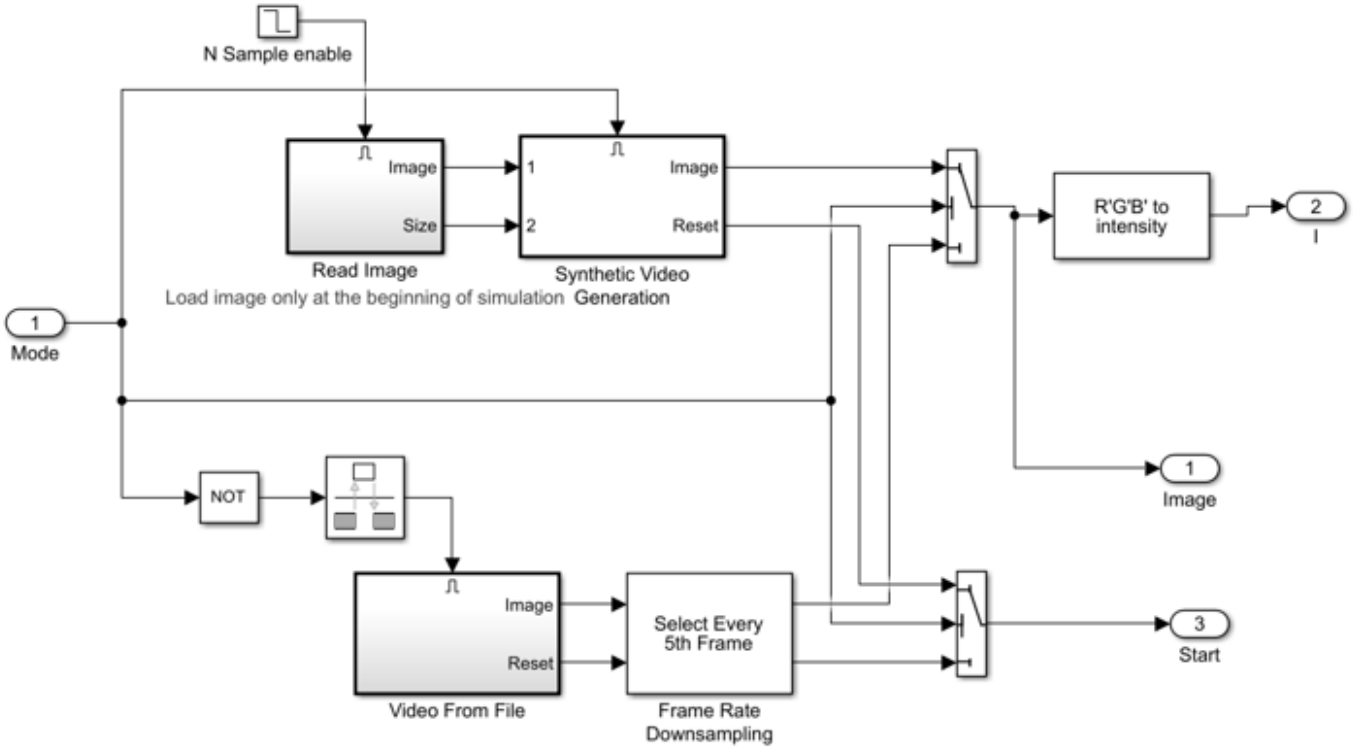
The following figure shows the Video Mosaicking model:



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.

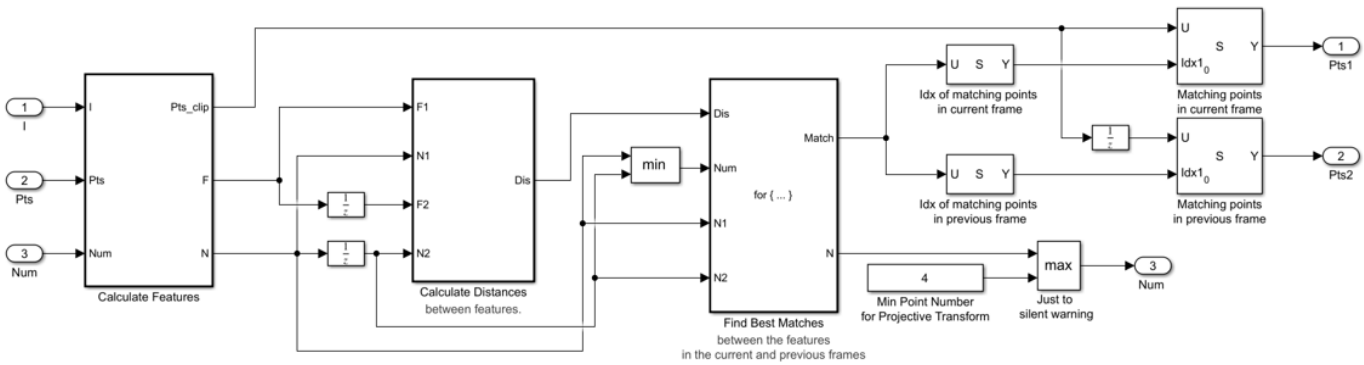


If you choose to use a video sequence from a file, you can reduce computation time by processing only some of the video frames. This is done by setting the downsampling rate in the Frame Rate Downsampling subsystem.

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

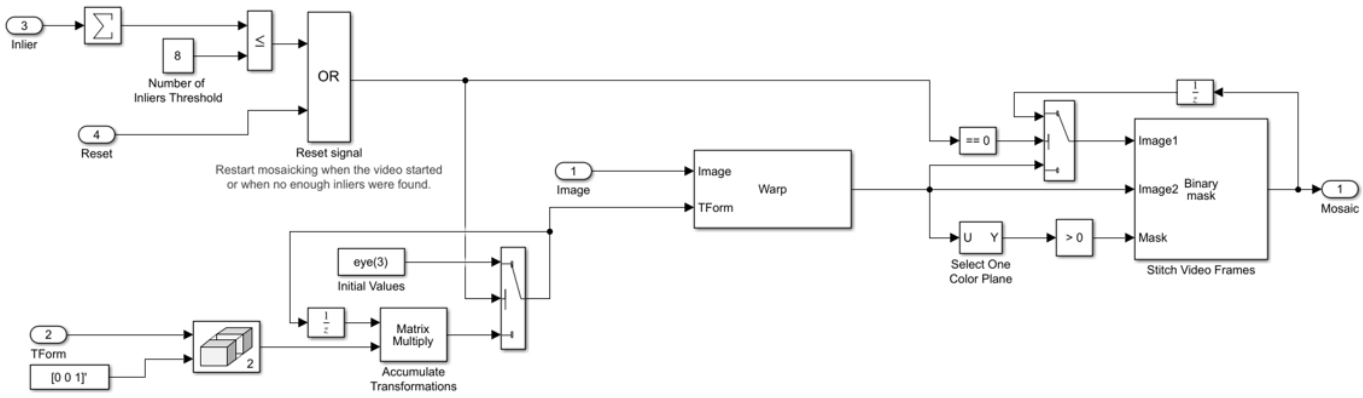
The subsystem finds corner features in the current video frame in one of three methods. The example uses Local intensity comparison (Rosen & Drummond), which is the fastest method. The other methods available are the Harris corner detection (Harris & Stephens) and the Minimum Eigenvalue (Shi & Tomasi).



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.

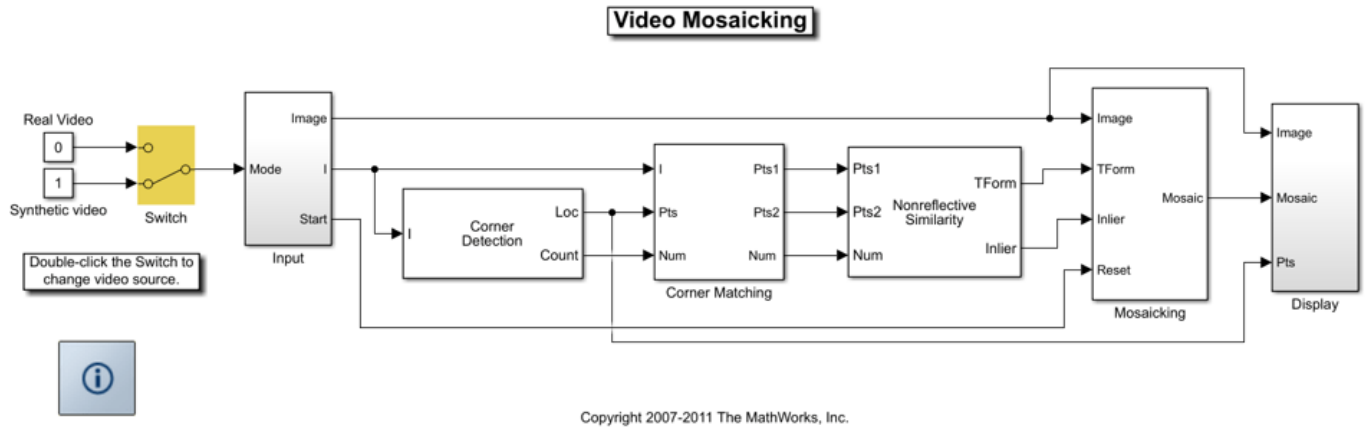
Mosaicking Subsystem

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

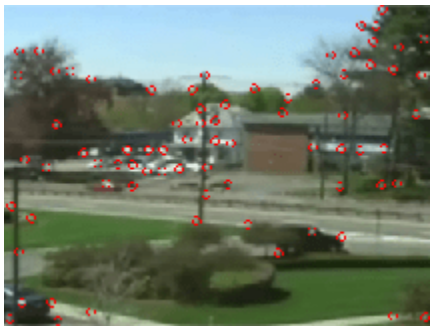


The Corners window shows the corner locations in the current video frame.

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.



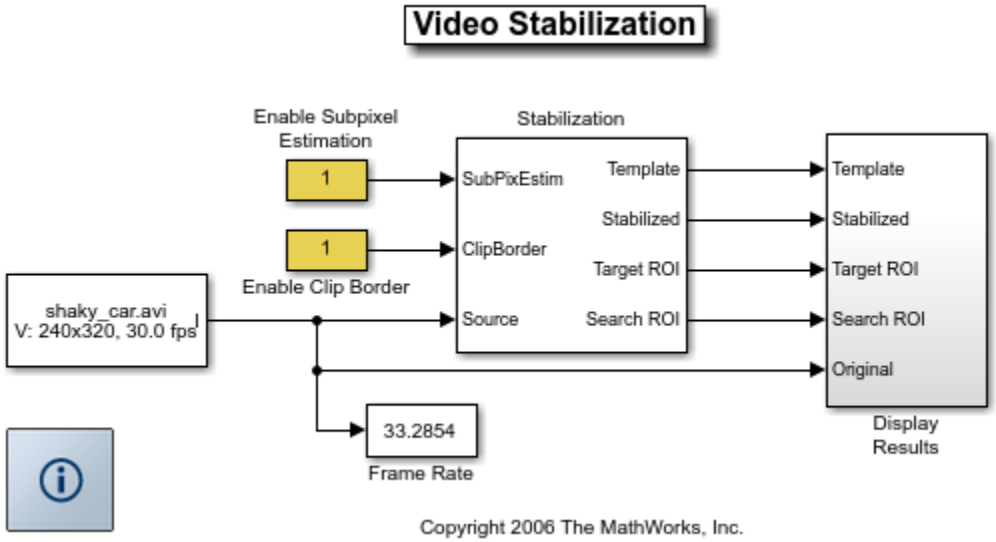
The Mosaic window shows the resulting mosaic image.

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.

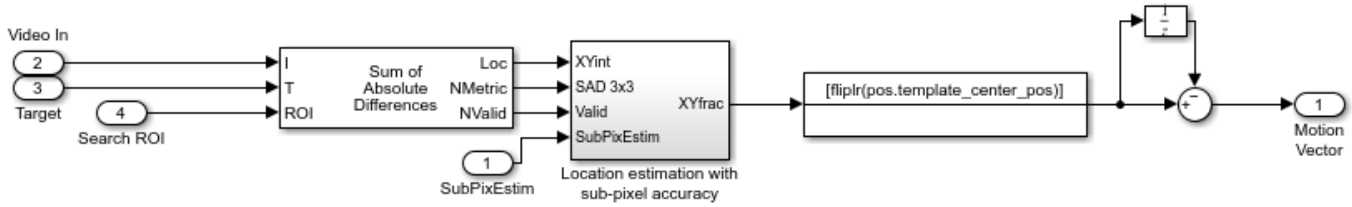
Example Model

The following figure shows the Video Stabilization model:



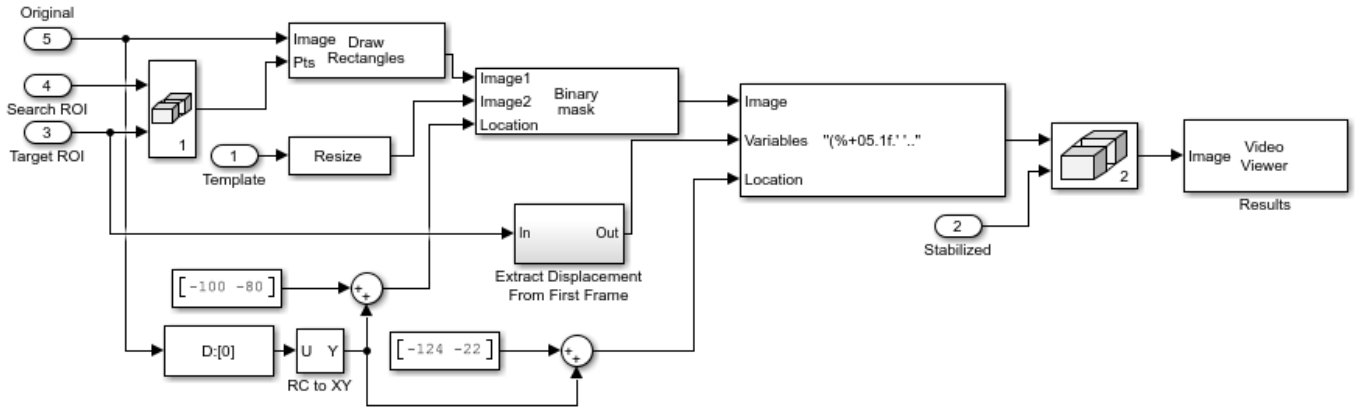
Estimate Motion Subsystem

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

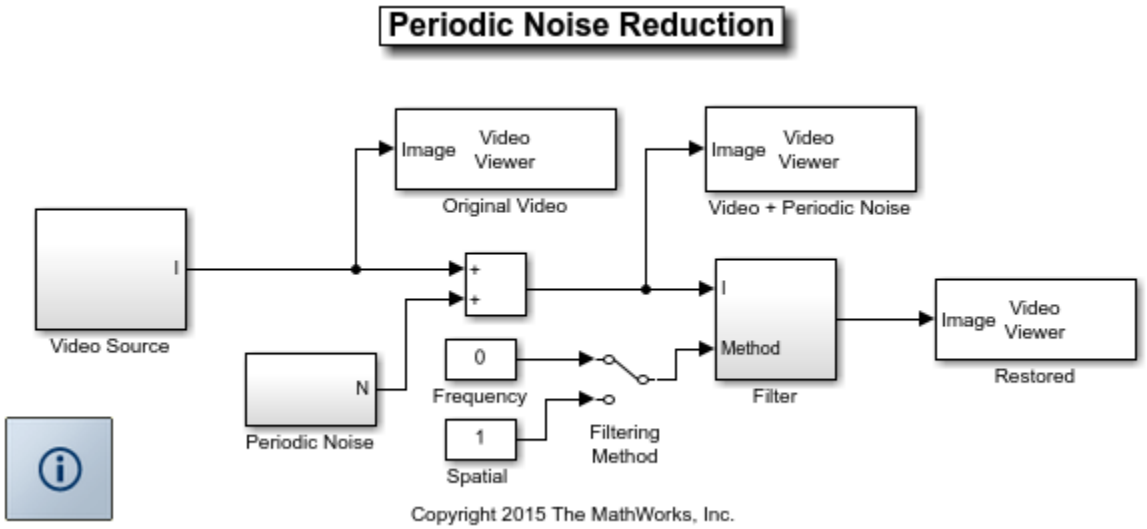
Fixed-point version of this example that simulates row major data organization: vipstabilize_fixpt_rowmajor.slx

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

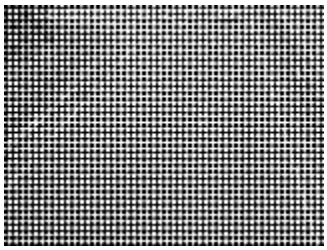
The following figure shows the Periodic Noise Reduction example model:



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.

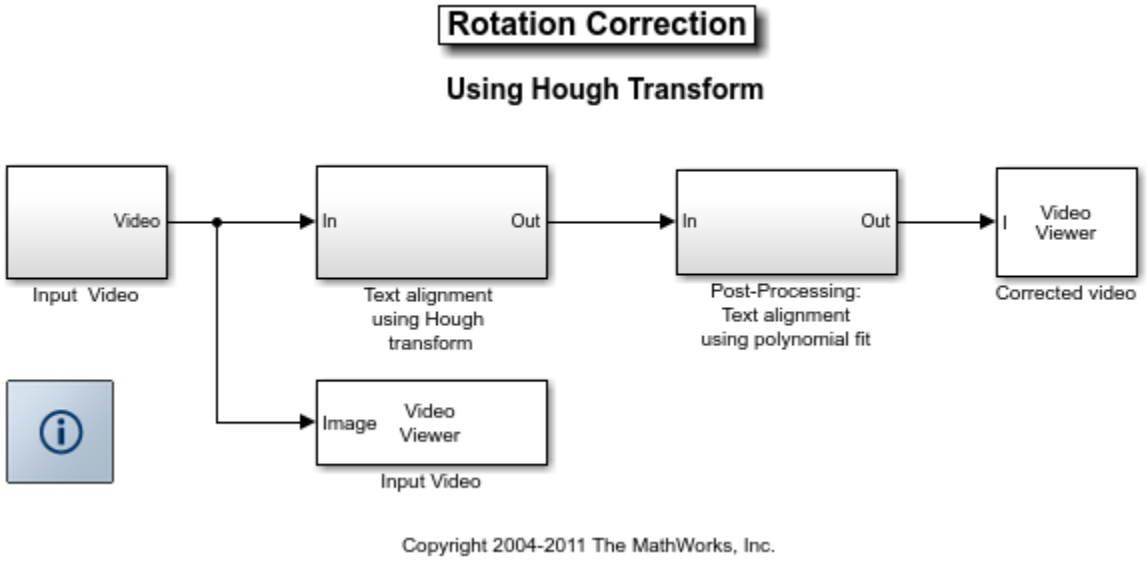


Rotation Correction

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

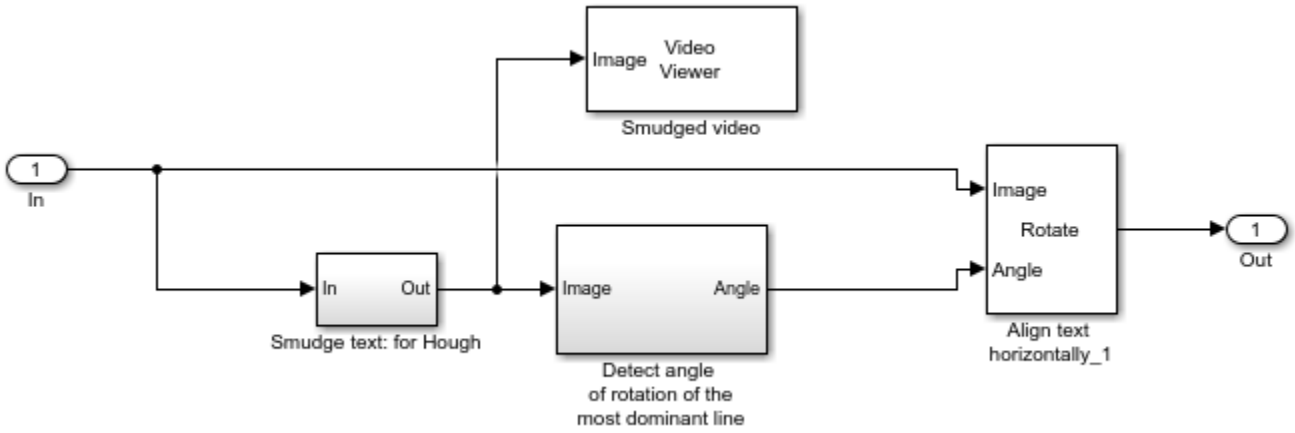
The following figure shows the Rotation Correction example model:



Text Alignment Using Hough Transform Subsystem

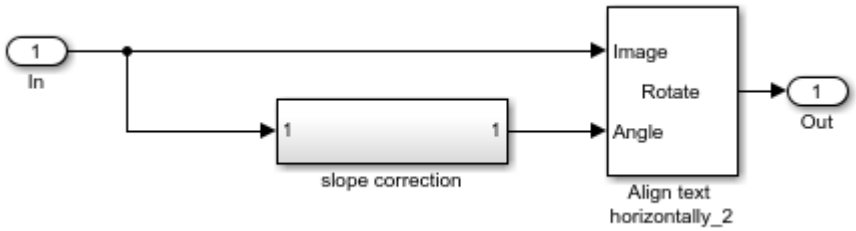
The morphological operators in the Smudge text subsystem blur the letters to create a binary image with two distinct lines. You can see the result of this process in the Smudged Video window.

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.



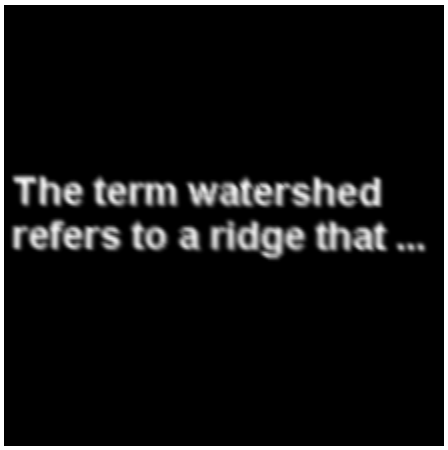
Post-Processing: Text Alignment Using Polynomial Fit Subsystem

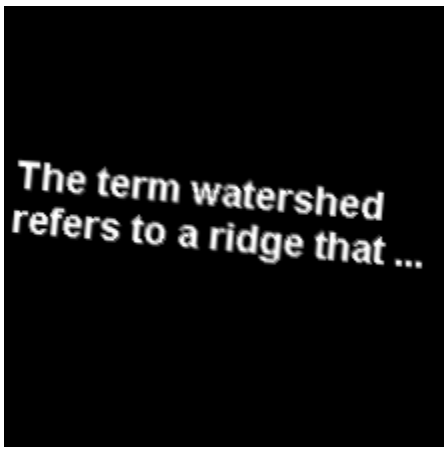
The example uses the Polyfit block, in the slope correction subsystem, and the Rotate block to eliminate small angular variations in the text. The Polyfit block fits a straight line to the smudged text. Then the slope correction subsystem calculates the slope of the line and its angle of inclination. The Rotate block uses this angle to correct for the small rotations.

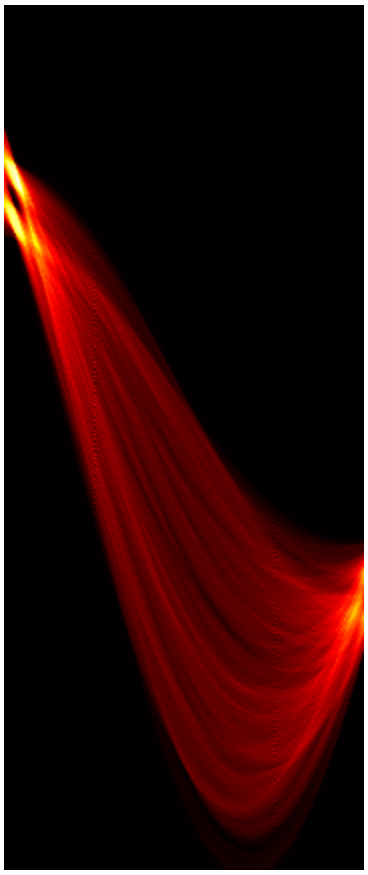


Rotation Correction Results

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.

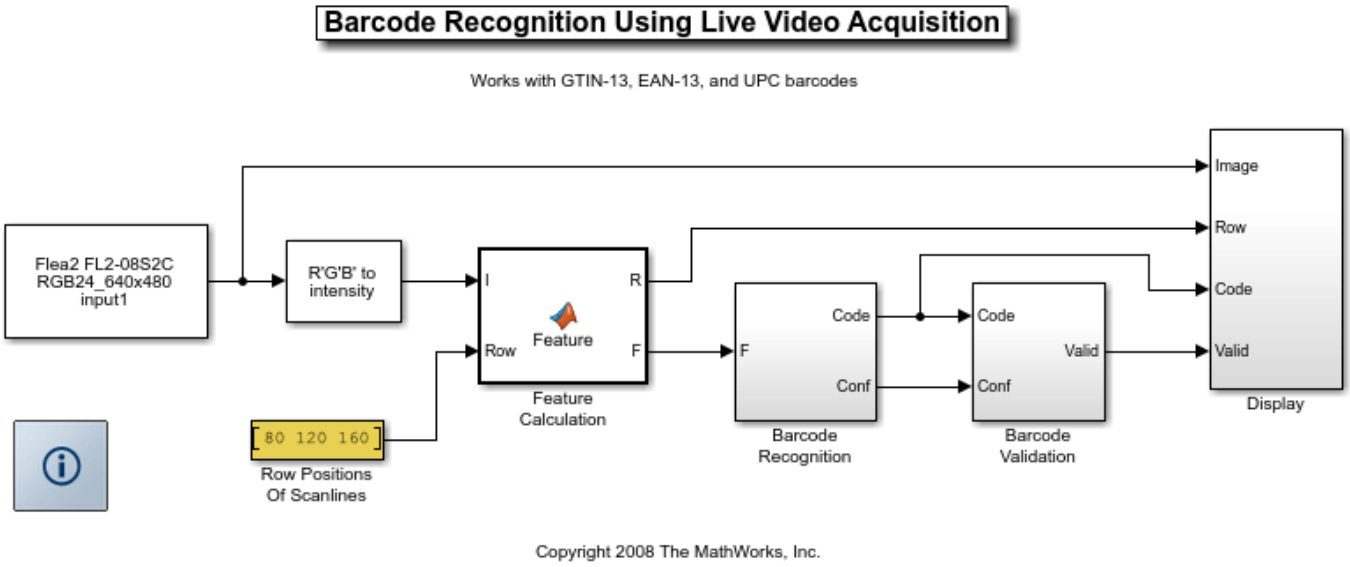
Image Acquisition Toolbox™ provides functions for acquiring images and video directly into MATLAB® and Simulink from PC-compatible imaging hardware. You can detect hardware automatically, configure hardware properties, preview an acquisition, and acquire images and video.

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)

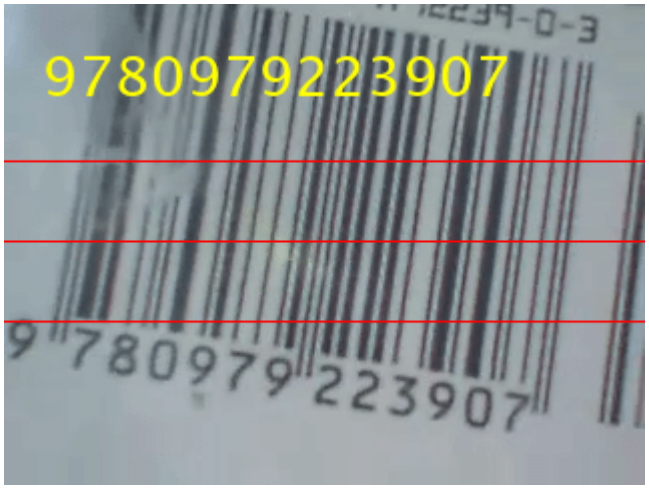
Example Model

The following figure shows the example model using the From Video Device block.



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.

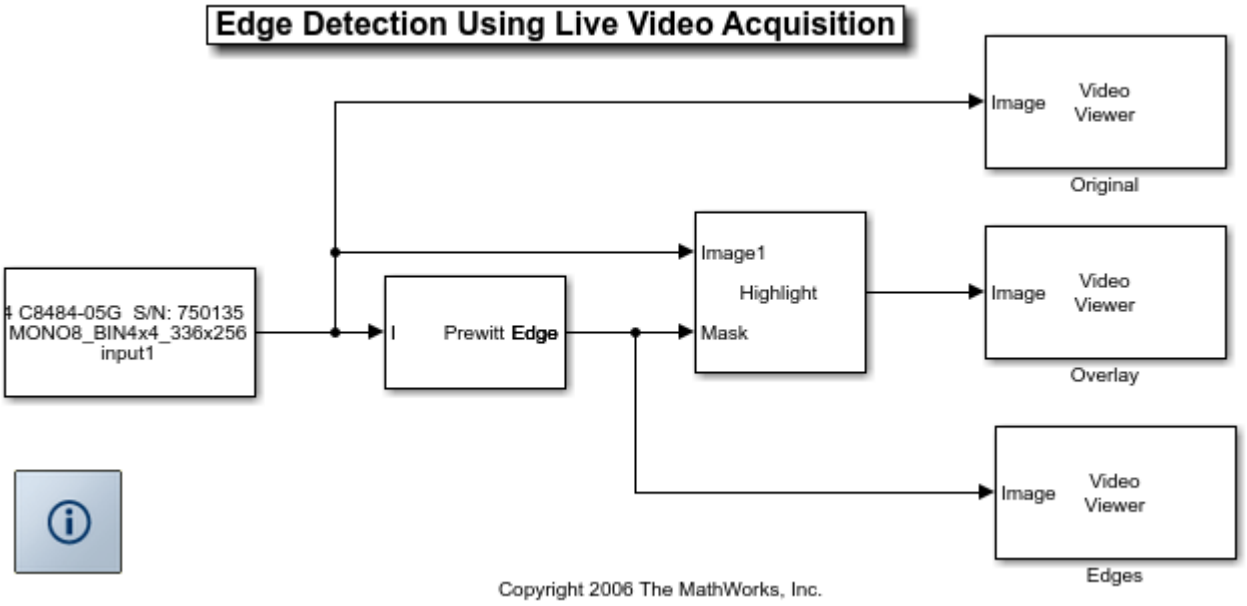
Image Acquisition Toolbox provides functions for acquiring images and video directly into MATLAB® and Simulink from PC-compatible imaging hardware. You can detect hardware automatically, configure hardware properties, preview an acquisition, and acquire images and video.

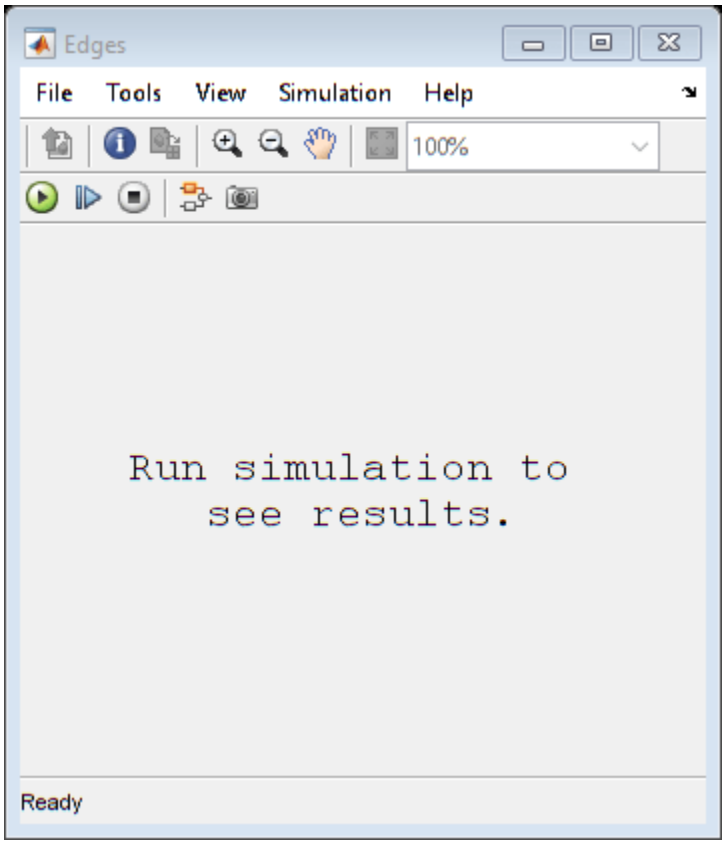
This example requires Image Acquisition Toolbox and Hamamatsu image acquisition device (C8484) to run the model.

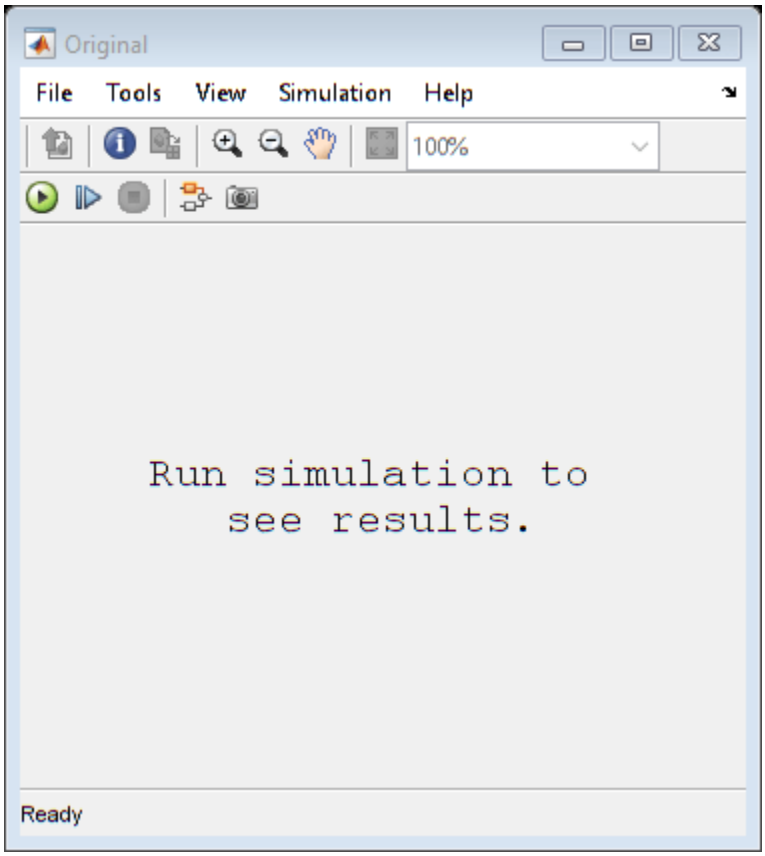
Watch edge detection using live video acquisition. (4 seconds)

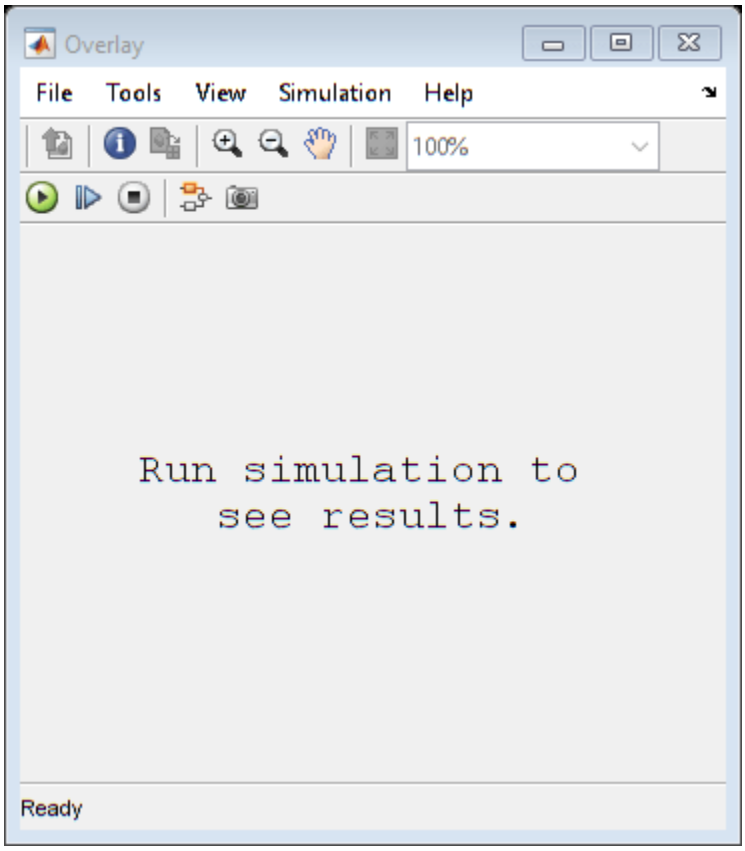
Example Model

The following figure shows the example model using the From Video Device block.




Edges
File
Tools
View
Simulation
Help
100%
Run simulation to
see results.
Ready


Original
File
Tools
View
Simulation
Help
100%
Run simulation to
see results.
Ready

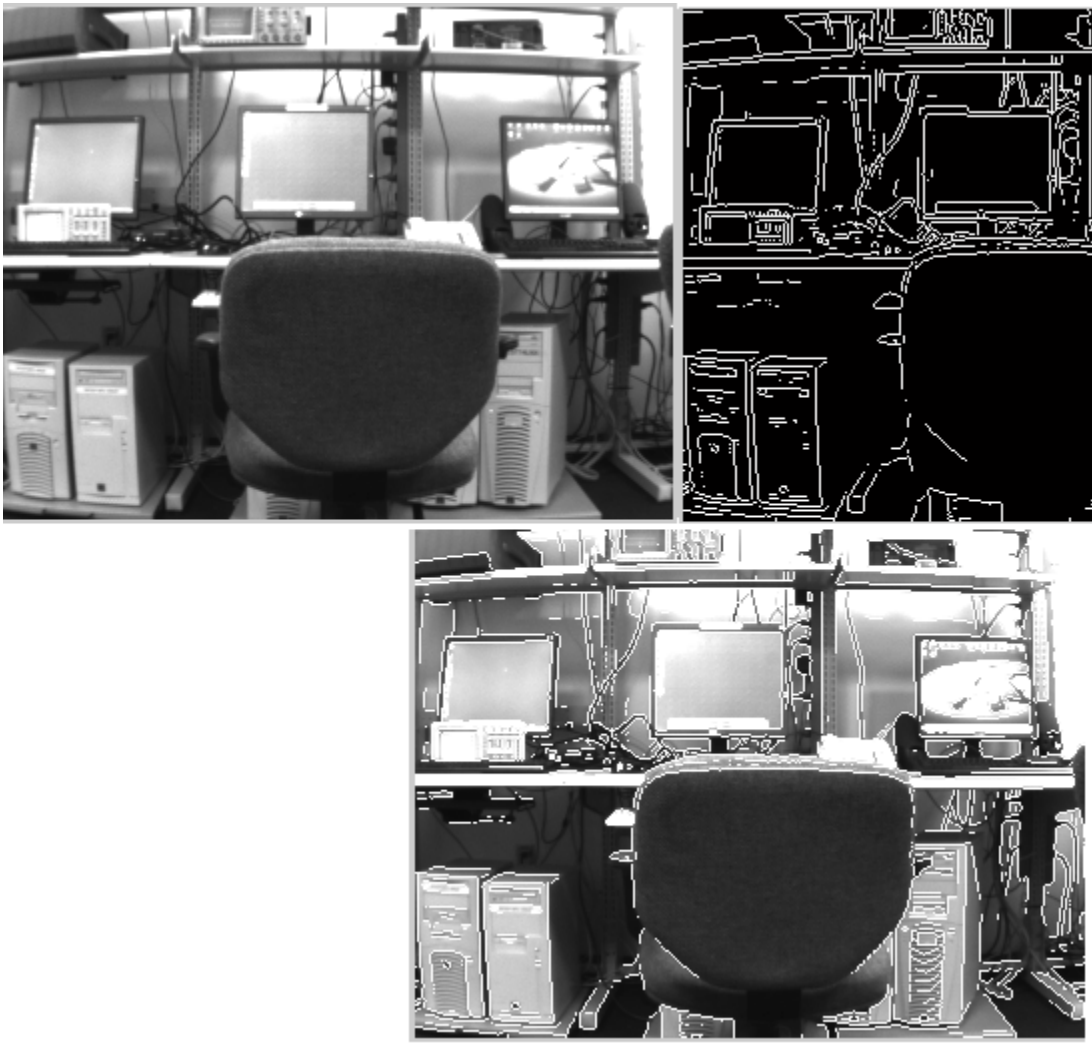


Live Video Input

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.

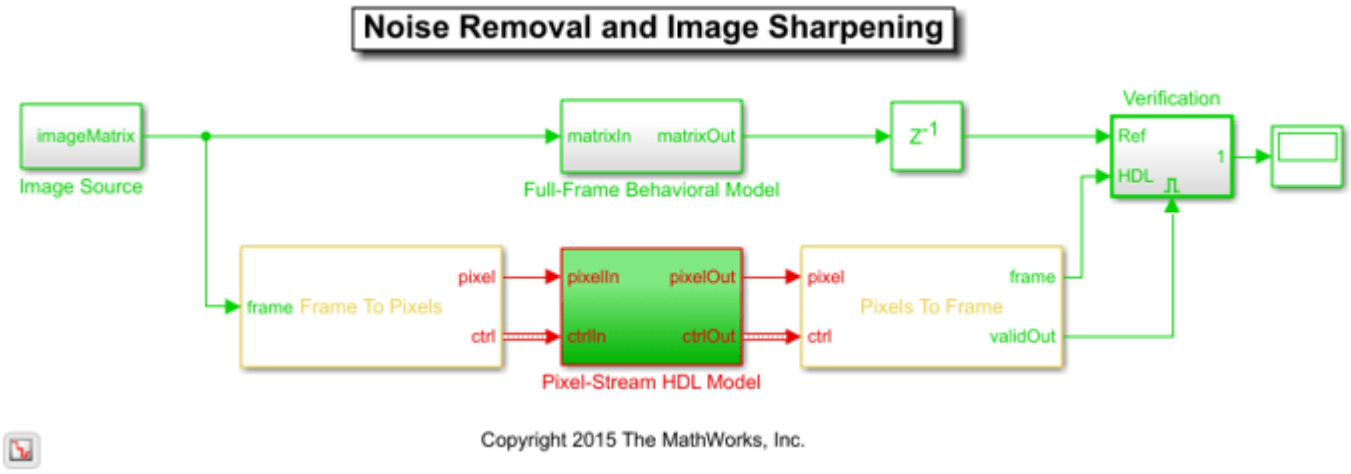
Vision HDL Toolbox provides video processing algorithms designed to generate readable, synthesizable code in VHDL and Verilog (with HDL Coder™). The generated HDL code can process 1080p video at a rate of 60 frames per second.

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

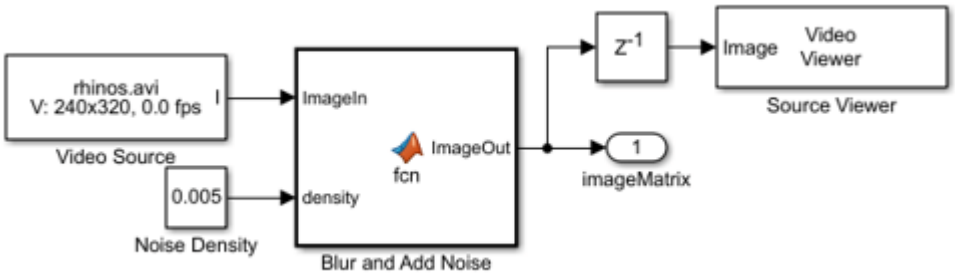
The `NoiseRemovalAndSharpeningHDLExample.slx` system is shown.



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.

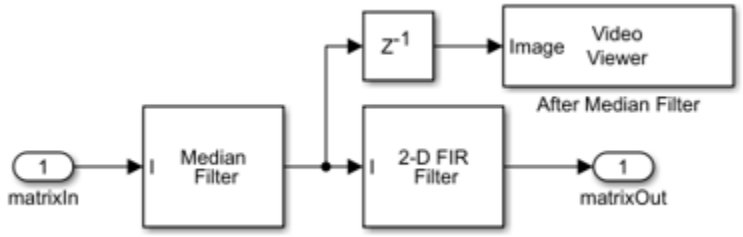
The difference in the color of the lines feeding the Full-Frame Behavioral Model and Pixel-Stream HDL Model subsystems indicates the change in the image rate on the streaming branch of the model. This rate transition occurs because the pixel stream is sent out in the same amount of time as the full video 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.

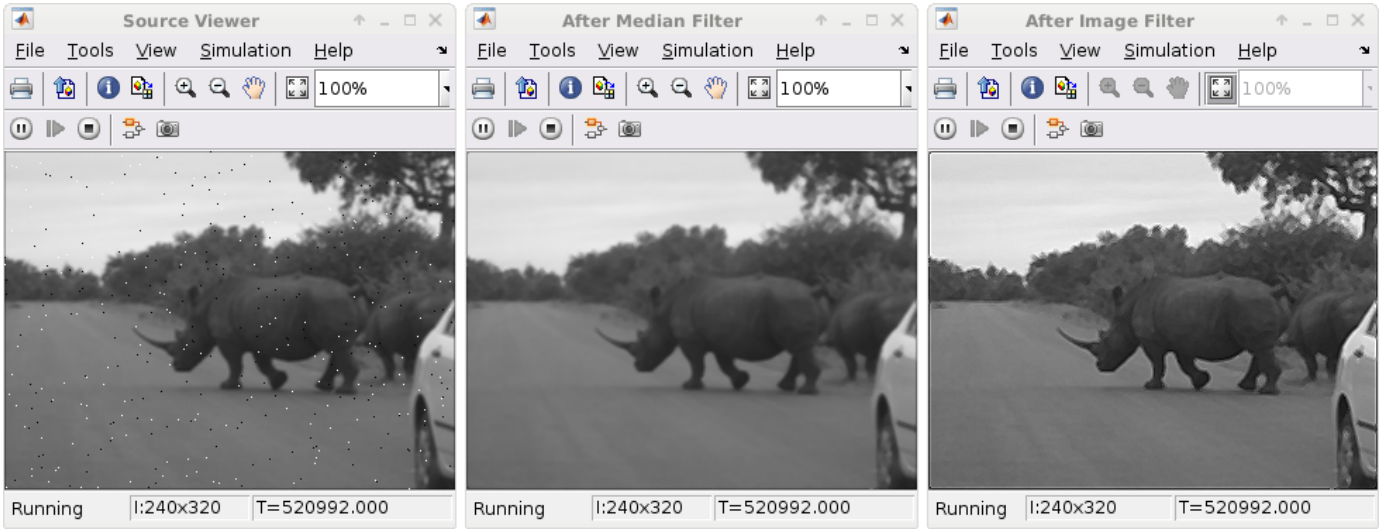


The Image Source block imports a greyscale image, then uses a MATLAB function block named Blur and Add Noise to blur the image and inject salt-and-pepper noise. The `imfilter` function uses a 3-by-3 averaging kernel to blur the image. The salt-and-pepper noise is injected by calling the `imnoise` command. 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.



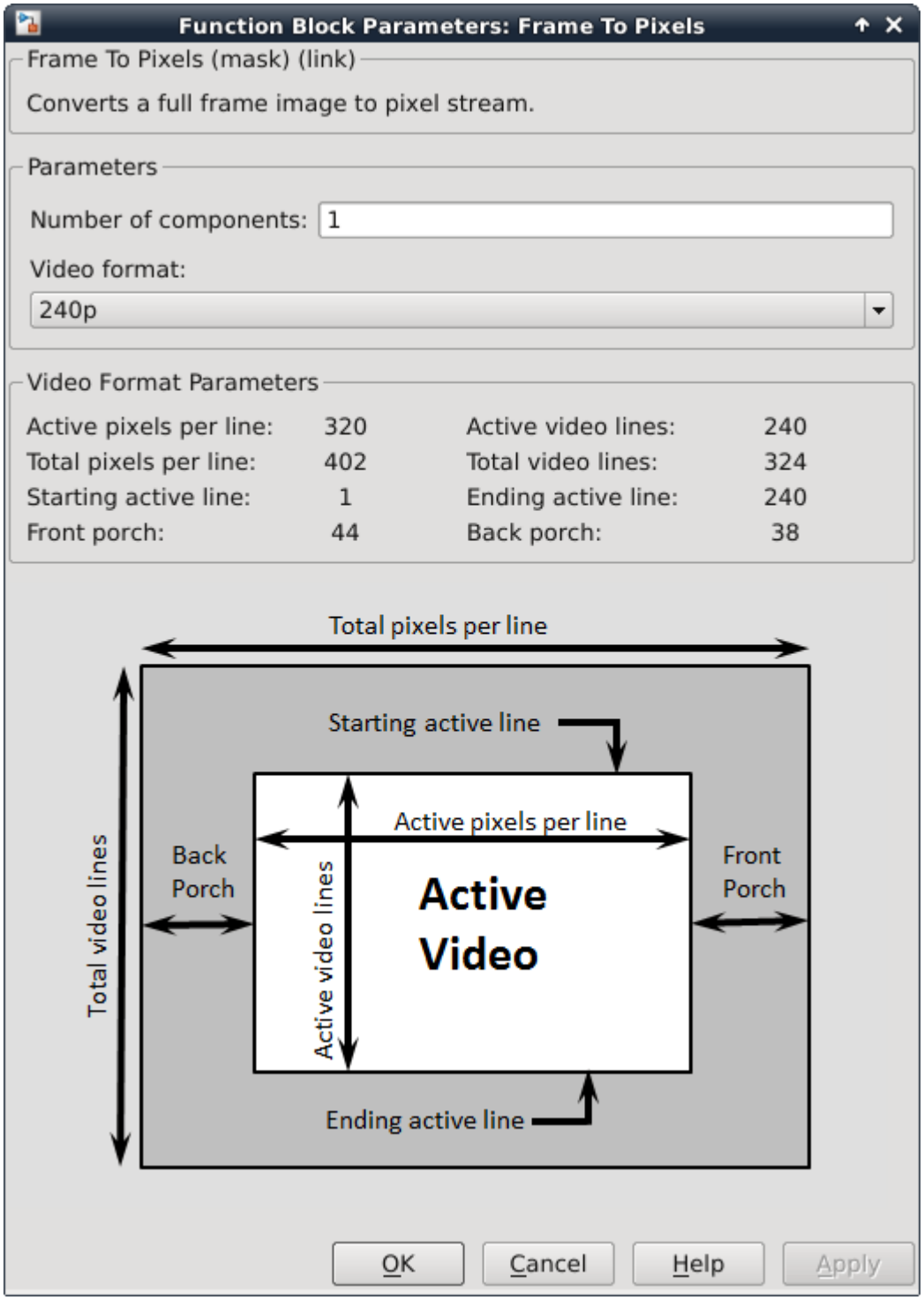
The displays below show one frame of the blurred and noisy source video, its de-noised version after median filtering, and the sharpened output after 2-D FIR filtering.



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:



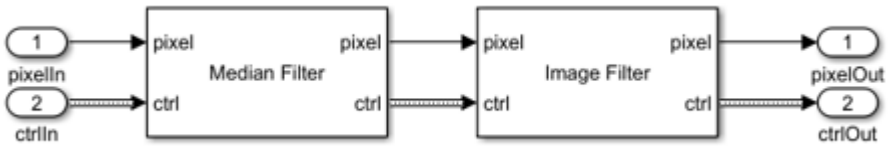
The Number of components field is set to 1 for grayscale image input, and the Video format field is 240p to match that of the video source.

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™.

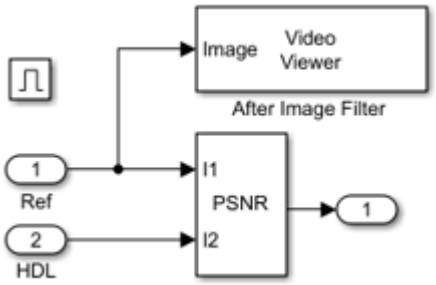


The Median Filter (Vision HDL Toolbox) block is used to remove the salt and pepper noise. Based on the filter coefficients, the Image Filter (Vision HDL Toolbox) block can be used to blur, sharpen, or detect the edges of the recovered image after median filtering. In this example, Image Filter is configured 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');
```

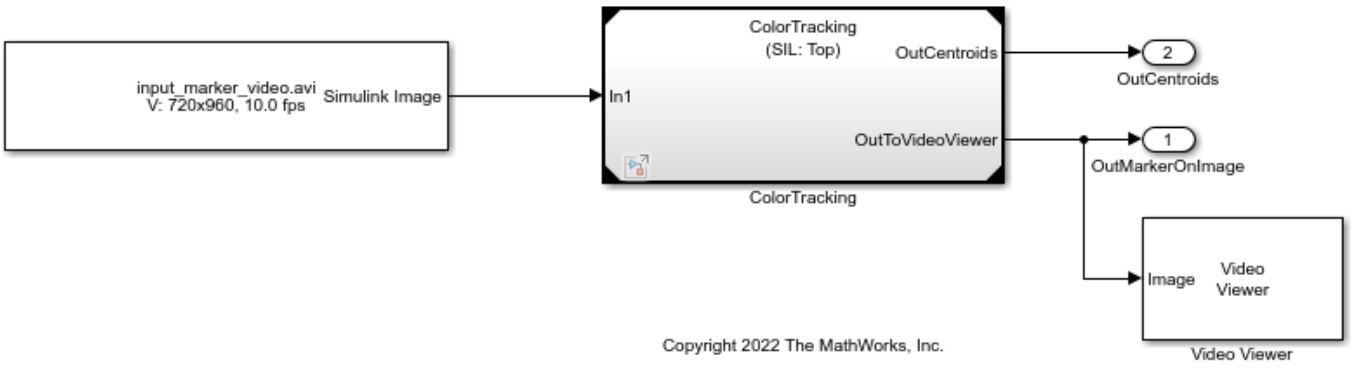
Track Marker Using Simulink Images

This example shows how to track a marker on a whiteboard by using images of the `Simulink.ImageType` data type.

Example Model

The `ex_tracking_marker` example model contains a model reference hierarcy. Each model contributes towards the image processing algorithm.

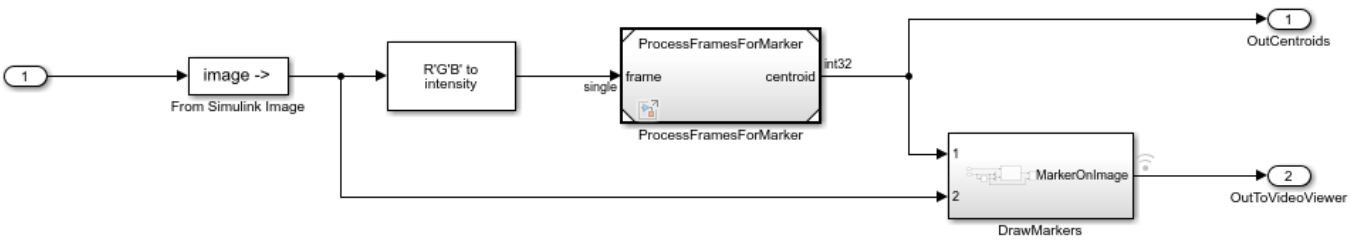
- `ex_tracking_marker` - Top model that tracks a marker in an input video. This model renders the output video by using the Video Viewer block and the logs the output frame in the `out.MarkerOnImage` workspace variable.
- `ColorTracking` - Referenced model that implements a basic image thresholding algorithm. Includes `ProcessFramesForMarker` processes the centroid of each frame. The `DrawMarkers` subsystem plots the line from the centroid of the previous frame to the current frame.
- `ProcessFramesForMarker` - Referenced model that looks for pixels in each frame and localizes the centroid in every frame. The MATLAB Function block `DetectMarker` uses the thresholding logic and the MATLAB Function block `FindContours` localizes the centroid of the detected pixels.



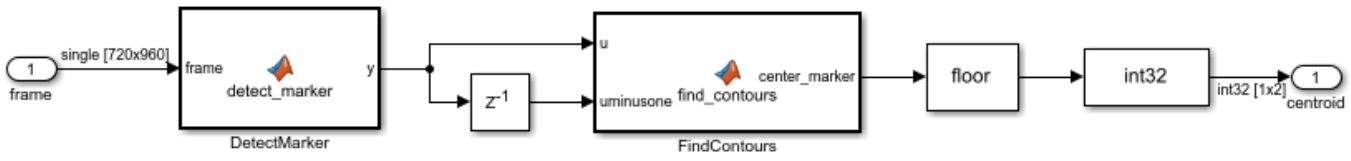
Working with Simulink Images

A `Simulink.ImageType` data type is an encapsulated object that defines an image with fixed meta-attributes specific to this data type. The `ex_tracking_marker` model takes the input video through From Multimedia File block as a Simulink image by setting the block parameter **Image signal** to `Simulink image signal`.

If your image processing algorithm includes blocks that currently don't support the `Simulink.ImageType` data type, use the From Simulink Image block to unpack data in from the Simulink image to a matrix. Specify the block parameter **Output dimensions** as `[720 960 3]` which corresponds to the rows, columns, and channels in the image. Because MATLAB Function blocks operate on matrix data only, the `ColorTracking` model uses a From Simulink Image block before passing the image data to the `ProcessFramesForMarker` model, which contains MATLAB Function blocks. To review these configurations, open the `ColorTracking` model.



The `DrawMarkers` subsystem draws the tracking line on the image, and converts the image from matrix format to a Simulink image by using the To Simulink Image block. This image is then fed to the Video Viewer block. To review these configurations, open the `ProcessFramesForMarker` model.



Simulate Model

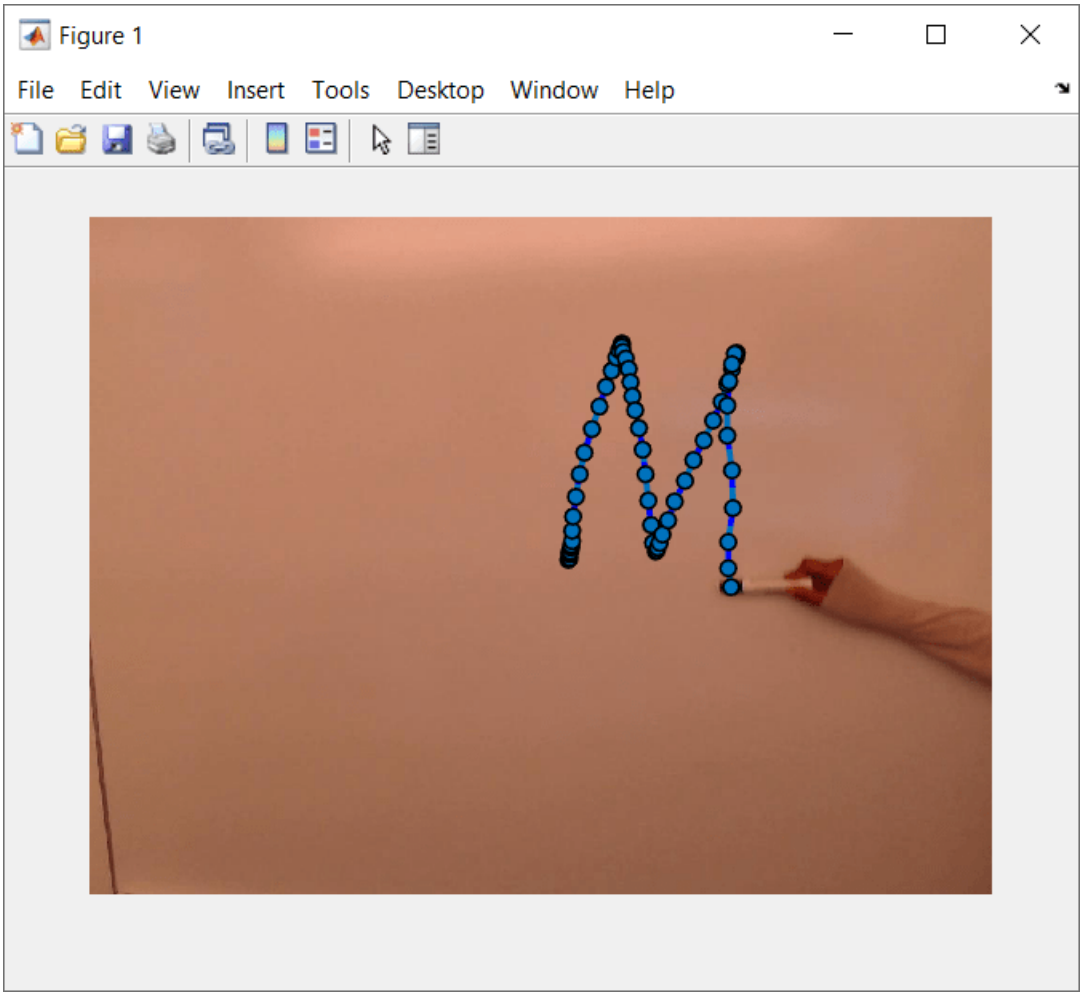
To simulate the model, on the Simulink toolstrip, in the **Simulation** tab, select **Run**. The Video Viewer block opens and displays the tracker tracing the marker. The simulation runs at a reduced pace so you can observe the real-time behaviour of the system. To change the pacing of the model, select **Run > Simulation Pacing**.

To generate a final image that draws over the logged outputs, in the MATLAB Command Window, run these commands.

```
vw = VideoWriter('WhatDidIDraw.mp4');
vw.open();
h=figure;
Ax = axes(h);
for i = 2:70
    imshow(out.yout{1}.Values.Data(:,:,:,i),'Parent',Ax);
    drawpolyline('Position',transpose(squeeze(out.yout{2}.Values.Data(:,:,1:i))),'StripeColor','
    vw.writeVideo(getframe(h));
end
vw.close();
```

Results

The script uses the logged output video to generate the path the marker followed and displays the result in the Video Viewer block.



See Also

`Simulink.ImageType` | To Simulink Image | From Simulink Image

Related Examples

- “Convert Between Simulink Image Type and Matrices” on page 13-69
- “Smile Detection by Using OpenCV Code in Simulink” on page 13-45

Video and Image Ground Truth Labeling

- “Export Ground Truth Object to Custom and COCO JSON Files” on page 7-2
- “Automate Ground Truth Labeling for Semantic Segmentation” on page 7-7

Export Ground Truth Object to Custom and COCO JSON Files

This example shows you how to export a ground truth object to a custom data format JavaScript Object Notation (JSON) file, and to a COCO [1] data format JSON file.

You can parse JSON files to use for training and inference in machine learning frameworks. The format of the custom data format JSON file is similar to the COCO data format. The difference is that the format for representing label annotations for the JSON file is modified to include various label types supported by the ground truth object.

This example uses a ground truth object that contains labels previously generated using the Video Labeler app.

To export a ground truth object from the Labeler app to the JSON file, follow these steps:

1. Use the Image Labeler or the Video Labeler App to label images or videos respectively.
2. Export the labels to file or the workspace to generate a ground truth object.
3. Follow the steps in this example.

Load Ground Truth Object

Load a ground truth object

```
load("VehicleImageSequenceGroundTruth.mat")
```

Display the label definitions

```
VehicleImageSequenceGroundTruth.LabelDefinitions
```

```
ans=3×6 table
      Name        Type              LabelColor                 Group       Description     Hiera
    _________    _______    ___________________________    ____________    ___________    _____

    {'Car'  }    Polygon     0.5862     0.8276     0.3103    {'Vehicle'}    {0×0 char}     {1×1 s
    {'Lane' }    Line       0.49412    0.18431    0.55686    {'Road'   }    {0×0 char}     {0×0 c
    {'Sunny'}    Scene            1    0.41176    0.16078    {'None'   }    {0×0 char}     {0×0 c
```

Add Info and License Information (Optional)

Add the metadata to the annotations by specifying the Info and the License Name-Value pairs

```
info = struct( ...
    'year', 2020, ...
    'version', '1', ...
    'description', 'Vehicles Dataset', ...
    'date_created', datetime ...
    );

licenses = struct( ...
    'url', {'www.mathworks.com', 'www.mathworks.com'}, ...
    'name', {'License 1', 'License 2'} ...
    );
```

Export the Ground Truth Object to a custom data format JSON file

Use `exportGroundTruthToJSON` function to export the ground truth object to a JSON file.

The `exportGroundTruthToJSON` function parses the data in the ground truth object, encodes it in a custom data format that is described below on page 7-0 , and then generates the JSON file.

```
annotationsFileName = 'vehicles_annotations.json';
exportGroundTruthToJSON(VehicleImageSequenceGroundTruth, annotationsFileName, 'Info', info, 'Lic
```

Export the Object Detection Ground Truth Object to a COCO data format JSON file

Use `exportGroundTruthToJSON` with the `'COCO'` Name-Value set to `true` to export object detection data that was labeled using polygons to the COCO object detection data format JSON file.

To create a ground truth object for object detection that can be exported to COCO data format JSON file, follow these steps:

1. Use the Polygon label type to label the object instances.
2. Use the Pixel label type to label the crowd regions of the object. Its label name should be as follows: polygonLabelName_crowd (Where polygonLabelName is the label name of the corresponding polygon label).

For example:

To annotate an image that contains both person instances and crowd regions,

1. Create a label of type polygon with name as `'person'` for annotating person instances.
2. Create a corresponding Pixel label with name `'person_crowd'` to annotate person-crowd regions.

The `'_crowd'` suffix is used to bind the two labels together. Therefore, ensure that for each crowd pixel label there is a corresponding polygon label.

Note: While exporting to COCO object detection data format, an expected warning is thrown specifying the label types that are being ignored. In this case, since the ground truth object has Line labels they will be ignored.

```
annotationsFileName = 'vehicles_annotations_coco.json';
exportGroundTruthToJSON(VehicleImageSequenceGroundTruth, annotationsFileName, ...
    'Info', info, 'License', licenses, 'COCO', true)
```

```
Warning: Label types, Line, Scene, are not supported and are ignored during export to a COCO data
```

Custom JSON Data Format

The exported JSON file contains these five fields:

1. info
2. licenses
3. categories
4. images/ video
5. annotations

Info

The Info field specifies the dataset and annotation information by using the 'Info' name-value pair argument. The value must be a scalar structure.

Default Value:

```
{
  "year": Current year,
  "version": "1",
  "description": "Created using MATLAB",
  "date_created": Current datetime
}
```

Licenses

The Licenses field specifies information about the licenses by using the License name-value pair argument. The value must be a structure array.

Default Value: `[]`

Categories

The Categories field contains the contents of the ground truth label definitions table. The variable names in the table are used as fields and are modified to be similar to the COCO data format. An additional field Id identifies each label definition.

Each category object contains the following two additional fields which replace the 'Hierarchy' variable if it exists:

1. Attributes: Contains attributes definitions of the label.
2. Sublabels: Conatins sublabels definitions of the label.

Images/ Video

The Images/ Video field contains a list of the images (or video) objects and related information.

Data Format:

```
{
  "id": Unique Image ID,
  "time_stamp": Time stamp of the image frame (Only applicable to data sources with timestamps),
  "width": Width of the image/video,
  "height": Height of the image/video,
  "file_name": Name of the file,
  "file_path": Absolute file path,
  "date_captured": Last modified date of the image/ Current datetime
}
```

Annotations

The Annotations field contains a list of annotation objects. The data format of the annotation objects depends on the label type and is as follows:

Rectangle ROI

```
{
  "id": Unique Annotation ID,
  "image_id": Image ID ,
  "category_id": Category ID,
  "position":  [x,y,width,height] bounding box location,
  "attributes": Contains attributes data,
```

```
  "sublabels": Contains sublabels data
}
```

- `x` and `y` specify the upper-left corner of the rectangle and are 0-indexed.
- `w` specifies the width of the rectangle, which is its length along the *x*-axis.
- `h` specifies the height of the rectangle, which is its length along the *y*-axis.

Polygon ROI

```
{
  "id": Unique Annotation ID,
  "image_id": Image ID ,
  "category_id": Category ID,
  "position":  [[x1 y1 x2 y2 ...  xN yN]] for N points in the polygon,
  "attributes": Contains attributes data,
  "sublabels": Contains sublabels data
}
```

PolyLine ROI

```
{
  "id": Unique Annotation ID,
  "image_id": Image ID,
  "category_id": Category ID,
  "position": [x1 y1 x2 y2 ...  xN yN] for N points in the polyline.,
  "attributes": Contains attributes data,
  "sublabels": Contains sublabels data
}
```

- x1, y1,... specify the point location and are 0-indexed.

ProjectedCuboid ROI

```
{
"id": Unique Annotation ID,
"image_id": Image ID,
"category_id": Category ID,
"position": List of the form [xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot],
  "attributes": Contains attributes data,
  "sublabels": Contains sublabels data
}
```

- `xctr`, `yctr`, and `zctr` specify the center of the projected cuboid and are 0-indexed.
- `xlen`, `ylen`, and `zlen` specify the length of the projected cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively, before rotation has been applied.
- `xrot`, `yrot`, and `zrot` specify the rotation angles for the projected cuboid along the *x*-axis, *y*-axis, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes.

PixelLabel ROI

```
{
  "image_id": Image ID,
  "file_name": Name of the pixel label image file,
  "file_path": Path to the pixel label image file,
  "segments_info": List of segment objects
}
```

Segment Objects format: Each segment in the pixel label image is representated as a segment object.

```
{
"id": PixelLabelID of the label or R+G*256+B*256^2 in case if PixeLabelID is an array,
"category_id": Category ID,
"area": Area covered by the pixels
}
```

Scene

```
{
  "id": Unique Annotation ID,
  "image_id": Image ID,
  "category_id": Category ID of the scene label applicable to the image
}
```

Limitations

1. Custom label types cannot be exported to the JSON file.
2. You cannot export data from a custom reader ground truth datasource to a JSON file format.
3. Only Polygon and crowd Pixel Label ROIs can be exported to the COCO object detection data format.
4. The JSON file generated by the "exportGroundTruthToJSON" helper function cannot be imported back into MATLAB as a ground truth object.

References

[1] Lin TY. et al. (2014) Microsoft COCO: Common Objects in Context. In: Fleet D., Pajdla T., Schiele B., Tuytelaars T. (eds) Computer Vision – ECCV 2014. ECCV 2014. Lecture Notes in Computer Science, vol 8693. Springer, Cham. https://doi.org/10.1007/978-3-319-10602-1_48

Automate Ground Truth Labeling for Semantic Segmentation

This example shows how to use a pretrained semantic segmentation algorithm to segment the sky and road in an image, and use this algorithm to automate ground truth labeling in the Ground Truth Labeler (Automated Driving Toolbox) app.

The Ground Truth Labeler App

Good ground truth data is crucial for developing automated driving algorithms and evaluating their performance. However, creating and maintaining a diverse and high-quality set of annotated driving data requires significant effort. The Ground Truth Labeler (Automated Driving Toolbox) app makes this process easy and efficient. This app includes features to annotate objects as rectangles, lines, or pixel labels. Pixel labeling is a process in which each pixel in an image is assigned a class or category, which can then be used to train a pixel-level segmentation algorithm. Although you can use the app to manually label all your data, this process requires a significant amount of time and resources, especially for pixel labeling. As an alternative, the app also provides a framework to incorporate algorithms to extend and automate the labeling process. You can use the algorithms you create to automatically label entire data sets, and then end with a more efficient, shorter manual verification step. You can also edit the results of the automation step to account for challenging scenarios that the algorithm might have missed.

In this example, you will:

- Use a pretrained segmentation algorithm to segment pixels that belong to the categories 'Road' and 'Sky'.
- Create an automation algorithm that can be used in the Ground Truth Labeler app to automatically label road and sky pixels.

This ground truth data can then be used to train a new semantic segmentation network, or retrain an existing one.

Create a Road and Sky Detection Algorithm

First, create a semantic segmentation algorithm that segments road and sky pixels in an image. The “Semantic Segmentation Using Deep Learning” on page 3-222 example describes how to train a deep learning network for semantic segmentation. This network has been trained to predict 11 classes of semantic labels including 'Road' and 'Sky'. The performance of these networks depends on how generalizable they are. Applying the networks to situations they did not encounter during training can lead to subpar results. Iteratively introducing custom training data to the learning process can make the network perform better on similar data sets.

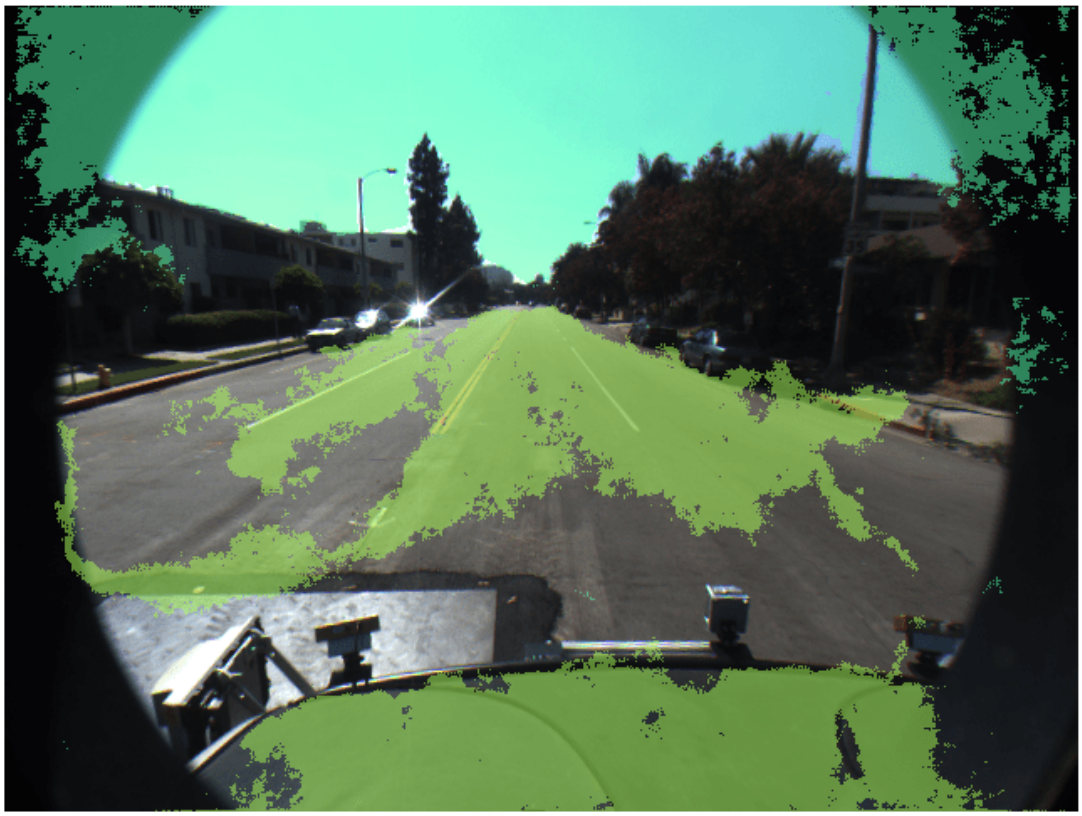
Download a network, which was pretrained on the CamVid dataset [1][2] from the University of Cambridge.

```
pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/segnetVGG16CamVid.mat';
pretrainedFolder = fullfile(tempdir,'pretrainedSegNet');
pretrainedSegNet = fullfile(pretrainedFolder,'segnetVGG16CamVid.mat');
if ~exist(pretrainedSegNet,'file')
    if ~exist(pretrainedFolder,'dir')
        mkdir(pretrainedFolder);
    end
    disp('Downloading pretrained SegNet (107 MB)...');
    websave(pretrainedSegNet,pretrainedURL);
end
```

```
Downloading pretrained SegNet (107 MB)...
```

Segment an image and display it.

```
% Load the semantic segmentation network
data = load(pretrainedSegNet);

% Load a test image from drivingdata
roadSequenceData = fullfile(toolboxdir('driving'), 'drivingdata', 'roadSequence');
I = imread(fullfile(roadSequenceData, 'f00000.png'));

% Run the network on the image
automatedLabels = semanticseg(I, data.net);

% Display the labels overlaid on the image, choosing relevant categories
figure, imshow(labeloverlay(I, automatedLabels, 'IncludedLabels', ["Sky", "Road"]));
```



The output of the network is represented in MATLAB® as a categorical matrix. The categories listed include all those that the semantic segmentation network has been trained on, not just the categories present in the output. This information is also available from the network object itself.

```
data.net.Layers(end).ClassNames
```

```
% List categories of pixels labeled
categories(automatedLabels)
```

```
ans = 11×1 cell
    {'Sky'       }
    {'Building'  }
    {'Pole'      }
    {'Road'      }
    {'Pavement'  }
    {'Tree'      }
    {'SignSymbol'}
    {'Fence'     }
    {'Car'       }
    {'Pedestrian'}
    {'Bicyclist' }
```

```
% The blue overlay indicates the 'Sky' category, and the green overlay
% indicates 'Road'.
```

Review the Pixel Segmentation Automation Class

Incorporate this semantic segmentation algorithm into the automation workflow of the app by creating a class that inherits from the abstract base class `vision.labeler.AutomationAlgorithm`. This base class defines the API that the app uses to configure and run the algorithm. The Ground Truth Labeler app provides a convenient way to obtain an initial automation class template. For details, see “Create Automation Algorithm for Labeling” on page 9-2. The `RoadAndSkySegmentation` class is based on this template and provides a ready-to-use automation class for pixel label segmentation.

The first set of properties in the `RoadAndSkySegmentation` class specify the name of the algorithm, provide a brief description of it, and give directions for using it.

```
   properties(Constant)

       %Name
       %   Character vector specifying name of algorithm.
       Name = 'RoadAndSkySegmentation'

       %Description
       %   Character vector specifying short description of algorithm.
       Description = 'This algorithm uses semanticseg with a pretrained network to annotate roads

       %UserDirections
       %   Cell array of character vectors specifying directions for
       %   algorithm users to follow in order to use algorithm.
       UserDirections = {...
           ['Automation algorithms are a way to automate manual labeling ' ...
           'tasks. This AutomationAlgorithm automatically creates pixel ', ...
           'labels for road and sky.'], ...
           ['Review and Modify: Review automated labels over the interval ', ...
           'using playback controls. Modify/delete/add ROIs that were not ' ...
           'satisfactorily automated at this stage. If the results are ' ...
           'satisfactory, click Accept to accept the automated labels.'], ...
           ['Accept/Cancel: If results of automation are satisfactory, ' ...
           'click Accept to accept all automated labels and return to ' ...
           'manual labeling. If results of automation are not ' ...
```

```
            'satisfactory, click Cancel to return to manual labeling ' ...
            'without saving automated labels.']};
    end
```

The next section of the `RoadAndSkySegmentation` class specifies the custom properties needed by the core algorithm. The `PretrainedNetwork` property holds the pretrained network. The `AllCategories` property holds the names of all the categories.

```
    properties
        % PretrainedNetwork saves the SeriesNetwork object that does the semantic
        % segmentation.
        PretrainedNetwork

        % Categories holds the default 'background', 'road', and 'sky'
        % categorical types.
        AllCategories = {'background'};

        % Store names for 'road' and 'sky'.
        RoadName
        SkyName
    end
```

`checkLabelDefinition`, the first method defined in `RoadAndSkySegmentation`, checks that only labels of type `PixelLabel` are enabled for automation. `PixelLabel` is the only type needed for semantic segmentation.

```
        function TF = checkLabelDefinition(~, labelDef)
            isValid = false;

            if (strcmpi(labelDef.Name, 'road') && labelDef.Type == labelType.PixelLabel)
                isValid = true;
                algObj.RoadName = labelDef.Name;
                algObj.AllCategories{end+1} = labelDef.Name;
            elseif (strcmpi(labelDef.Name, 'sky') && labelDef.Type == labelType.PixelLabel)
                isValid = true;
                algObj.SkyName = labelDef.Name;
                algObj.AllCategories{end+1} = labelDef.Name;
            elseif(labelDef.Type == labelType.PixelLabel)
                isValid = true;
            end
        end
```

The next set of functions control the execution of the algorithm. The `vision.labeler.AutomationAlgorithm` class includes an interface that contains methods like `'initialize'`, `'run'`, and `'terminate'` for setting up and running the automation with ease. The `initialize` function populates the initial algorithm state based on the existing labels in the app. In the `RoadAndSkySegmentation` class, the `initialize` function has been customized to load the pretrained semantic segmentation network from `tempdir` and save it to the `PretrainedNetwork` property.

```
        function initialize(algObj, ~, ~)

            % Point to tempdir where pretrainedSegNet was downloaded.
            pretrainedFolder = fullfile(tempdir,'pretrainedSegNet');
            pretrainedSegNet = fullfile(pretrainedFolder,'segnetVGG16CamVid.mat');
            data = load(pretrainedSegNet);
            % Store the network in the 'PretrainedNetwork' property of this object.
```

```
        algObj.PretrainedNetwork = data.net;
    end
```

Next, the `run` function defines the core semantic segmentation algorithm of this automation class. `run` is called for each video frame, and expects the automation class to return a set of labels. The `run` function in `RoadAndSkySegmentation` contains the logic introduced previously for creating a categorical matrix of pixel labels corresponding to "Road" and "Sky". This can be extended to any categories the network is trained on, and is restricted to these two for illustration only.

```
    function autoLabels = run(algObj, I)
        % Setup categorical matrix with categories including road and
        % sky
        autoLabels = categorical(zeros(size(I,1), size(I,2)),0:2,algObj.AllCategories,'Ordina

        pixelCat = semanticseg(I, algObj.PretrainedNetwork);
        if ~isempty(pixelCat)
            % Add the selected label at the bounding box position(s)
            autoLabels(pixelCat == "Road") = algObj.RoadName;
            autoLabels(pixelCat == "Sky") = algObj.SkyName;
        end
    end
```

This algorithm does not require any cleanup, so the `terminate` function is empty.

Use the Pixel Segmentation Automation Class in the App

The properties and methods described in the previous section have been implemented in the `RoadAndSkySegmentation` automation algorithm class file. To use this class in the app:

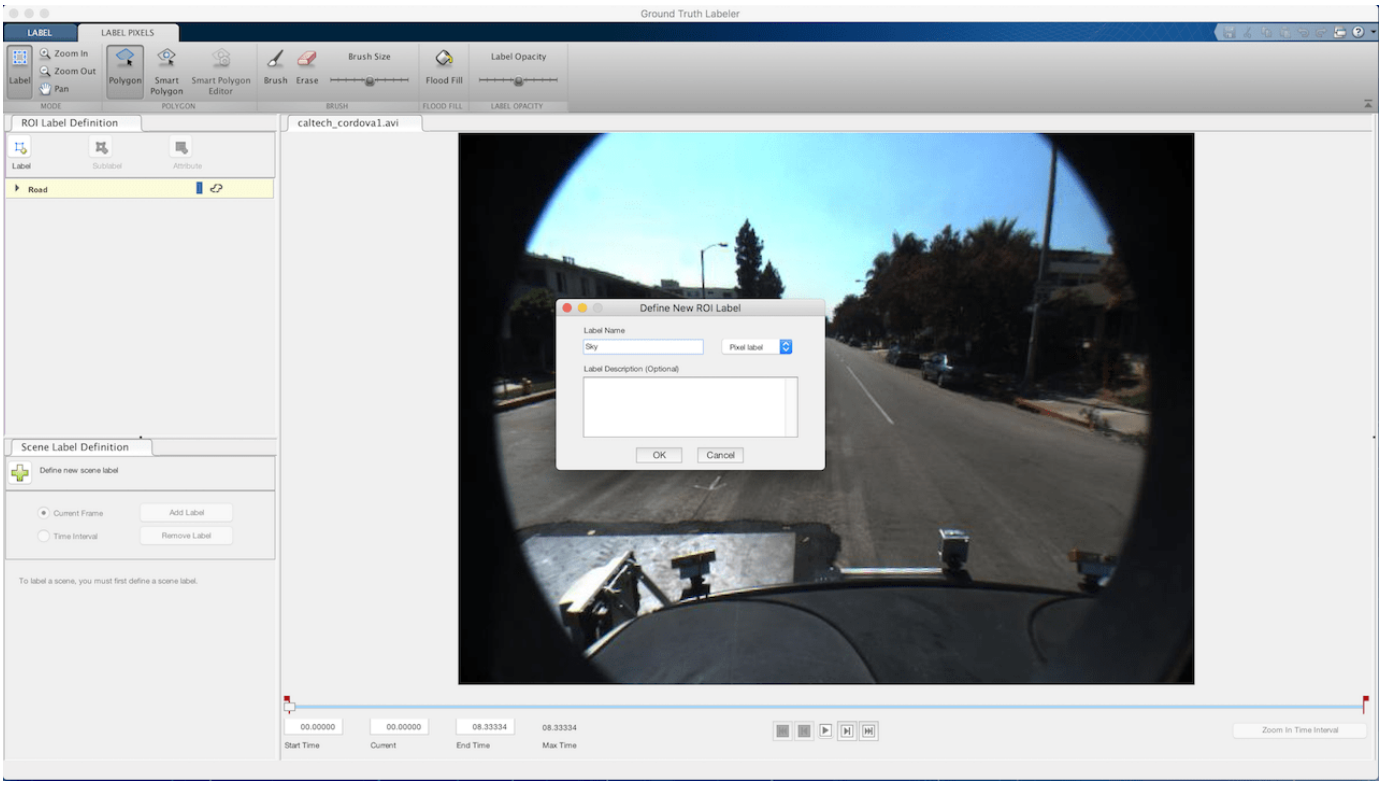
- Create the folder structure `+vision/+labeler` required under the current folder, and copy the automation class into it.

  ```
  mkdir('+vision/+labeler');
  copyfile('RoadAndSkySegmentation.m','+vision/+labeler');
  ```

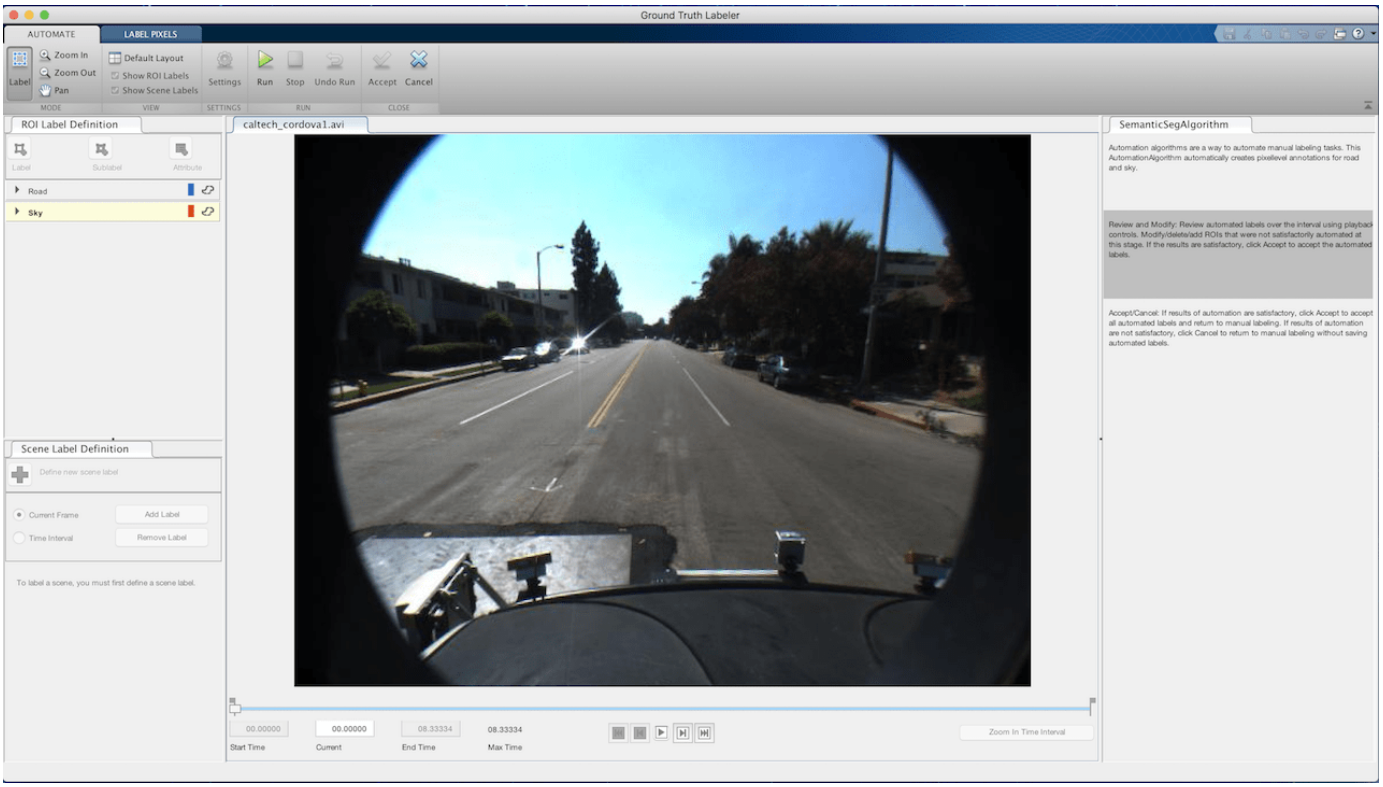
- Open the `groundTruthLabeler` app with custom data to label. For illustration purposes, open the `caltech_cordova1.avi` video.

  ```
  groundTruthLabeler caltech_cordova1.avi
  ```

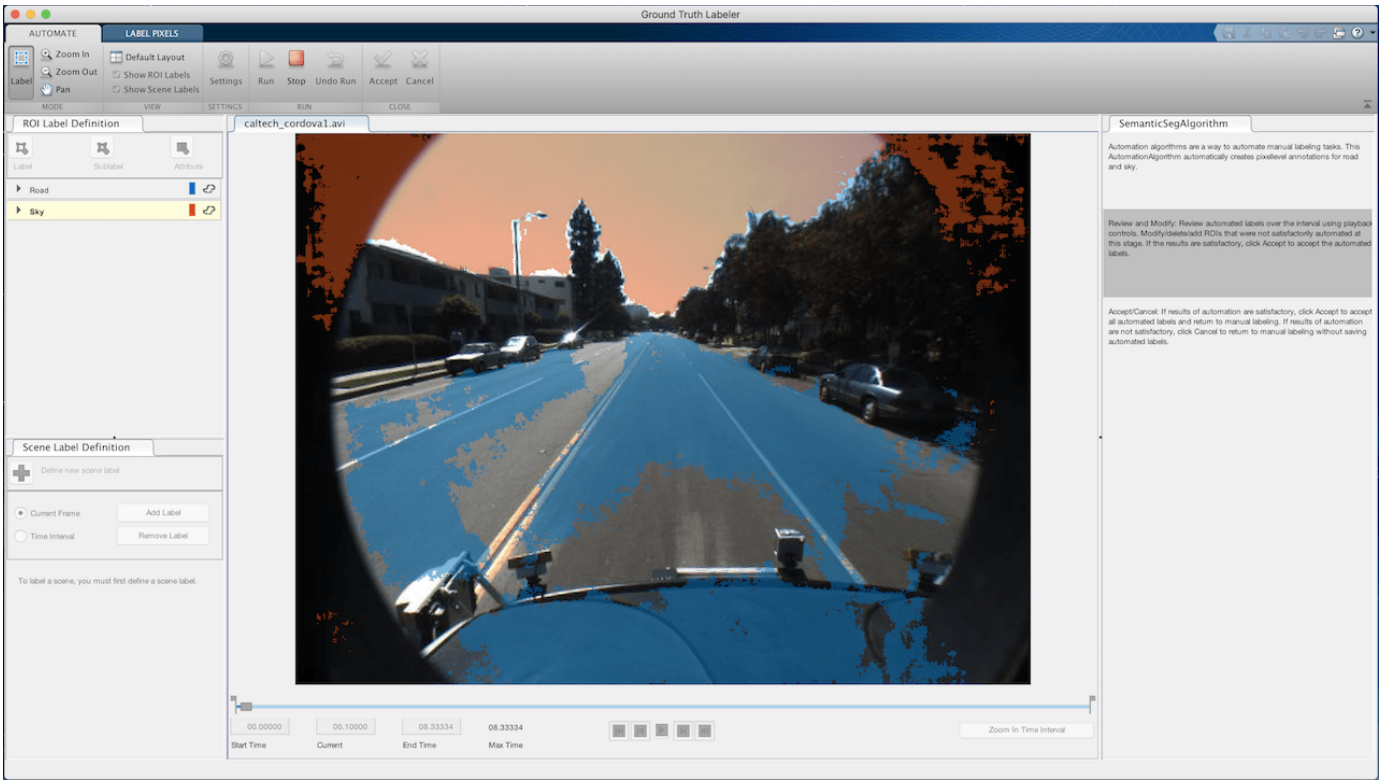
- On the left pane, click the **Define new ROI label** button and define two ROI labels with names `Road` and `Sky`, of type `Pixel label` as shown.



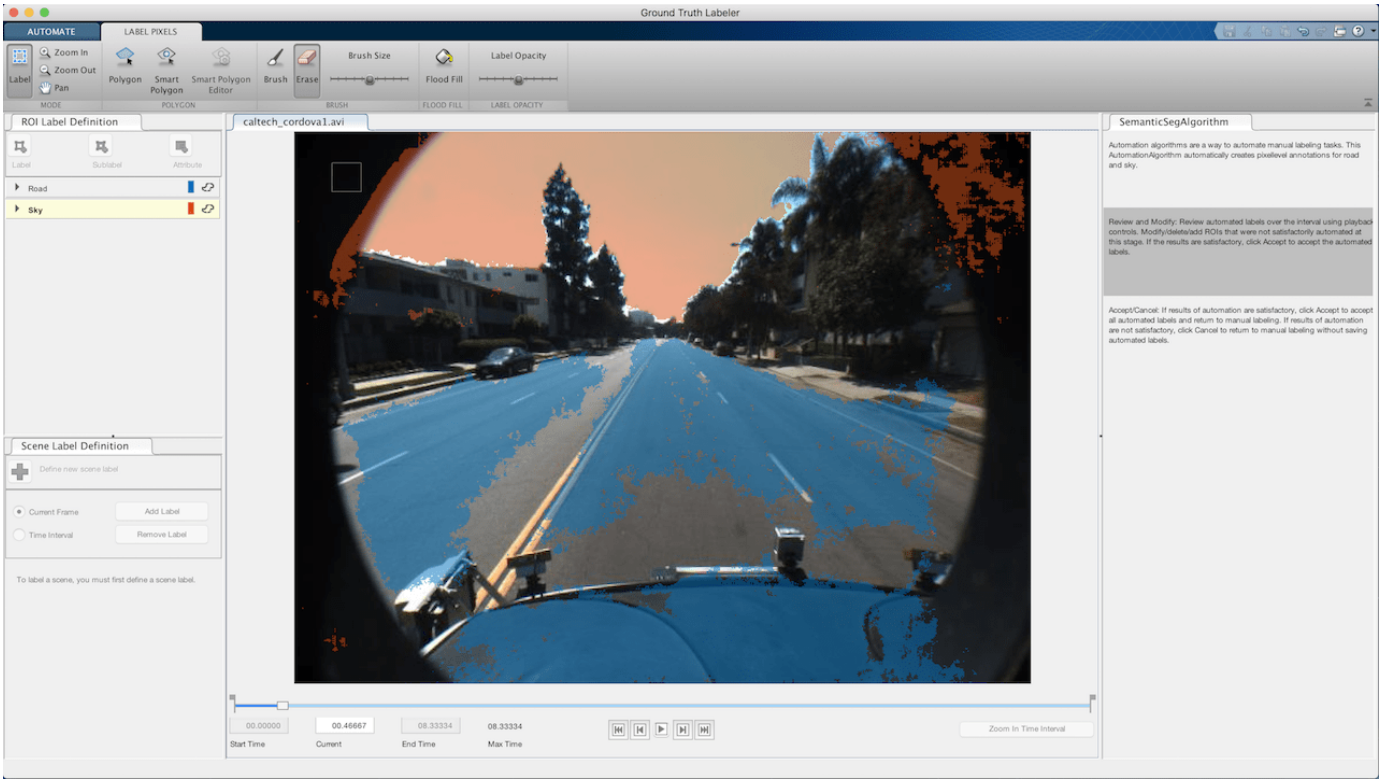
- Click **Algorithm > Select Algorithm > Refresh list**.
- Click **Algorithm > RoadAndSkySegmentation**. If you do not see this option, ensure that the current working folder has a folder called `+vision/+labeler`, with a file named `RoadAndSkySegmentation.m` in it.
- Click **Automate**. A new panel opens, displaying directions for using the algorithm.



- Click **Run**. The created algorithm executes on each frame of the video, segmenting "Road" and "Sky" categories. After the run is completed, use the slider or arrow keys to scroll through the video and verify the result of the automation algorithm.



- It is evident that regions outside the camera field of view are incorrectly labeled as "Sky", and parts of the ego vehicle itself are marked as "Road". These results indicate that the network has not been previously trained on such data. This workflow allows for making manual corrections to these results, so that an iterative process of training and labeling (sometimes called *active learning* or *human in the loop*) can be used to further refine the accuracy of the network on custom data sets. You can manually tweak the results by using the brush tool in the **Label Pixels** tab and adding or removing pixel annotations. Other tools like flood fill and smart polygons are also available in the **Label Pixels** tab and can be used when appropriate.



- Once you are satisfied with the pixel label categories for the entire video, click **Accept**.

Automation for pixel labeling for the video is complete. You can now proceed with labeling other objects of interest, save the session, or export the results of this labeling run.

Conclusion

This example showed how to use a pretrained semantic segmentation network to accelerate labeling of road and sky pixels in the Ground Truth Labeler app using the `AutomationAlgorithm` interface.

References

1 Brostow, Gabriel J., Jamie Shotton, Julien Fauqueur, and Roberto Cipolla. "Segmentation and Recognition Using Structure from Motion Point Clouds." *ECCV*. 2008.

2 Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A High-Definition Ground Truth Database." *Pattern Recognition Letters*. 2008.

Tracking and Motion Estimation Examples

- “Video Stabilization” on page 8-2
- “Video Stabilization Using Point Feature Matching” on page 8-5
- “Face Detection and Tracking Using CAMShift” on page 8-15
- “Face Detection and Tracking Using the KLT Algorithm” on page 8-20
- “Face Detection and Tracking Using Live Video Acquisition” on page 8-26
- “Motion-Based Multiple Object Tracking” on page 8-31
- “Tracking Pedestrians from a Moving Car” on page 8-40
- “Use Kalman Filter for Object Tracking” on page 8-50
- “Detect Cars Using Gaussian Mixture Models” on page 8-61

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];
```

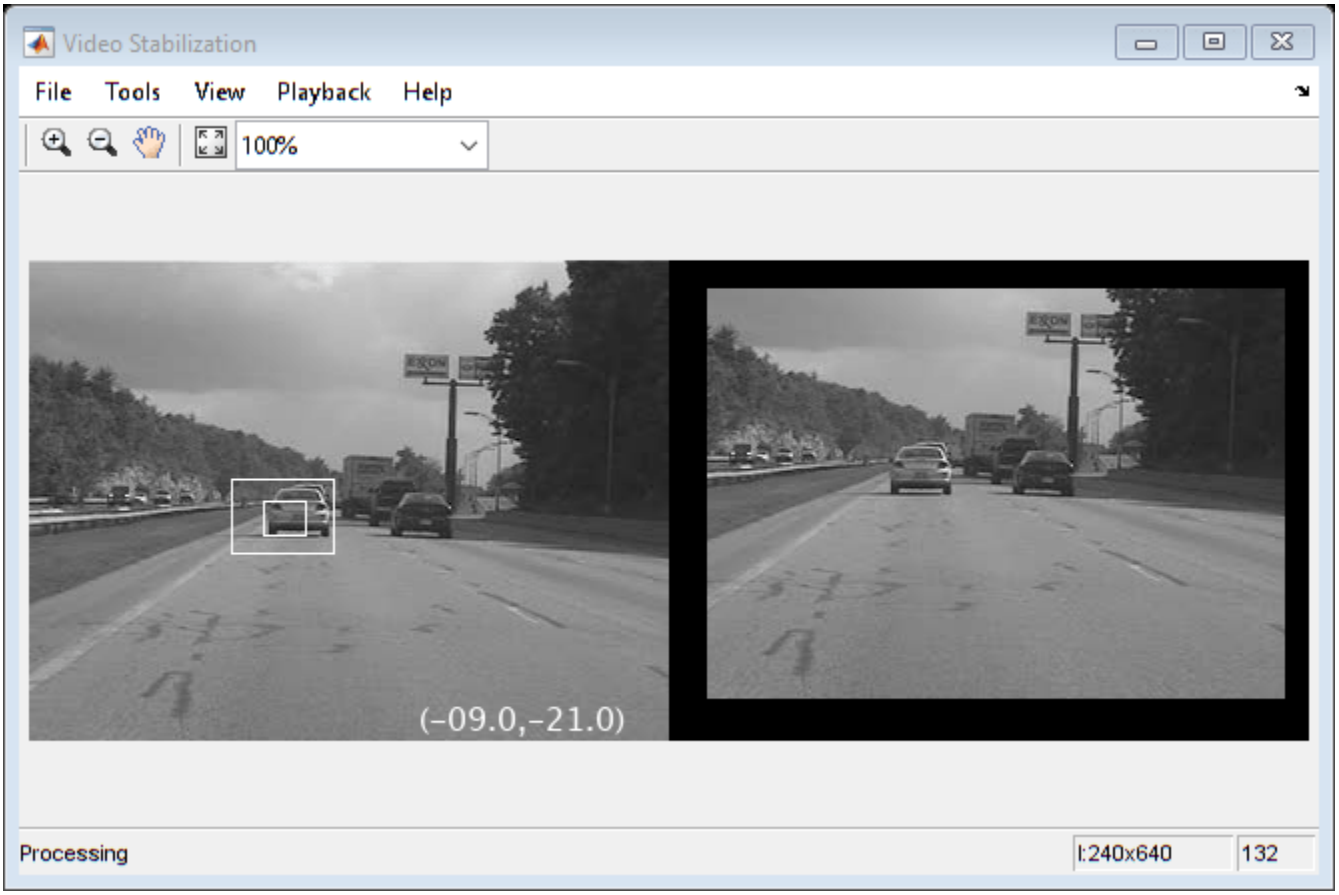
Here we initialize some variables used in the processing loop.

```
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.

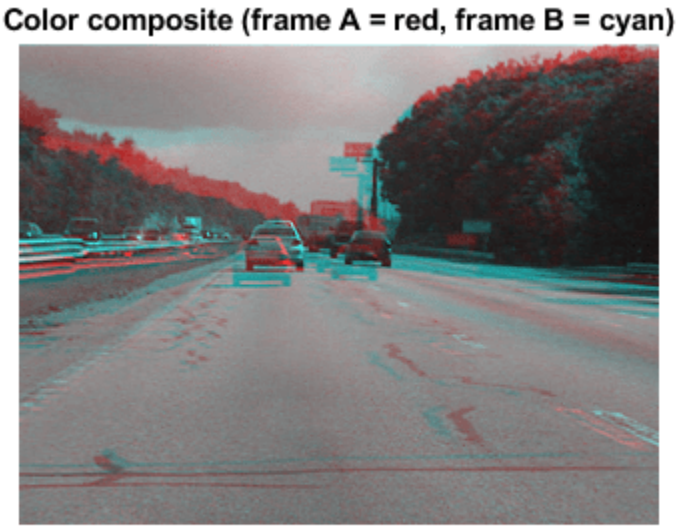
This stabilization algorithm involves two steps. First, we determine the affine image transformations between all neighboring frames of a video sequence using the `estimateGeometricTransform2D` function applied to point correspondences between two images. Second, we warp the video frames to achieve a 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)');
```



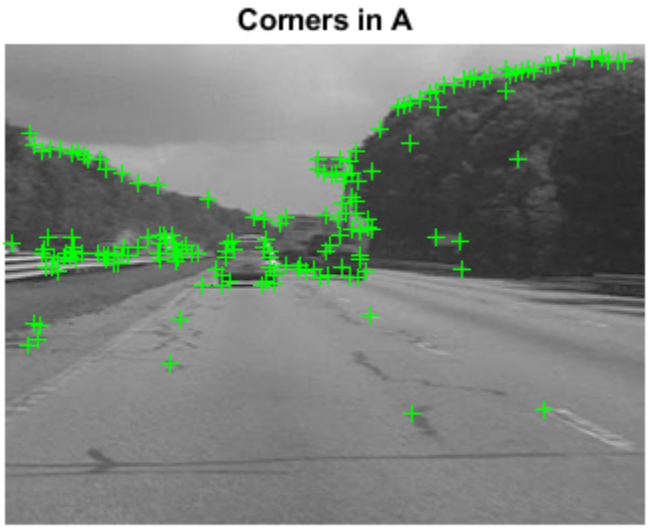
Step 2. Collect Salient Points from Each Frame

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.

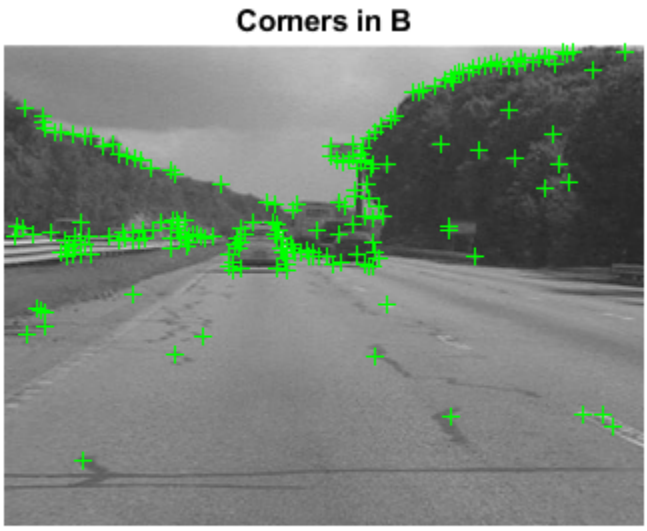
In this step we produce these candidate points for each frame. To have the best chance that these points will have corresponding points in the other frame, we want points around salient image features such as corners. For this we use the `detectFASTFeatures` function, which implements one of the fastest 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');
```




```
figure; imshow(imgB); hold on;
plot(pointsB);
title('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.

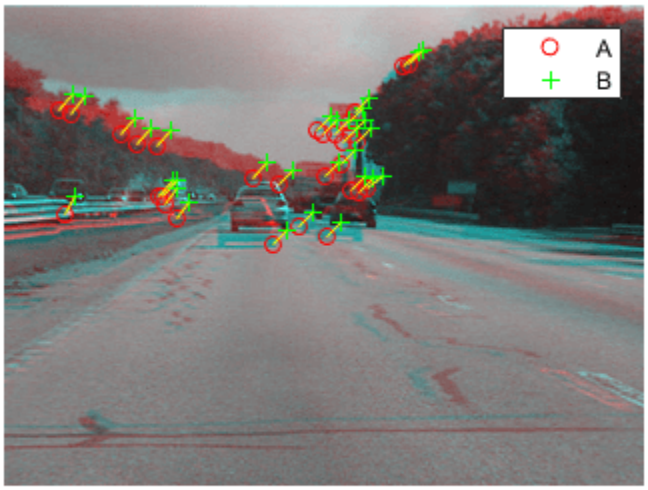
```
% Extract FREAK descriptors for the corners
[featuresA, pointsA] = extractFeatures(imgA, pointsA);
[featuresB, pointsB] = extractFeatures(imgB, pointsB);
```

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, $i = 0, 1, 2...$, 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');
```

Step 6. Run on the Full Video

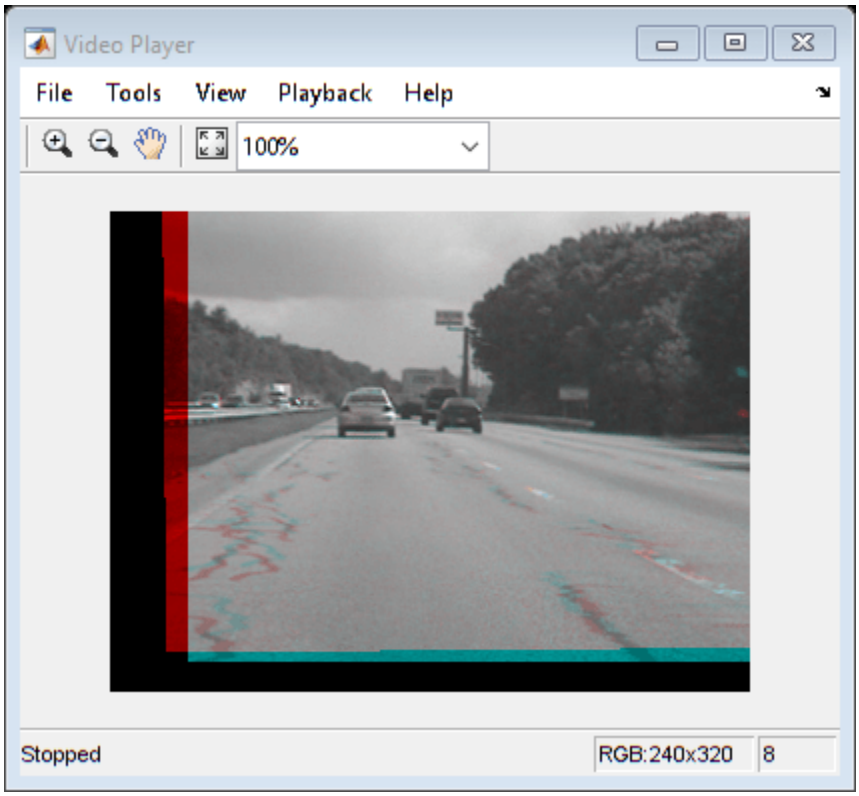
Now we apply the above steps to smooth a video sequence. For readability, the above procedure of estimating the transform between two images has been placed in the MATLAB® function cvexEstStabilizationTform. The function cvexTformToSRT also converts a general affine transform into a scale-rotation-translation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

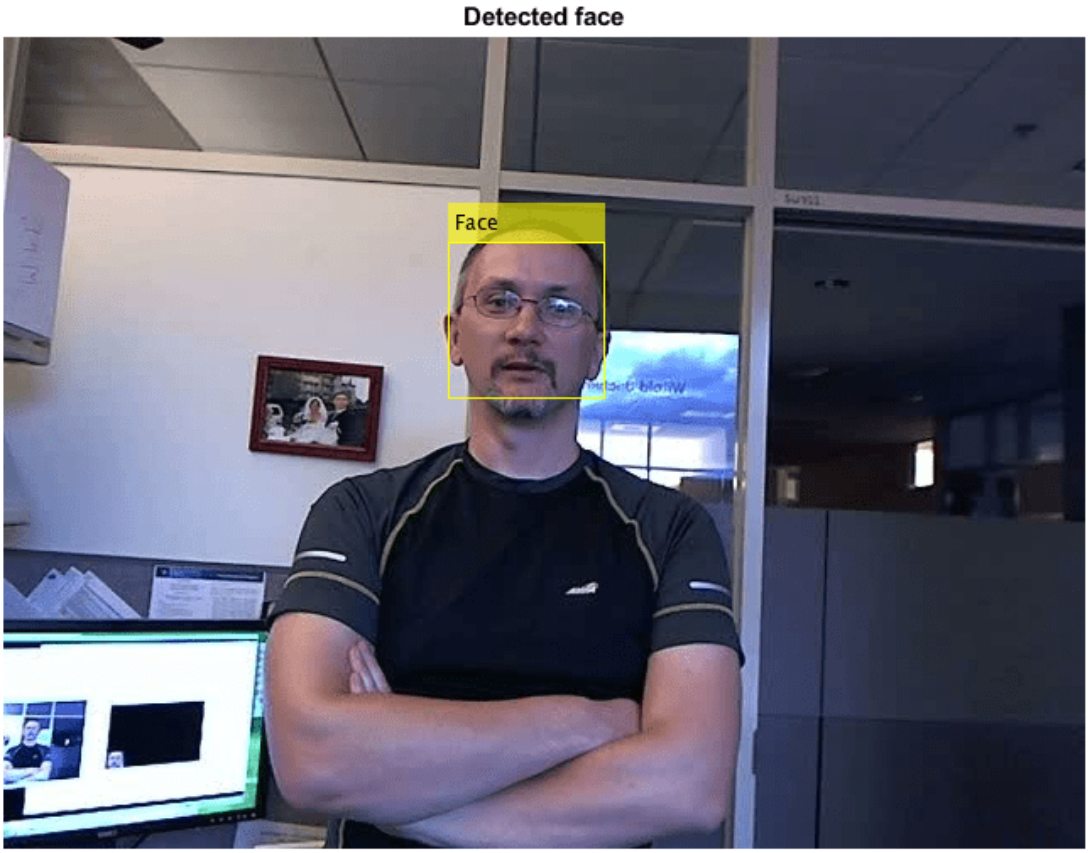
Object detection and tracking are important in many computer vision applications including activity recognition, automotive safety, and surveillance. In this example, you will develop a simple face tracking system by dividing the tracking problem into three separate problems:

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.

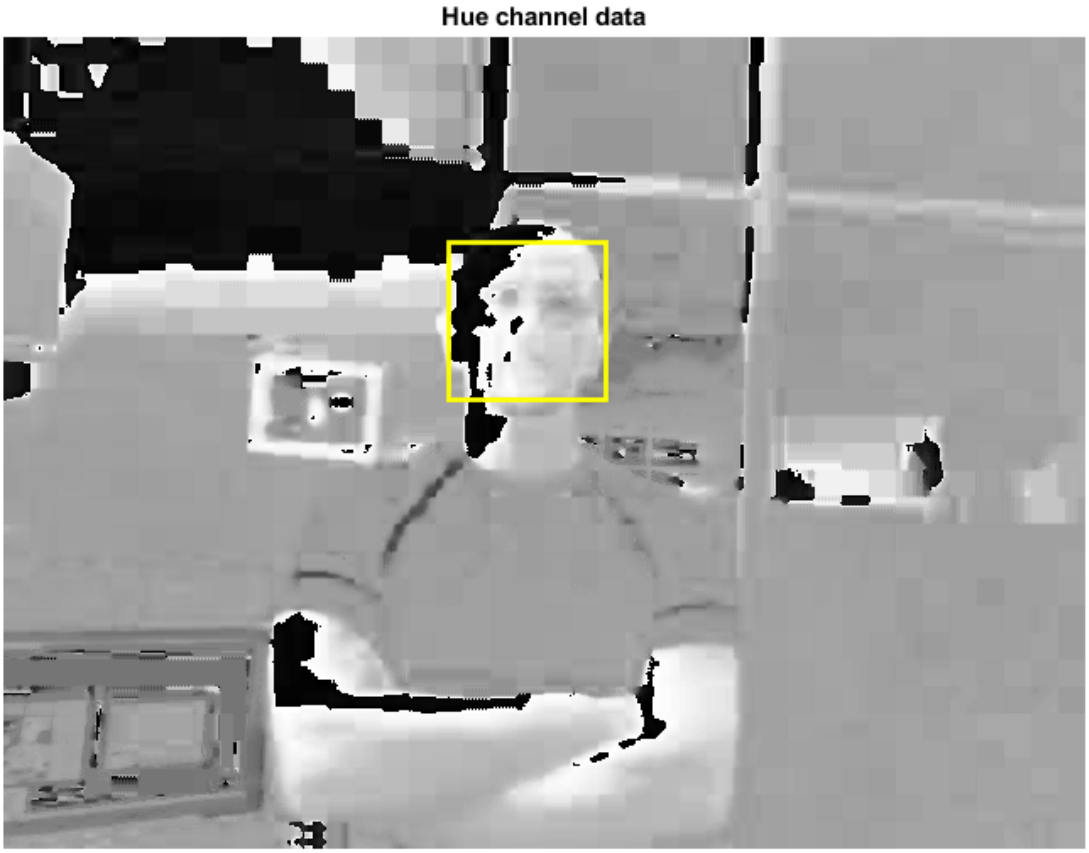
Step 2: Identify Facial Features To Track

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])
```



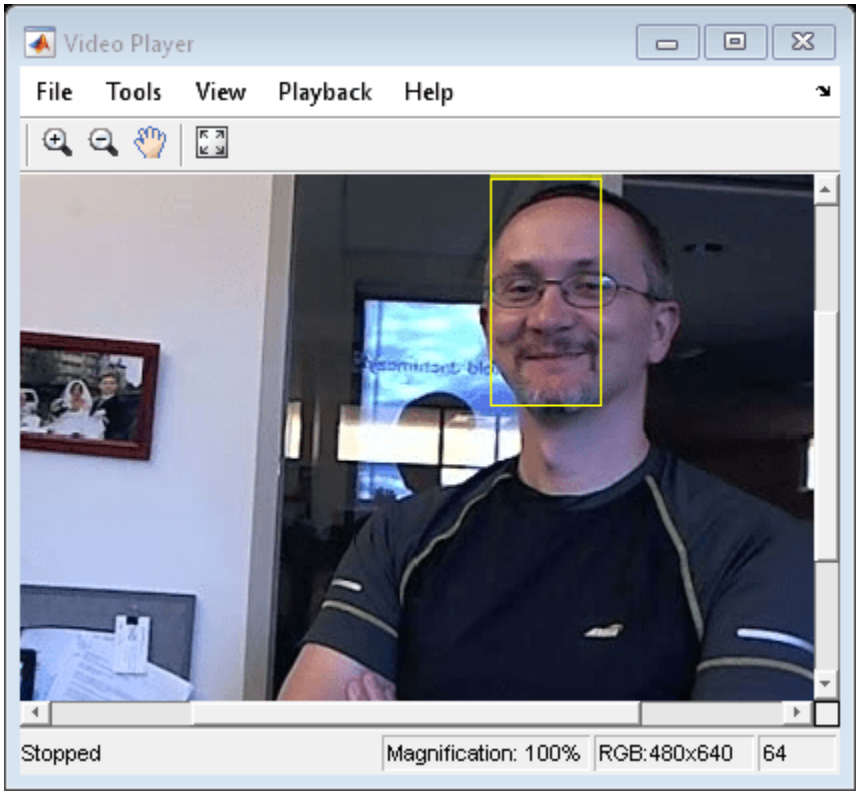
Step 3: Track the Face

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);
noseBBox     = 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, noseBBox(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

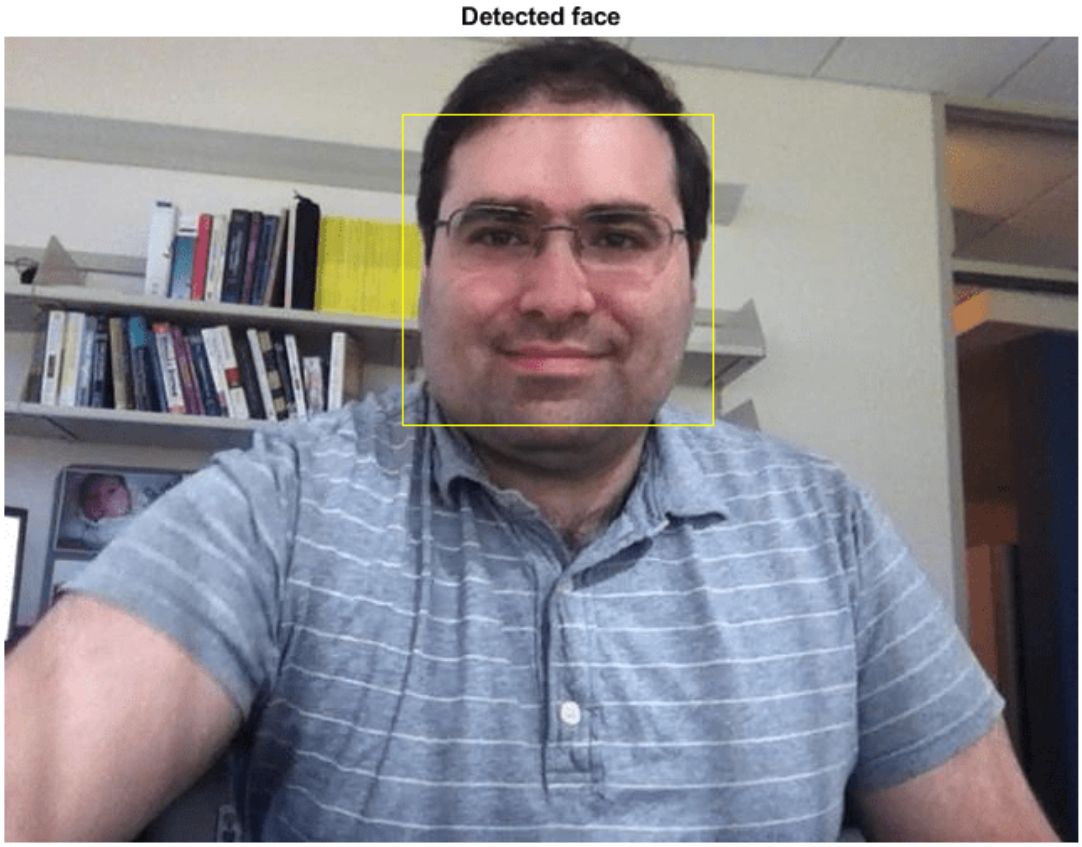
Object detection and tracking are important in many computer vision applications including activity recognition, automotive safety, and surveillance. In this example, you will develop a simple face tracking system by dividing the tracking problem into three parts:

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');
```




```
% 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.

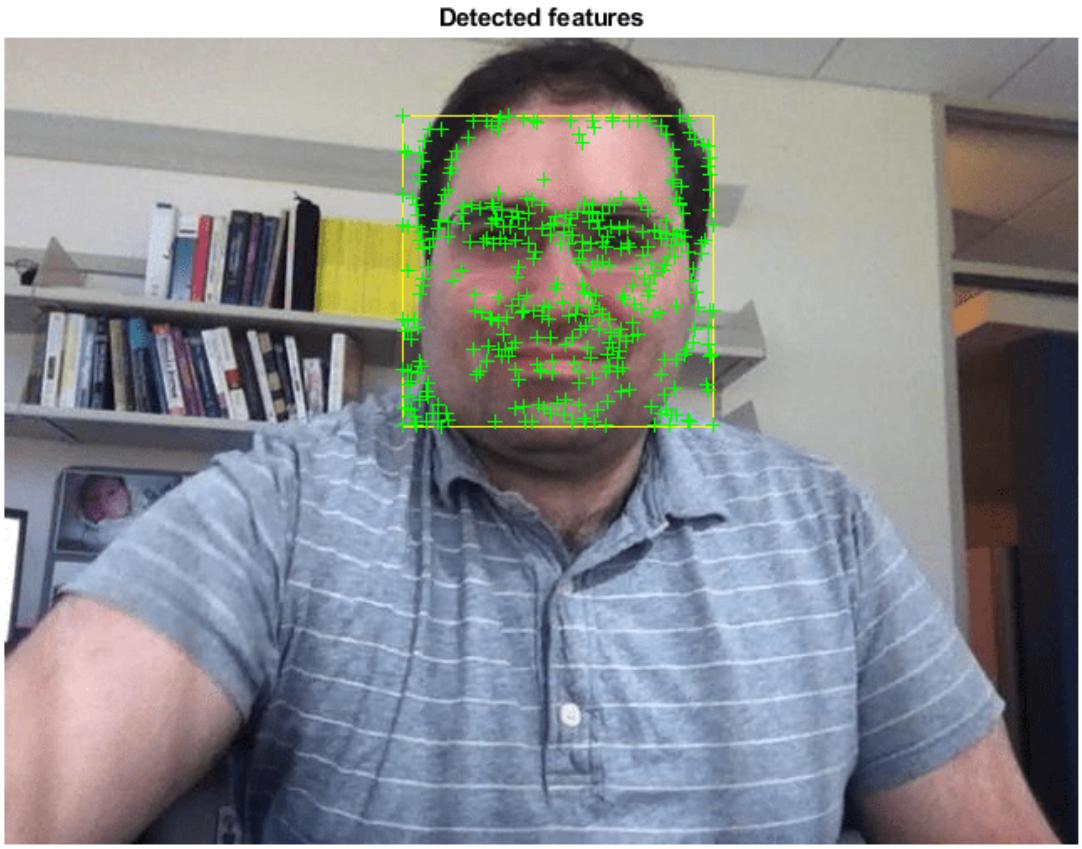
Identify Facial Features To Track

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.

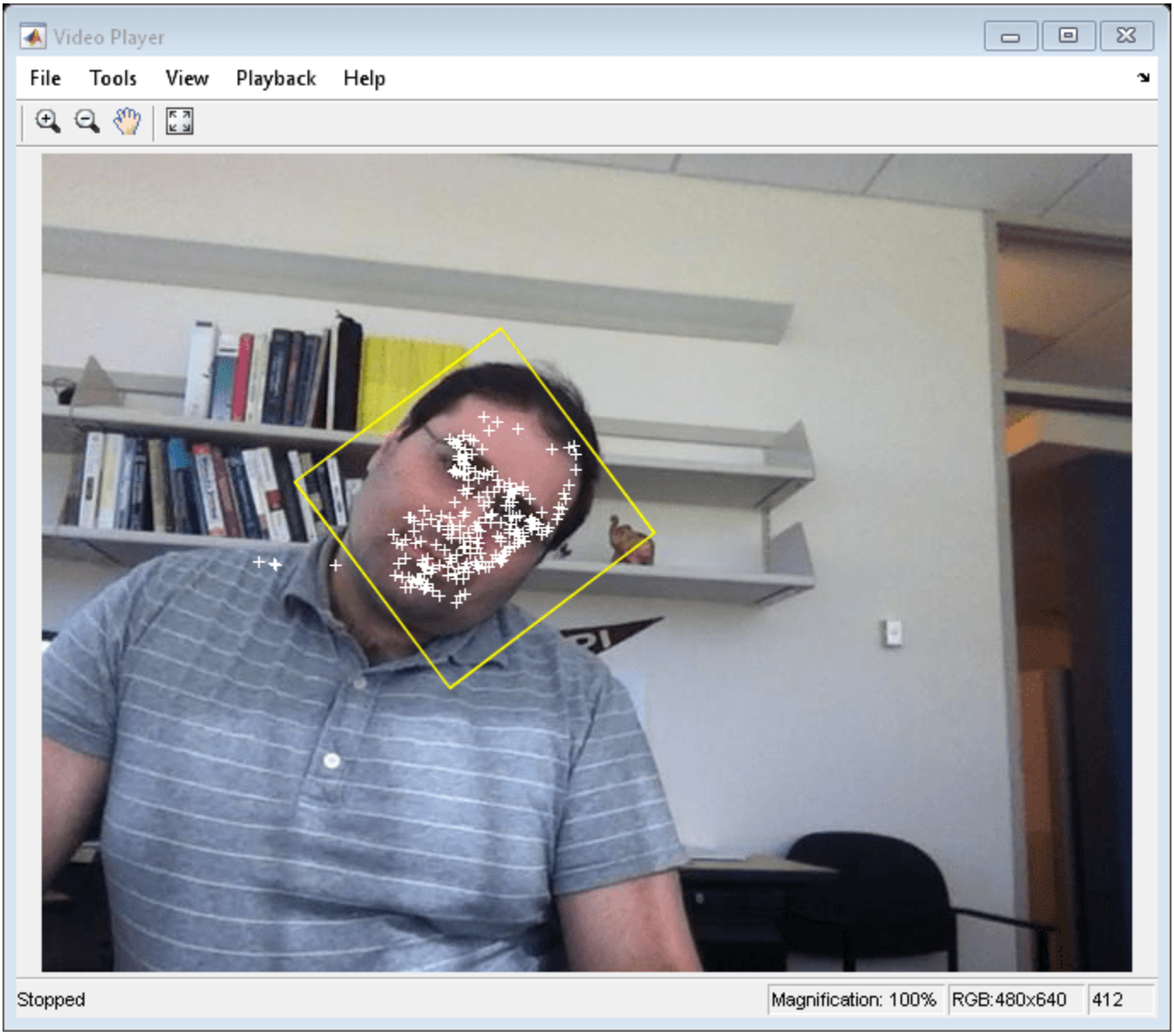
```
videoPlayer  = vision.VideoPlayer('Position',...
    [100 100 [size(videoFrame, 2), size(videoFrame, 1)]+30]);
```

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]);
```

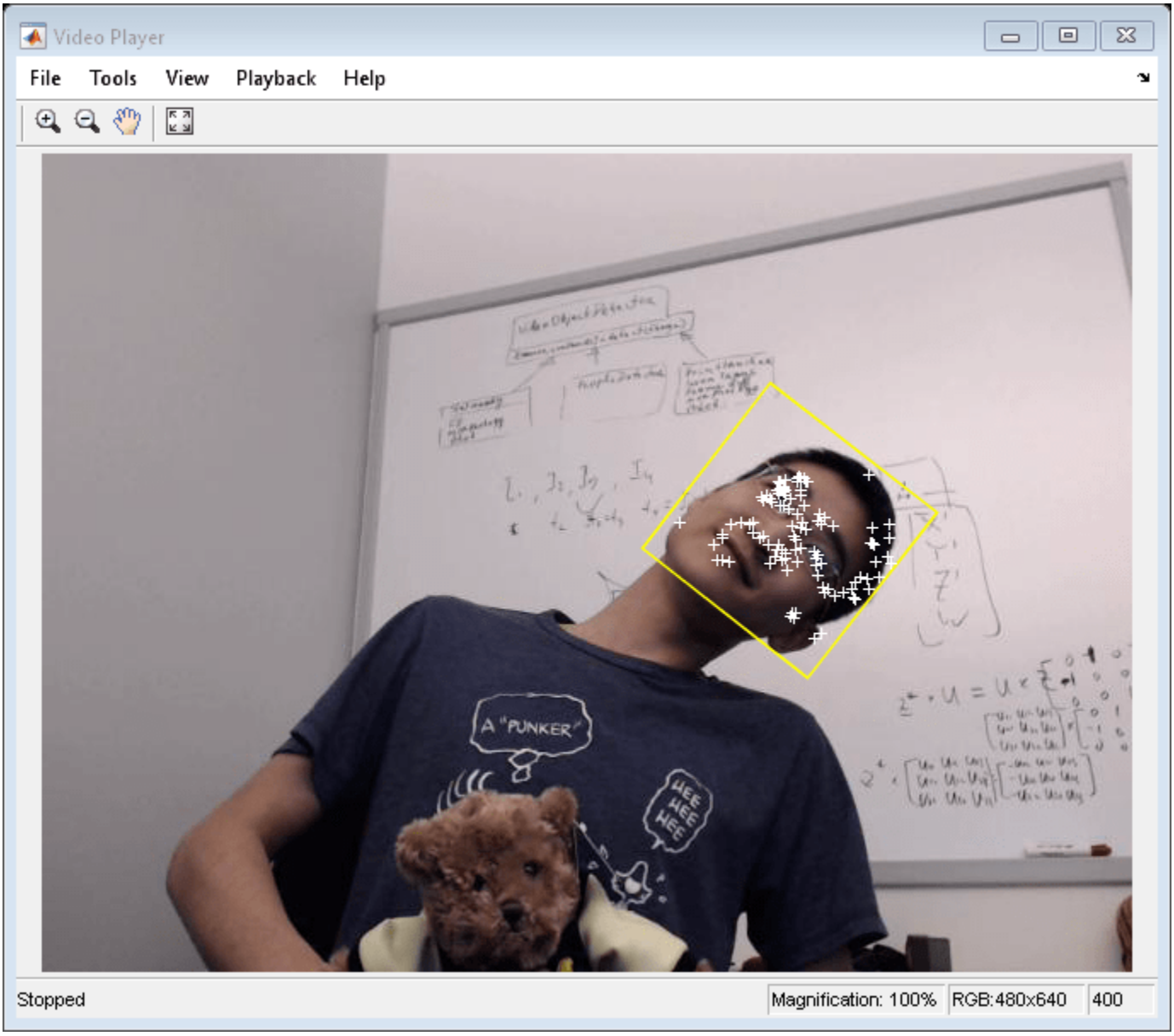
Detection and Tracking

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.

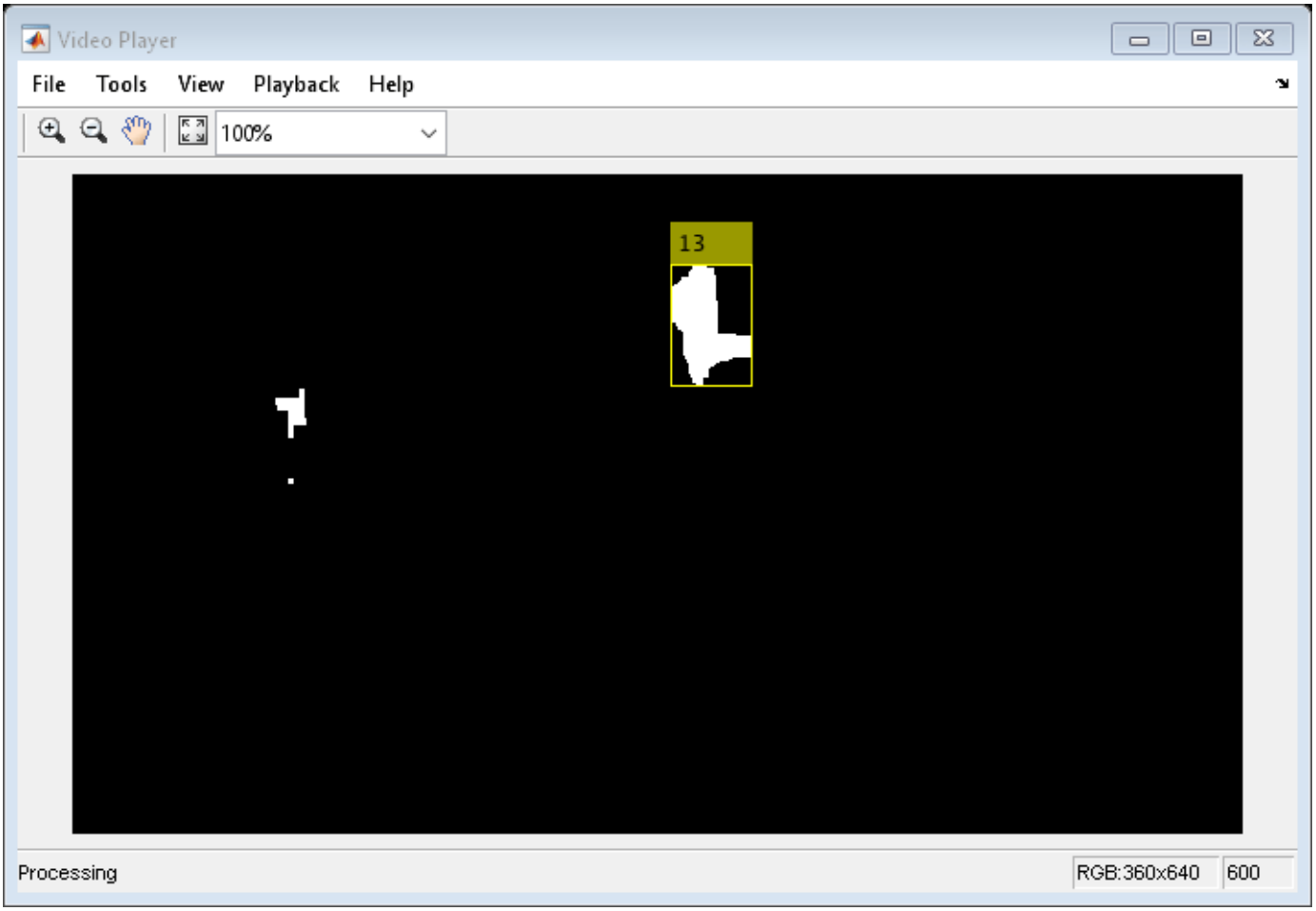
Each track keeps count of the number of consecutive frames, where it remained unassigned. If the count exceeds a specified threshold, the example assumes that the object left the field of view and it deletes the track.

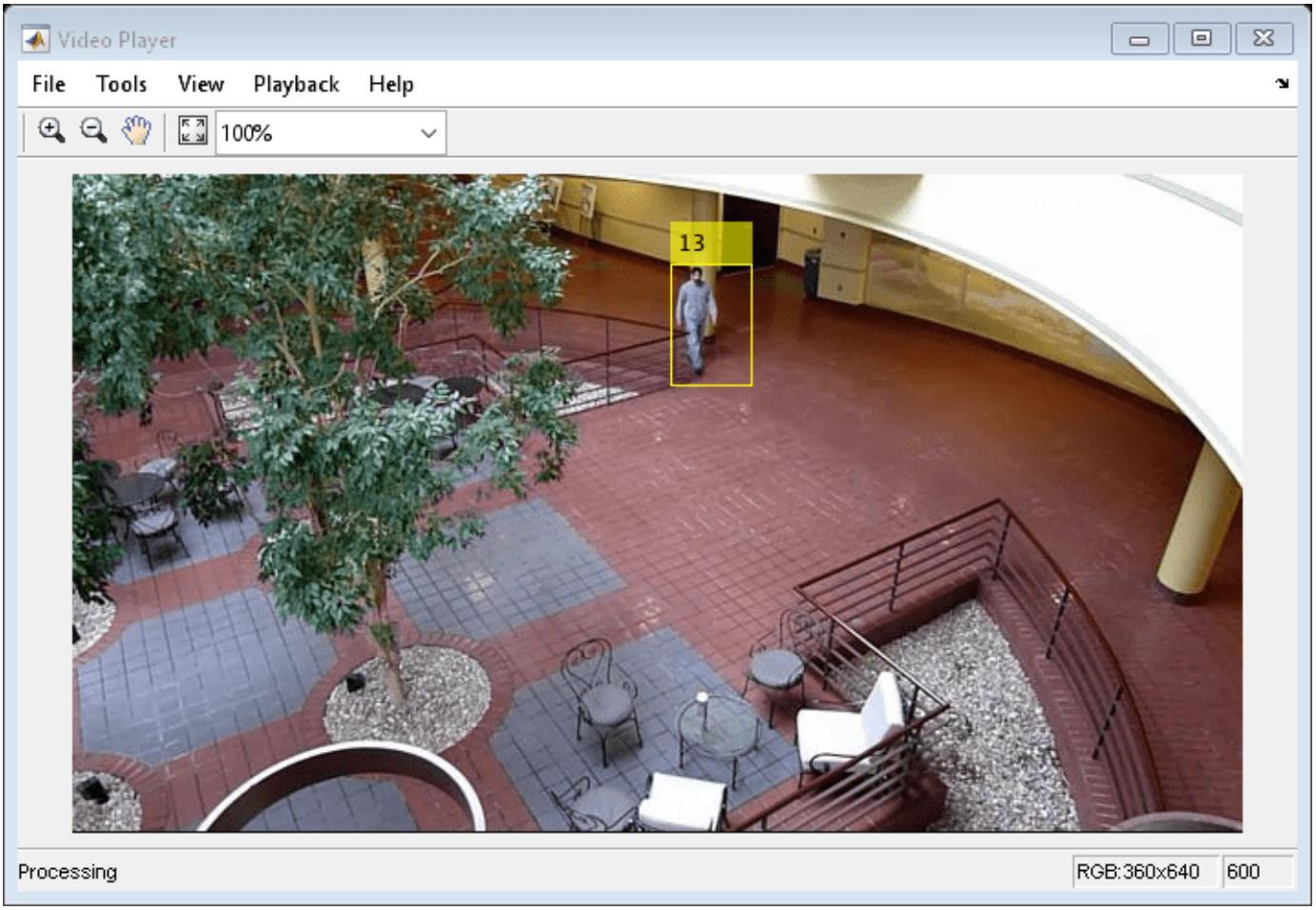
For more information please see “Multiple Object Tracking” on page 18-2.

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 MxN 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 x 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.

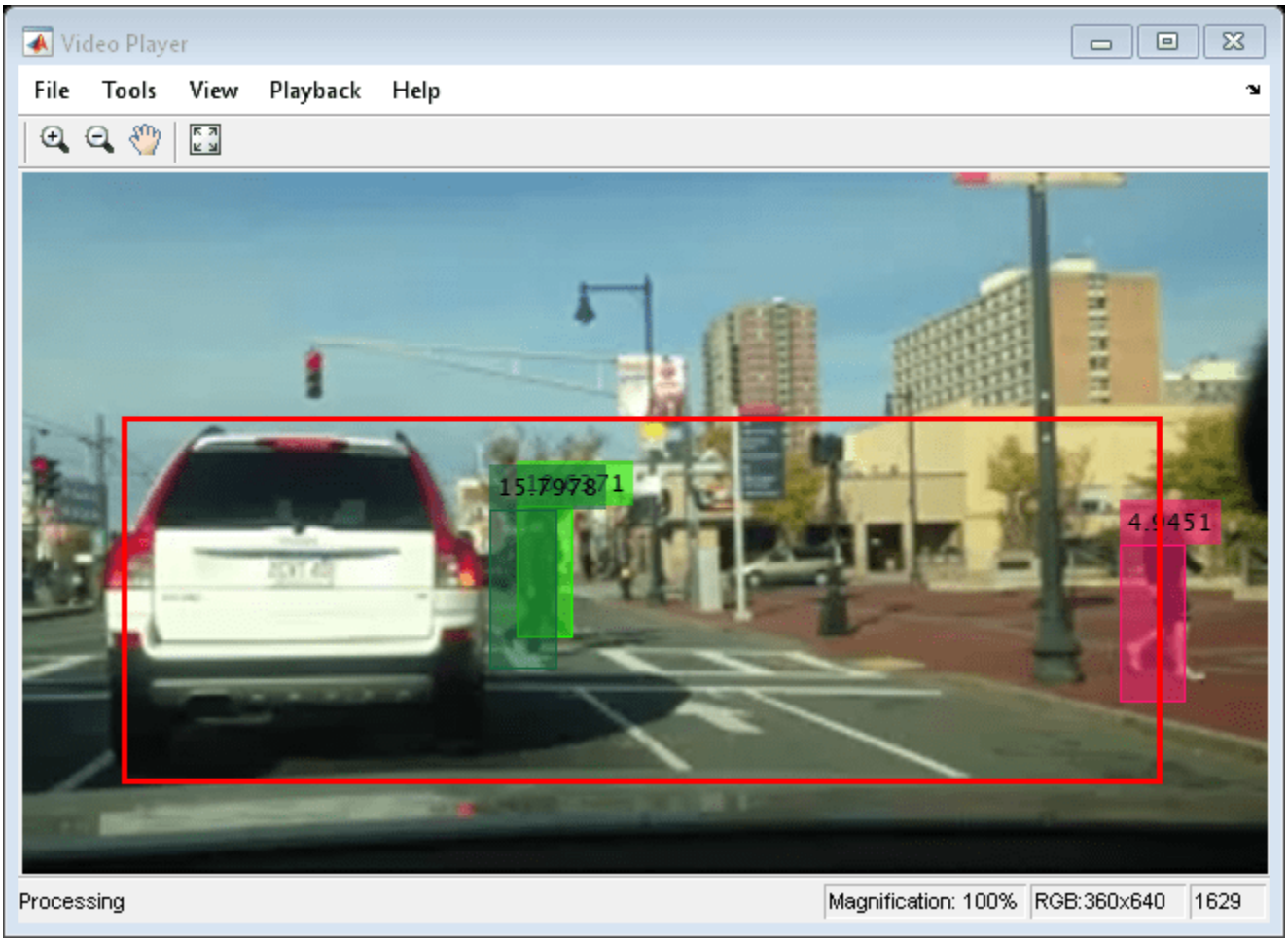
Overview

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 18-2.

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 pedestr
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 i
option.gatingThresh         = 0.9;              % A threshold to reject a candidate match betwee
option.gatingCost           = 100;              % A large value for the assignment cost matrix th
option.costOfNonAssignment  = 10;               % A tuning parameter to control the likelihood of
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 req
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();
```

```
        updateUnassignedTracks();
        deleteLostTracks();
        createNewTracks();

        displayTrackingResults();

        % Exit the loop if the video player figure is closed.
        if ~isOpen(obj.videoPlayer)
            break;
        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.
    resizeRatio = 1.5;
```

```
        frame = imresize(frame, resizeRatio, '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) / resizeRatio;
        y = (bboxes(:,2)-1) / resizeRatio + 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 MxN 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 x 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.
```

```
            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

            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

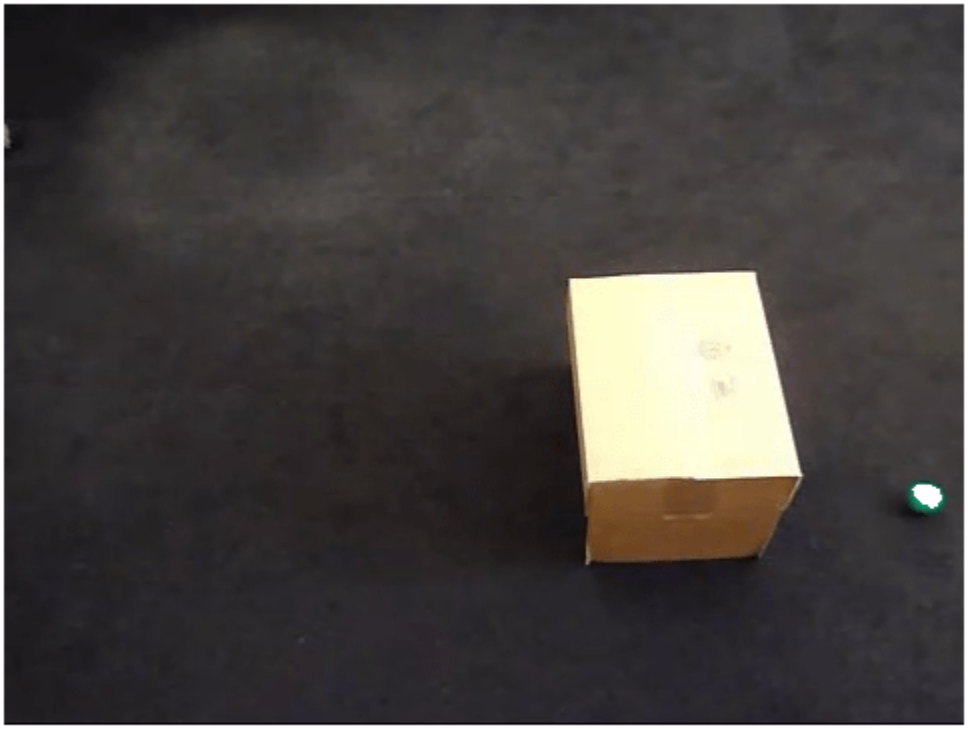
The Kalman filter has many uses, including applications in control, navigation, computer vision, and time series econometrics. This example illustrates how to use the Kalman filter for tracking objects and focuses on three important features:

- 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.

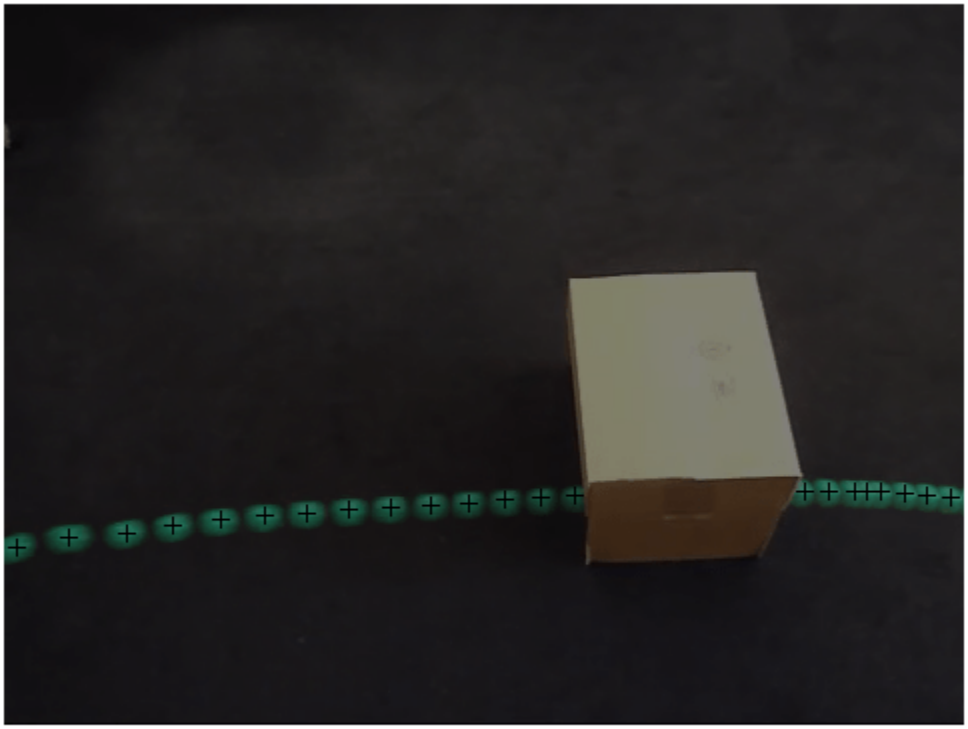
```
showDetections();
```



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.

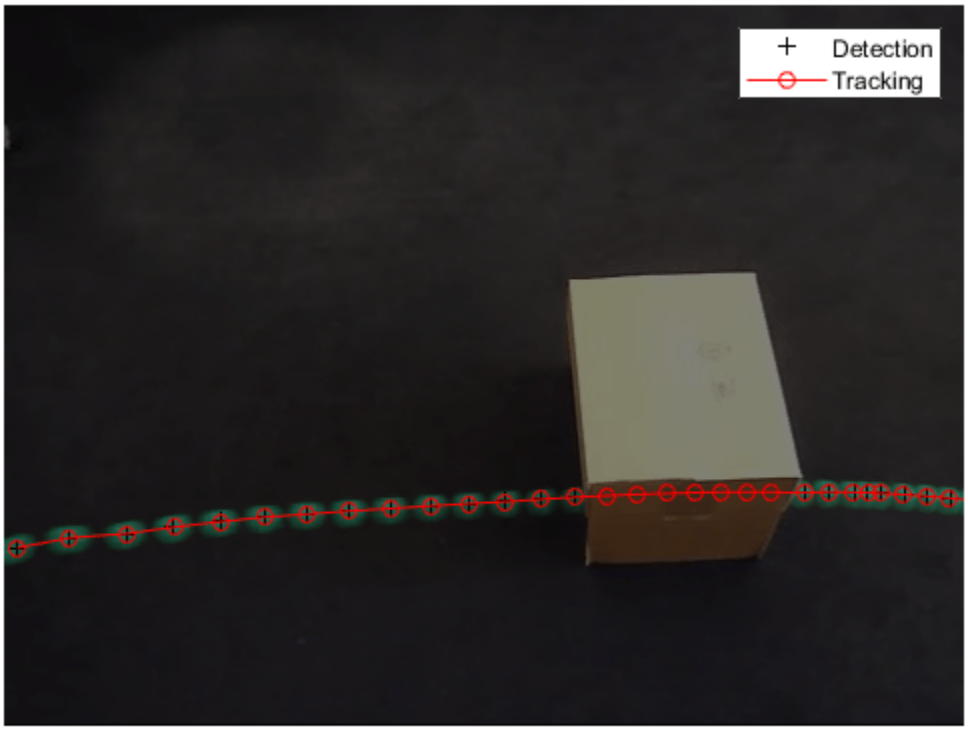
```
function trackSingleObject(param)
  % Create utilities used for reading video, detecting moving objects,
  % and displaying the results.
  utilities = createUtilities(param);

  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

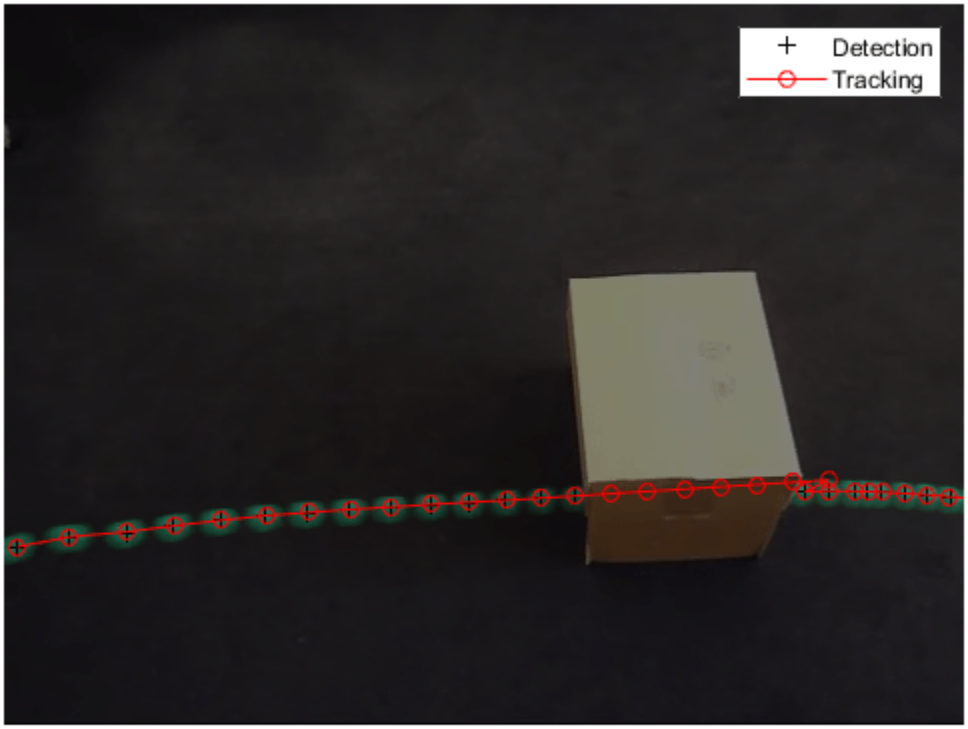
Configuring the Kalman filter can be very challenging. Besides basic understanding of the Kalman filter, it often requires experimentation in order to come up with a set of suitable configuration parameters. The `trackSingleObject` function, defined above, helps you to explore the various configuration 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
```

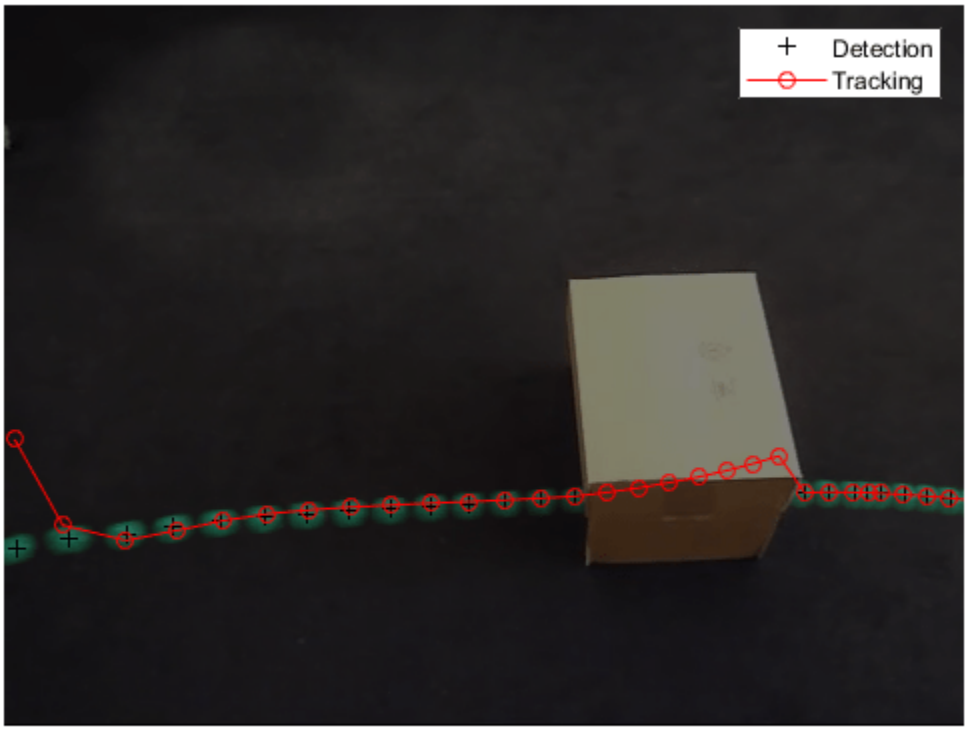


Notice that the ball emerged in a spot that is quite different from the predicted location. From the time when the ball was released, it was subject to constant deceleration due to resistance from the carpet. Therefore, constant acceleration model was a better choice. If you kept the constant velocity 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.

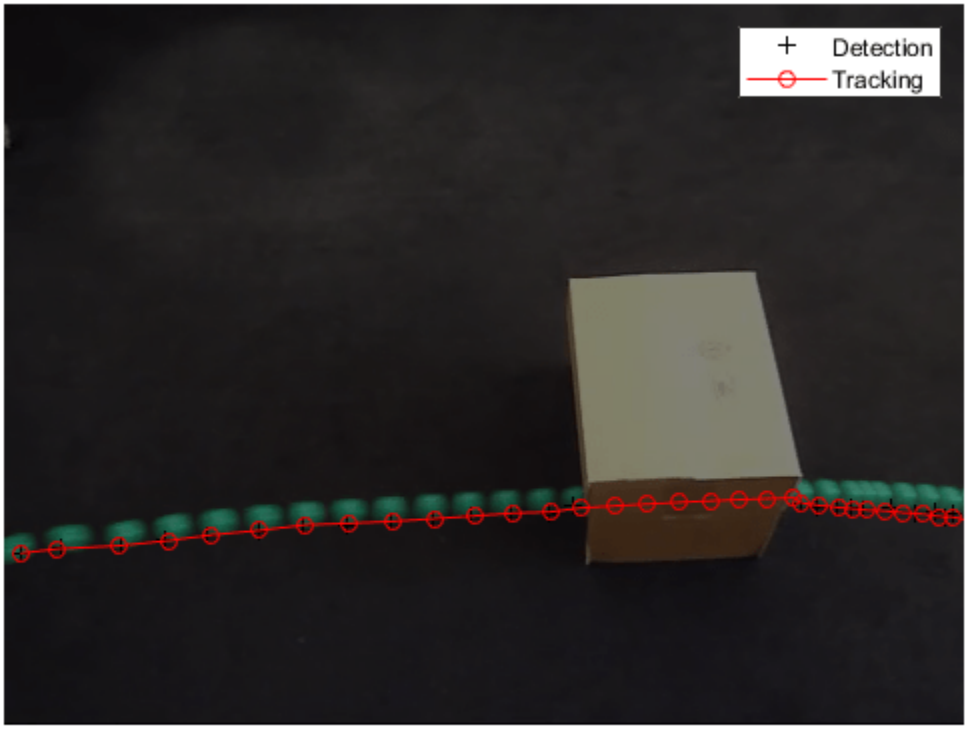
```
param = getDefaultParameters();  % get parameters that work well
param.initialLocation = [0, 0];  % location that's not based on an actual detection
param.initialEstimateError = 100*ones(1,3); % use relatively small values

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 8-31.

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 8-31.

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.

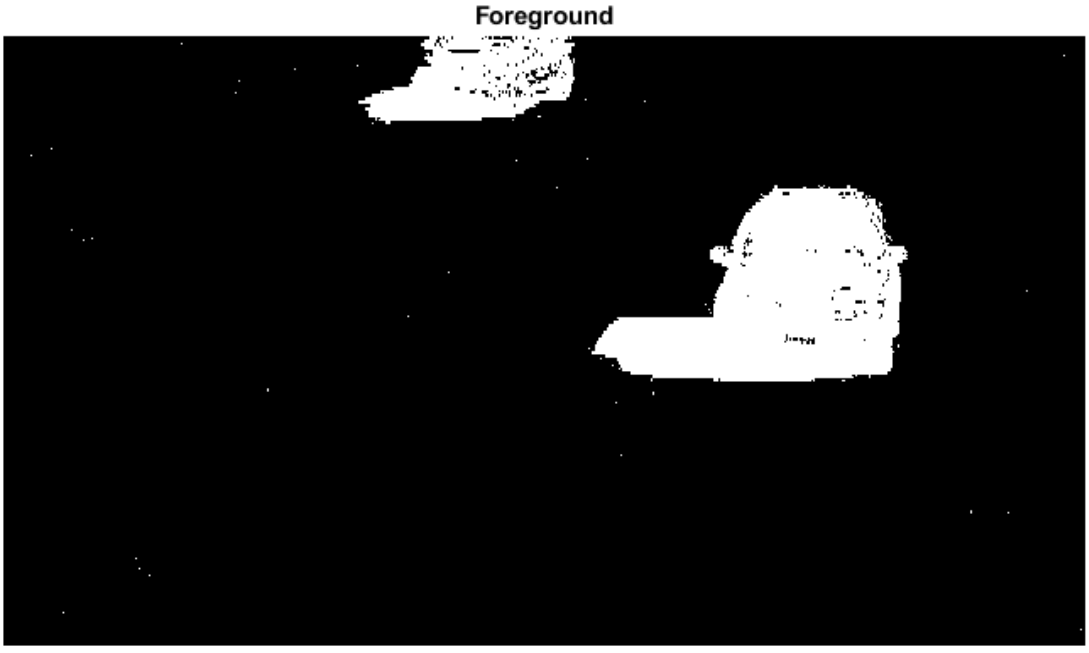
The foreground detector requires a certain number of video frames in order to initialize the Gaussian mixture model. This example uses the first 50 frames to initialize three Gaussian modes in the mixture model.

```
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');
```

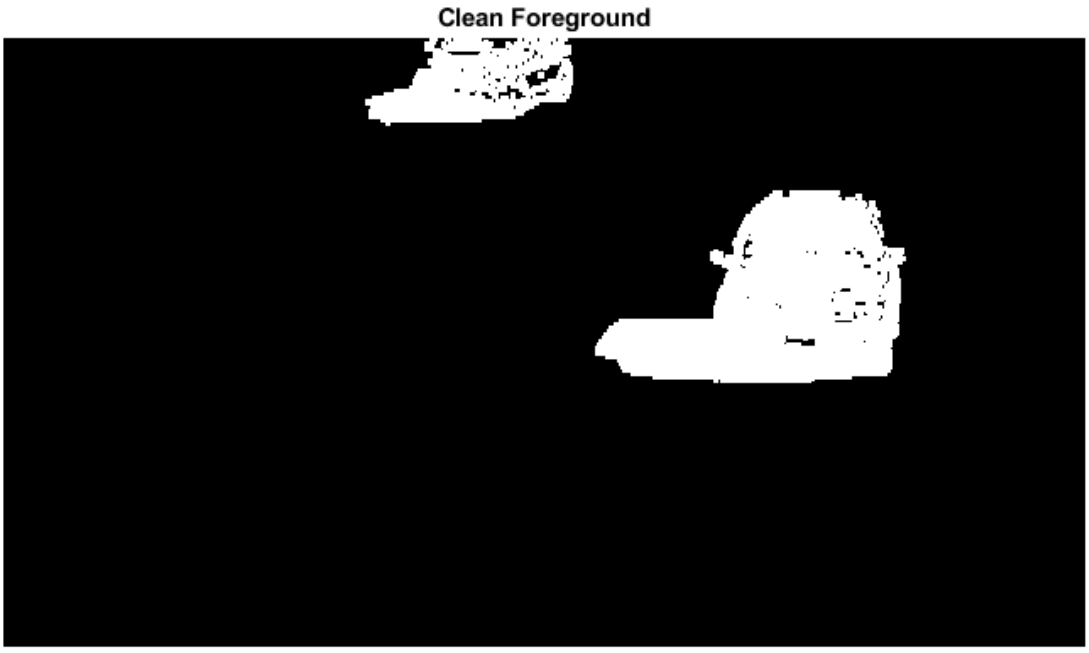

```
figure; imshow(foreground); title('Foreground');
```



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.

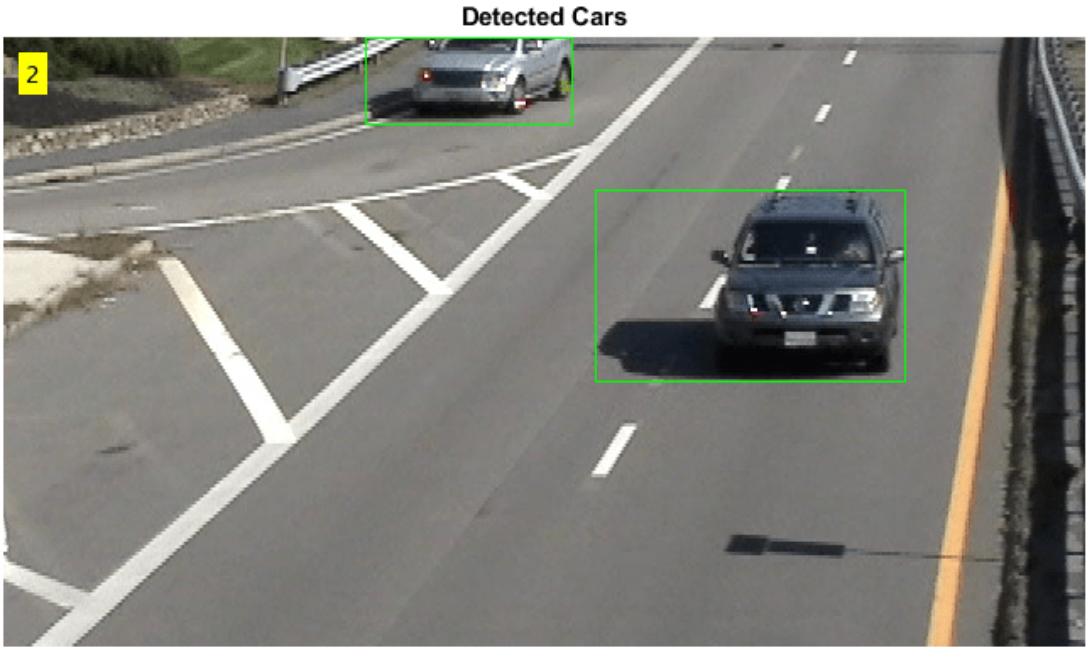
```
blobAnalysis = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
    'AreaOutputPort', false, 'CentroidOutputPort', false, ...
    'MinimumBlobArea', 150);
bbox = step(blobAnalysis, filteredForeground);
```

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');
```

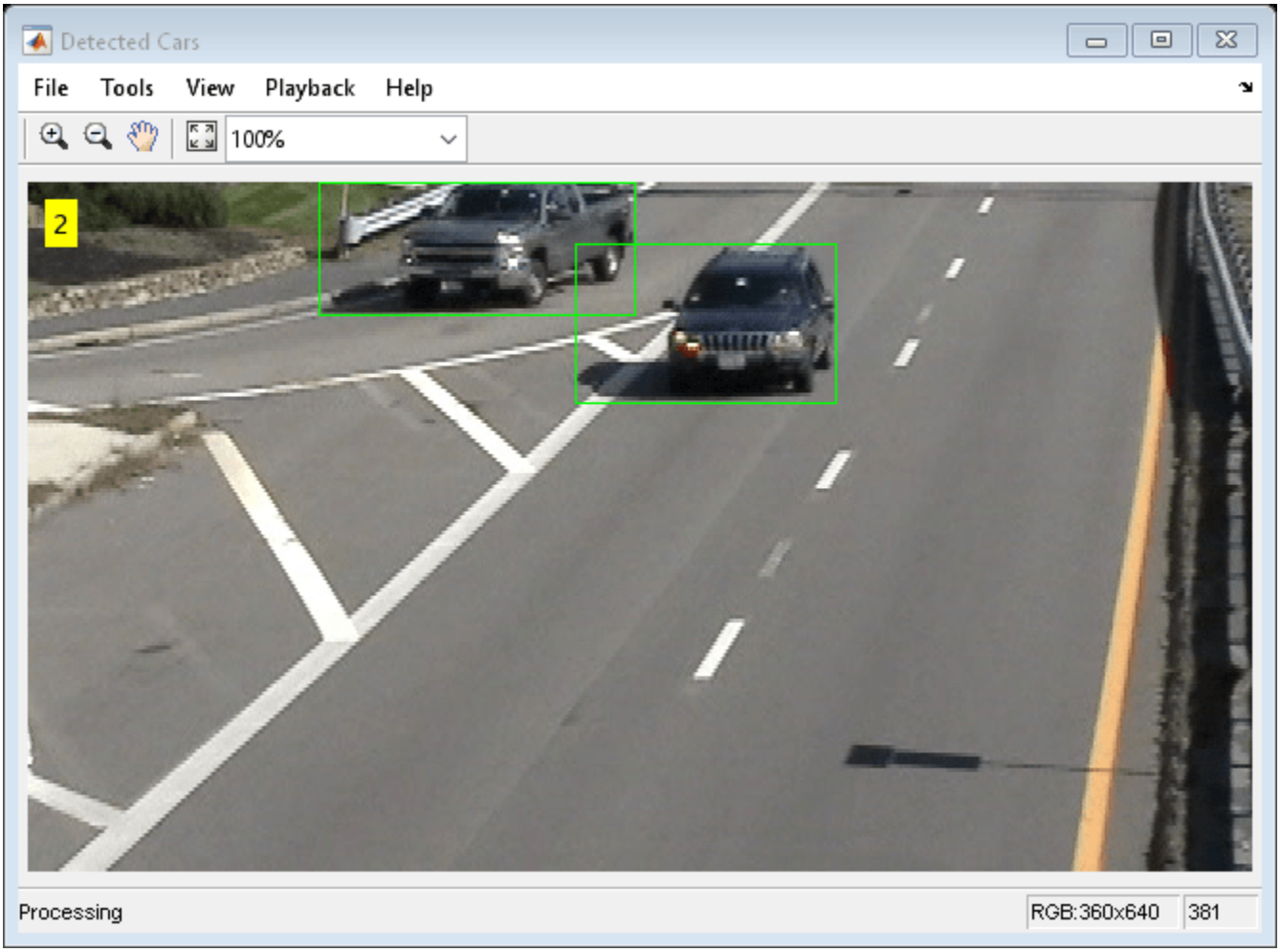


Step 3 - Process the Rest of Video Frames

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.

Labelers

- “Create Automation Algorithm for Labeling” on page 9-2
- “Label Large Images in the Image Labeler” on page 9-6
- “Label Pixels for Semantic Segmentation” on page 9-14
- “Label Objects Using Polygons” on page 9-23
- “Get Started with the Image Labeler” on page 9-30
- “Choose an App to Label Ground Truth Data” on page 9-40
- “Get Started with the Video Labeler” on page 9-44
- “Use Custom Image Source Reader for Labeling” on page 9-57
- “Keyboard Shortcuts and Mouse Actions for Video Labeler” on page 9-59
- “Keyboard Shortcuts and Mouse Actions for Image Labeler” on page 9-63
- “Share and Store Labeled Ground Truth Data” on page 9-67
- “View Summary of Ground Truth Labels” on page 9-73
- “Temporal Automation Algorithms” on page 9-77
- “Blocked Image Automation Algorithms” on page 9-79
- “Use Sublabels and Attributes to Label Ground Truth Data” on page 9-80
- “Training Data for Object Detection and Semantic Segmentation” on page 9-84

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 9-77.

If the algorithm is blocked image based, your custom automation algorithm must also inherit from the `vision.labeler.mixin.BlockedImageAutomation` class. For more details on implementing blocked image algorithms, see “Blocked Image Automation Algorithms” on page 9-79.

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 app. On the app toolstrip, select **Select Algorithm** > **Refresh list**.

Import Existing Algorithm

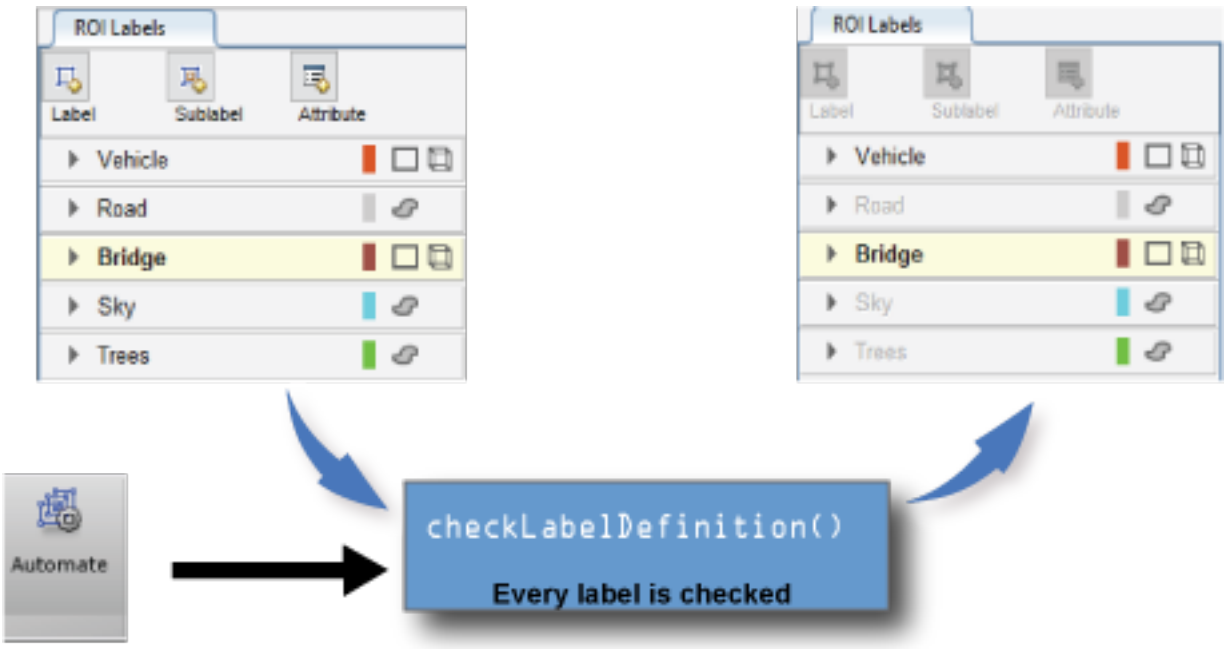
To import an existing custom algorithm into a labeling app, on the app toolstrip, select **Select Algorithm** > **Add Algorithm** > **Import Algorithm** and then refresh the list.

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`.



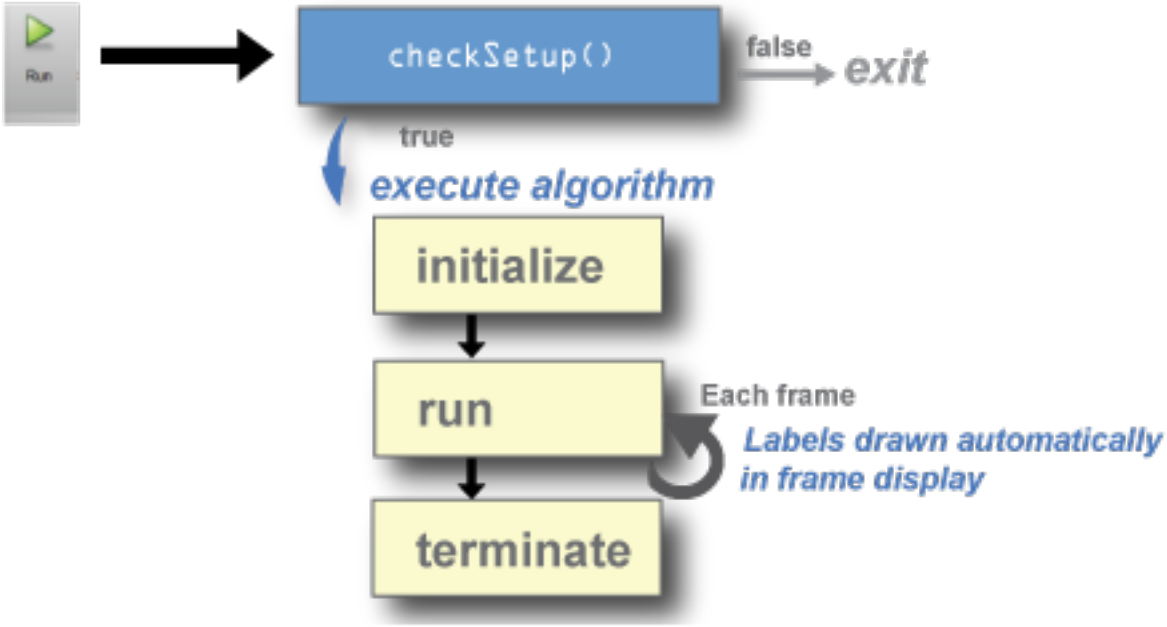
Control Settings

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
Video Labeler | **Image Labeler** | **Ground Truth Labeler** | **Lidar Labeler**

Functions
`vision.labeler.AutomationAlgorithm` | `vision.labeler.mixin.Temporal` | `vision.labeler.mixin.BlockedImageAutomation`

See Also

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)

See Also

More About

- “Get Started with the Image Labeler” on page 9-30
- “Get Started with the Video Labeler” on page 9-44
- “Get Started with the Lidar Labeler” (Lidar Toolbox)
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Temporal Automation Algorithms” on page 9-77
- “Blocked Image Automation Algorithms” on page 9-79

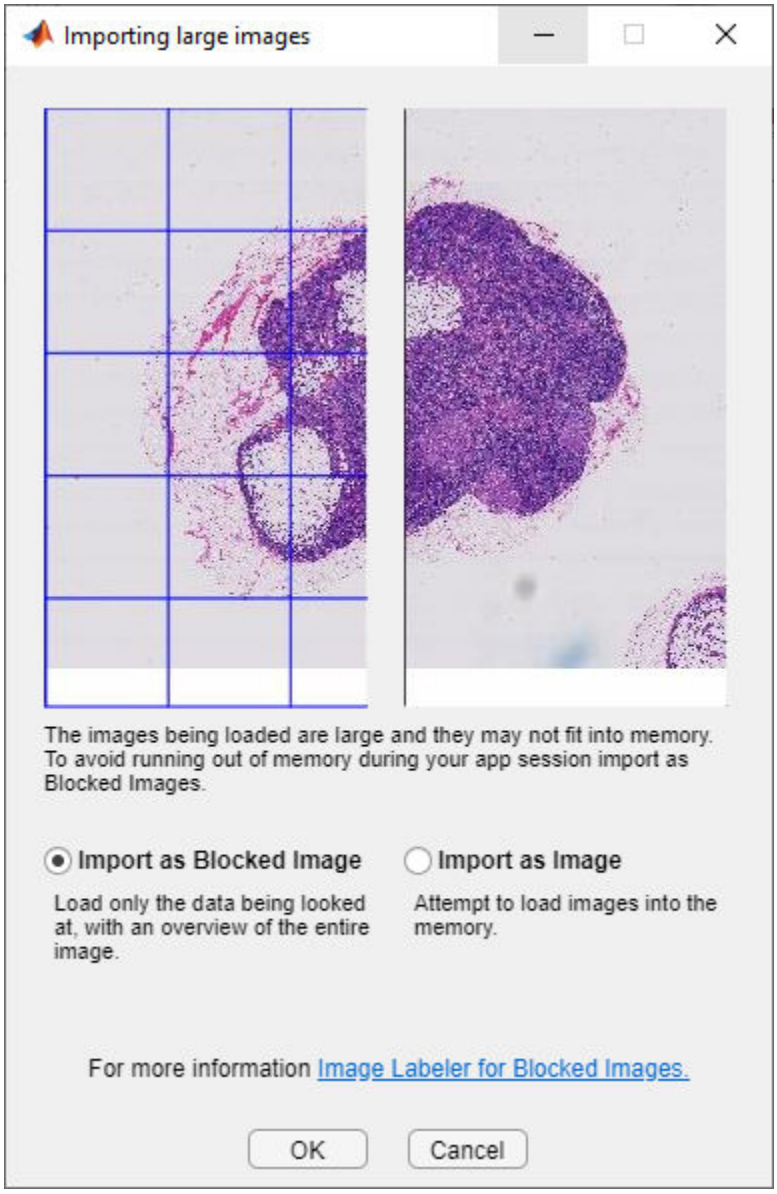
Label Large Images in the Image Labeler

The **Image Labeler** app enables you to label images that might be too large to fit into memory. The app achieves this by converting the large image into a blocked image. A blocked image divides the large image into smaller blocks that can fit in memory.

In the **Image Labeler** app, you can label a blocked image as you would any other image.

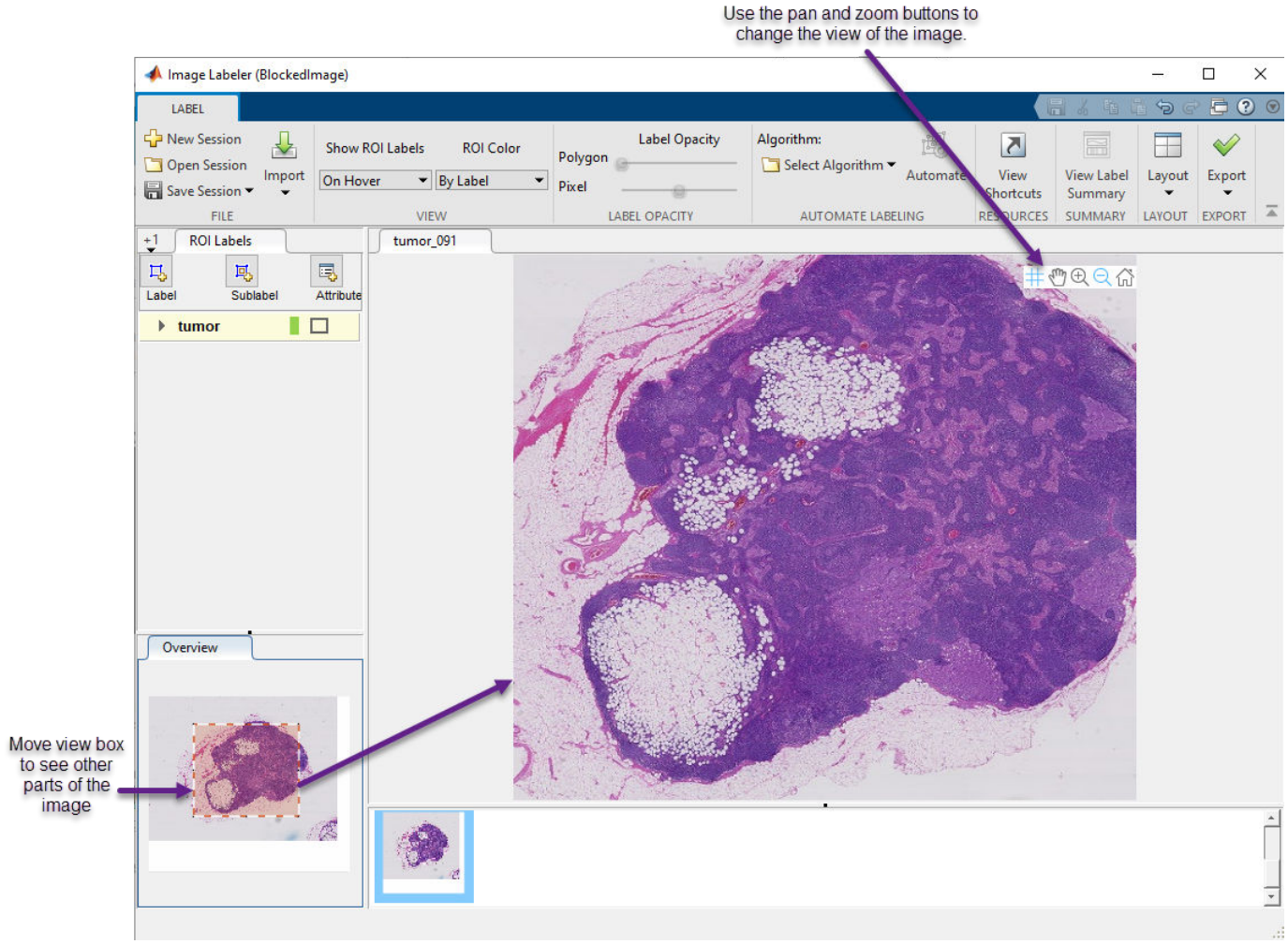
Import Blocked Image into Image Labeler

When loading images into the **Image Labeler** app, if an image appears to be too large to fit into memory (having at least one dimension larger than 8000 pixels) or is a multiresolution image, the app displays the Importing Large Images dialog box, which offers you the option to convert the image into a blocked image. The image on the left side of the dialog box represents a blocked image, with visible grid lines to show how an image is divided into blocks. To choose to use a blocked image, select **Import as Blocked Image** and click **OK**.



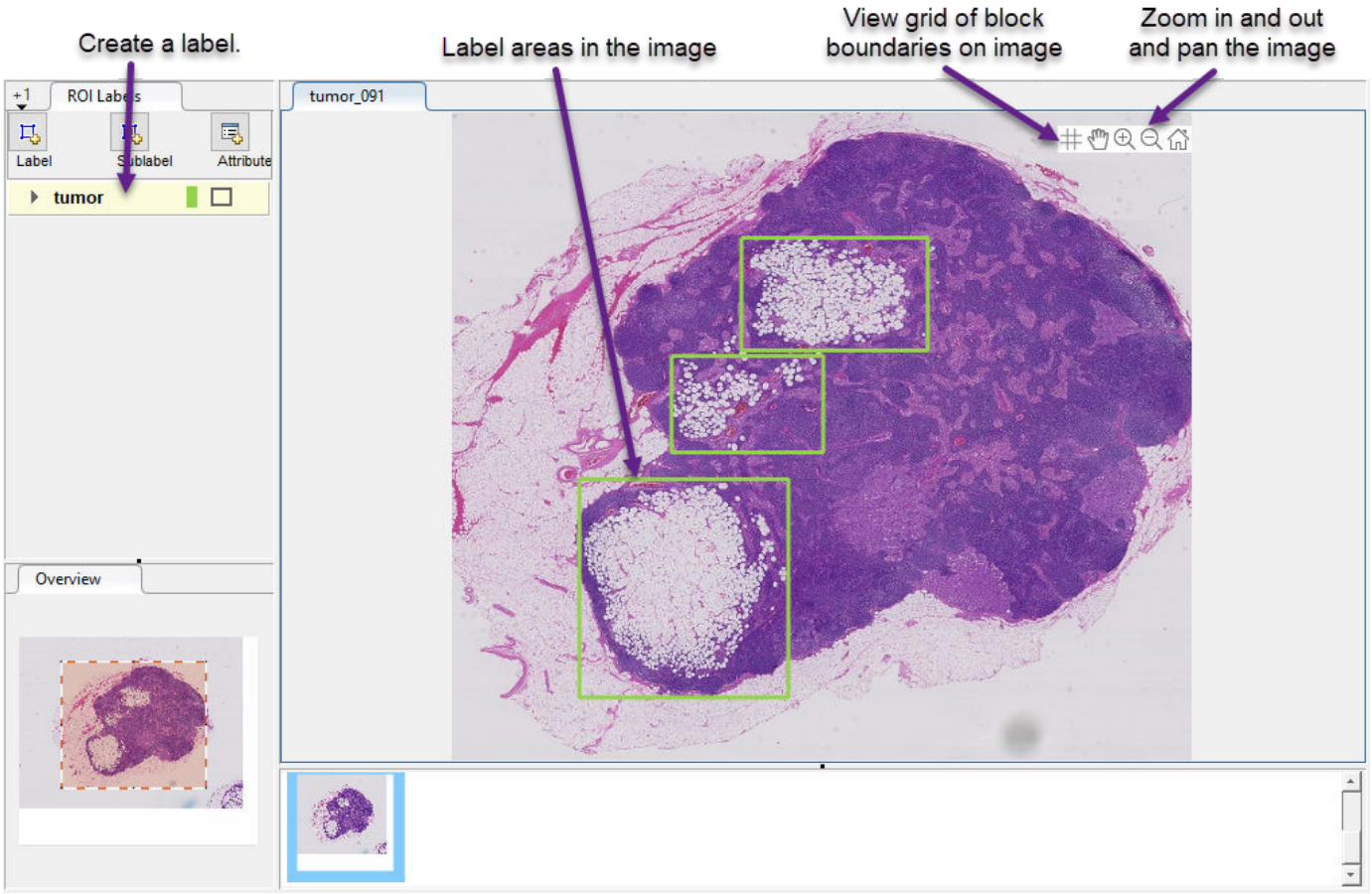
Initially, the app loads blocks that form the center of the image (50% of image height and width). For multiresolution images, the app chooses the resolution to use depending on how much of the image is visible. As you zoom in on an image, the app switches to a finer resolution. For more information about displaying multiresolution images, see `bigimageshow`.

When using blocked images, the app also provides a view of the entire image in the **Overview** pane. In the **Overview** pane, the display overlays a box on the image that shows the portion of the image that is currently visible in the main view pane in the center of the app window. To view other parts of the image in the app, drag the view box to a different part of the image in the **Overview** pane. To get a more detailed view of the image, use the zoom buttons in the axes toolbar that appears when you move the mouse over the image in the main view pane. The axes toolbar also contains a button to enable a view of the block boundaries grid over the image. The current resolution level used by the app can affect the visibility of these block boundaries.



Work with Blocked Images in the Image Labeler

Once you load a blocked image into the **Image Labeler** app, you can work with the blocked image as you would with nonblocked images. You can create labels, sublabels, and attributes and, when you are finished labeling, you can export the labels as a `groundTruth` object.



Limitations

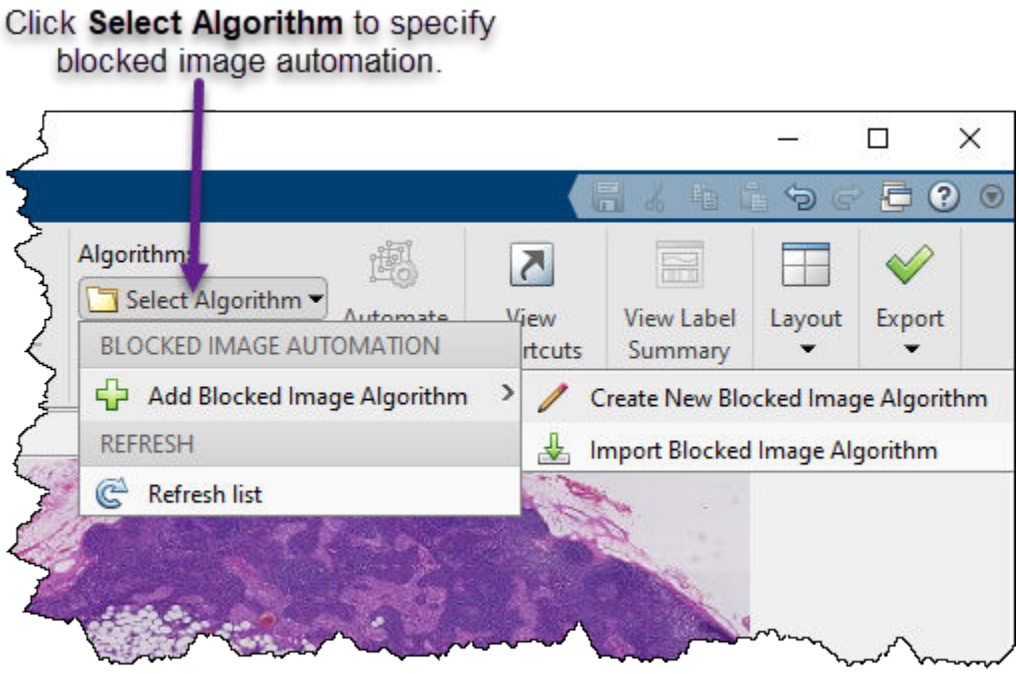
Using blocked images enables you to label images in the app that do not fit in memory. However, there are some limitations with use of blocked images in the **Image Labeler** app.

- Pixel labeling is not supported. You can only create labels using ROI shapes, such as rectangles, lines, and projected cuboid polygons.
- When working with a multiresolution image, the images at every resolution must be in registered to each other, that is, in alignment.
- Thumbnails of images displayed in the app cannot be rotated.

Use Blocked Image Automation with Images

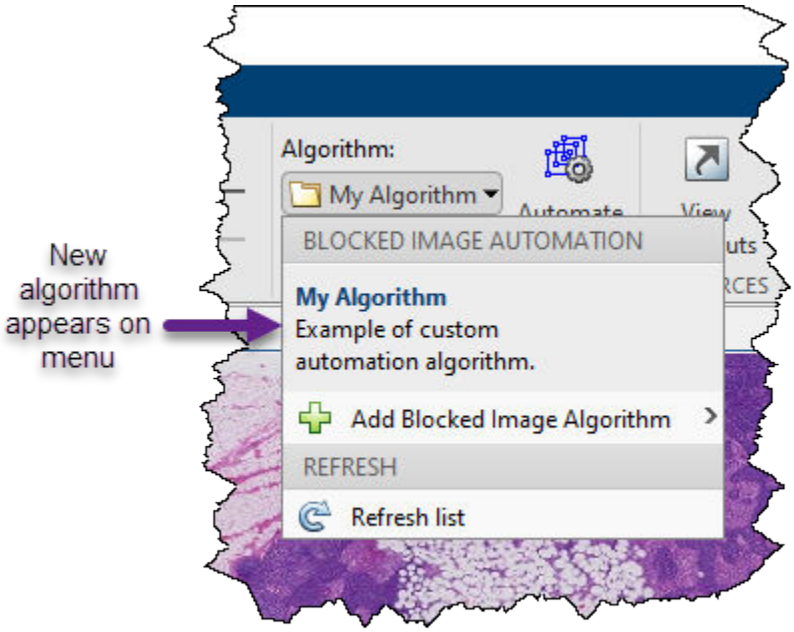
With blocked images, as with nonblocked images, you can take advantage of automated processing to help with the labeling process. For example, with medical images, you can do automatic tumor detection using blocked image automation. You can train a object detector deep learning network and use that in the app to automatically label tumors with bounding boxes.

To use blocked image automation, click **Select Algorithm**, select **Add Blocked Image Algorithm**, and choose to either create a new algorithm, by selecting **Create New Blocked Image Algorithm**, or import an existing algorithm, by selecting **Import Blocked Image Algorithm**.



When you choose to create a new blocked image algorithm, the app opens a class file template in the editor. The class subclasses `vision.labeler.mixin.BlockedImageAutomation`. The template contains stubs for standard methods that you must implement for your algorithm, including the `blockedImageAutomationAlgorithm` method. For more information about implementing this method, see `blockedImageAutomationAlgorithm`. Add the code you want the algorithm to execute. For more information about creating labeler automation algorithms and where to save this class file, see “Create Automation Algorithm for Labeling” on page 9-2.

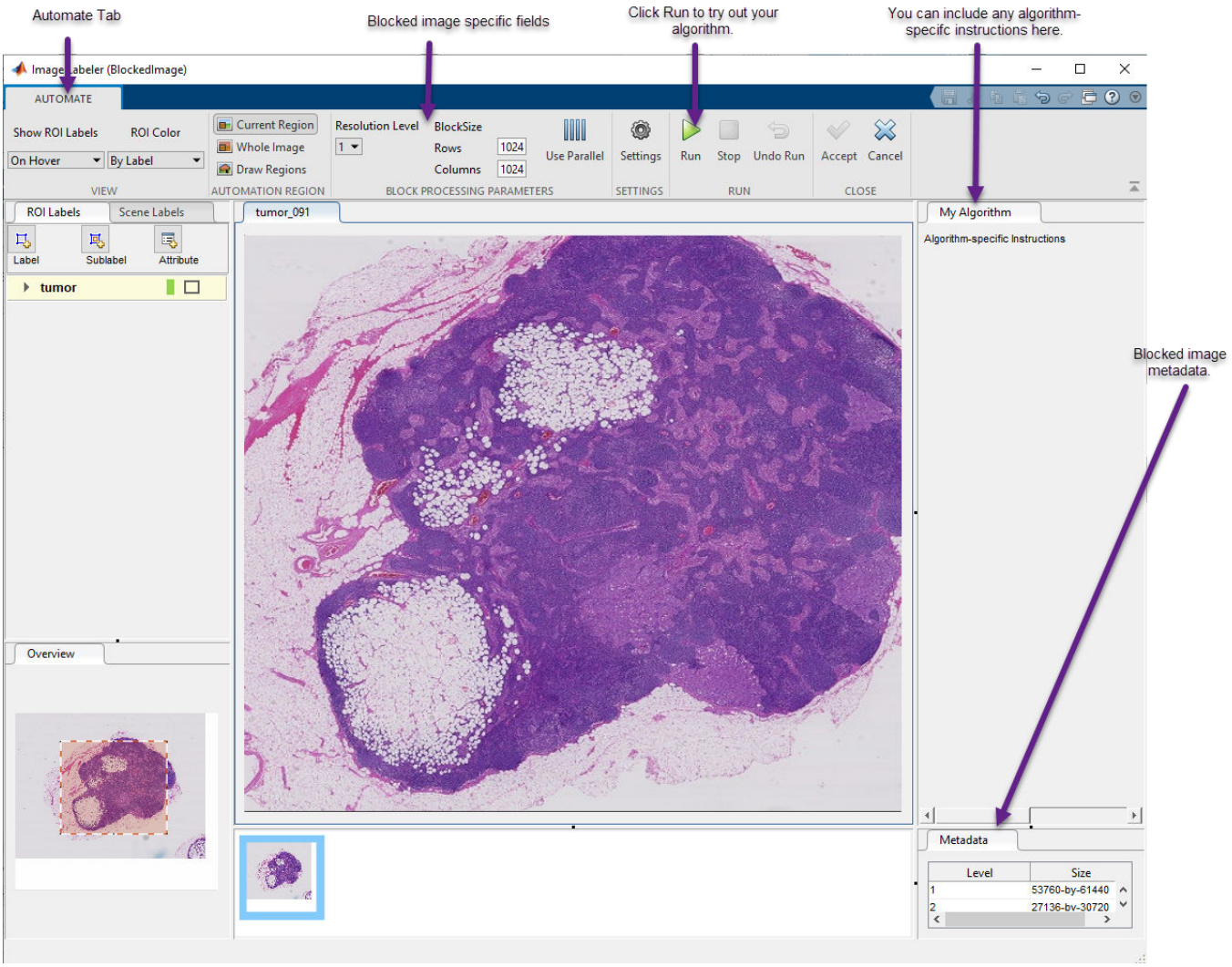
After saving the class file and adding the location folder to the path, click **Select Algorithm** in the Image Labeler app, and select **Refresh list**. You can select your new blocked image automation algorithm by clicking **Select Algorithm**, and then selecting your algorithm.



Select **My Algorithm** and click **Automate**. The app opens the **Automate** tab. Use this tab to control execution of your blocked image automation algorithm.

- Determine what parts of the image the algorithm processes: the whole image, the current region, or a region that you specify by drawing an ROI.
- Specify which resolution level you want to process. Level 1 is the finest resolution level. The resolution level with the highest number is the coarsest resolution level.
- Specify the size of the blocks in the blocked image. By default, blocks are 1024-by-1024.
- Click **Use Parallel** to take advantage of parallel processing techniques
- Click **Settings** to access other parameters made available by the algorithm

The **Automate** tab includes a **Metadata** pane that provides information about the blocked image. You can include text instructions about your algorithm in the class file, displayed in the right-most pane of the app. To try out the algorithm, click **Run**.



Postprocess Exported Labels to Create a Labeled Blocked Image

You can convert polygon coordinates exported as a `groundTruth` object from the **Image Labeler** app into a labeled blocked image using the `polyToBlockedImage` function. For more information, see “Convert Image Labeler Polygons to Labeled Blocked Image for Semantic Segmentation”.

See Also

Image Labeler | `vision.labeler.mixin.BlockedImageAutomation` | `blockedImageAutomationAlgorithm`

Related Examples

- “Get Started with the Image Labeler” on page 9-30
- “Convert Image Labeler Polygons to Labeled Blocked Image for Semantic Segmentation”

More About

- “Blocked Image Automation Algorithms” on page 9-79
- “Create Automation Algorithm for Labeling” on page 9-2

Label Pixels for Semantic Segmentation

The **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** 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:

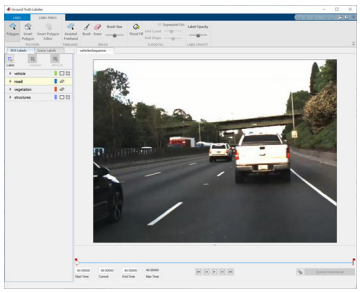
- **Image Labeler** — “Get Started with the Image Labeler” on page 9-30
- **Video Labeler** — “Get Started with the Video Labeler” on page 9-44
- **Ground Truth Labeler** — “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)

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 Labels** 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**, **Superpixel**, 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**.

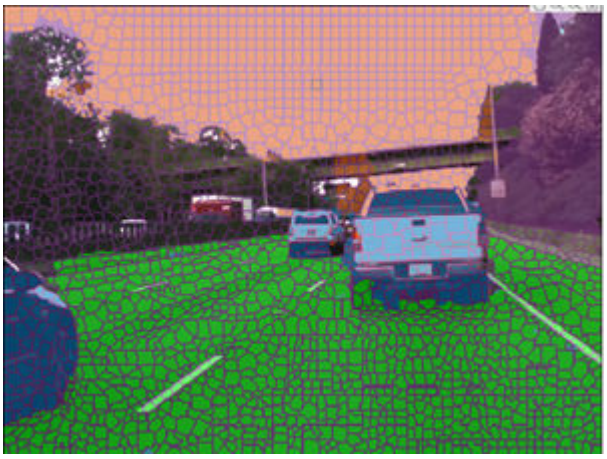
Label Pixels Using Superpixel Tool

The **Superpixel** tool labels sections of adjustable-sized grids of pixels. It enables you to quickly define objects in the scene. The **Superpixel** tool appears in the toolstrip after you create a pixel label definition.

To label pixels using **Superpixel**:

1. Select the **Superpixel** tool from the toolstrip. The app creates a super pixel grid layout over the image. You can change the grid resolution using the **Grid Count** slider.
2. Select a predefined pixel label definition and then left-click a grid to select it. The color of the grid changes to match the color of the pixel label you selected. You can select multiple grids by left-clicking each grid or by holding down the left mouse button and move the pointer continuously. Use the refining tools to modify the selection at any time during labeling.

3 Modify the **Grid Count**, while you are labeling, as needed to best capture an area. For example, to label large areas, lower the grid count. To more accurately capture the edges of an area, increase the grid count.



Label Pixels Using Smart Polygon Tool

The **Smart Polygon** tool estimates the shape of an object of interest within a polygon that you draw. The tool is useful when the shape of the object is not a simple polygon. This example uses **Smart Polygon** to label the vegetation, which has a complicated boundary with the sky.

To label pixels using **Smart Polygon**:

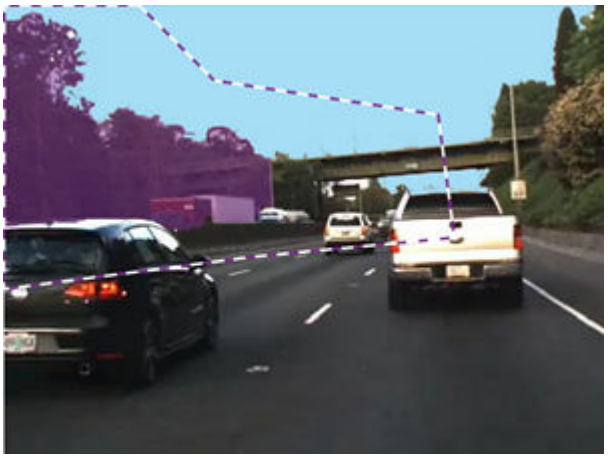
1 Select the tool and a label. The pointer changes to a crosshair +.

2 Click to add polygon vertices. Completely surround the object of interest, with some space between the object and the polygon.

3 Close the polygon by clicking the first vertex after placing the other vertices. Alternatively, you can double-click to add the last vertex and close the polygon in one step.

After you close the polygon, the tool draws an initial label.

4 Adjust the shape and position of the polygon. When the object of interest extends to the edge of the image, drag vertices to the edge of the image to ensure that the smart polygon completely encloses the object. For instance, this example shows the two leftmost vertices placed at the left edge of the image.



Smart Polygon Actions

Goal	Control
Move vertex	Click and drag the vertex.
Add vertex	• Right-click the polygon boundary at the position of the new vertex, and select **Add Point**. • Double-click the point on the boundary.
Delete vertex	Right-click the vertex and select **Delete Vertex**.
Move polygon	Click and drag any point on the polygon boundary (excluding vertices).
Delete polygon	Right-click the polygon boundary and select **Delete Polygon**.

5 Use the **Smart Polygon Editor** tools to refine the label.

- Select **Mark Foreground** to mark areas inside the region that you want to label. Foreground marks appear in green.
- Select **Mark Background** to mark areas inside the region that you do not want to label. Background marks appear in red.
- Select **Erase Marks** to remove foreground or background marks that are no longer needed.
- See Tips on page 9-21 for additional suggestions on using the **Smart Polygon** tool.

6 To finalize the label, press **Enter** or select a new label from the **ROI Labels** pane. You can no longer edit the polygon vertices or mark foreground and background regions.

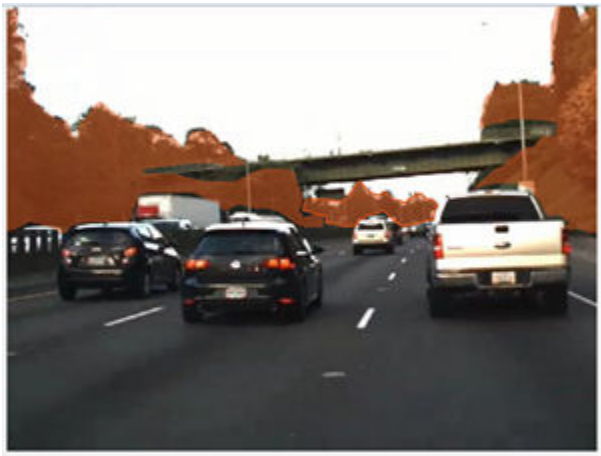
Label Pixels Using Polygon Tool

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

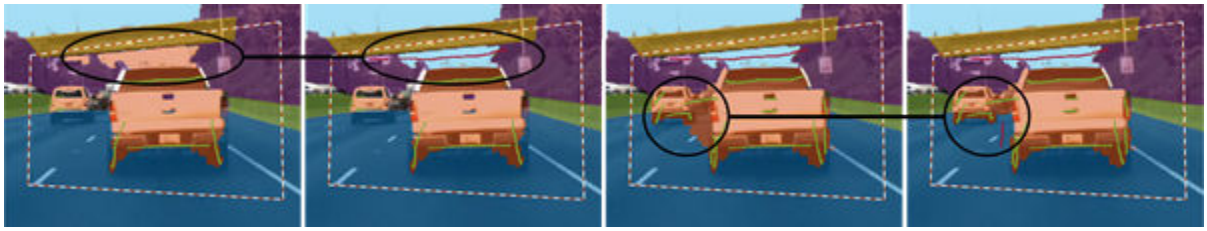
The **Assisted Freehand** tool enables you to draw an ROI that automatically follows the edge of the subject in the underlying image. You can also adjust the size and position of the ROI by using your mouse.



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.



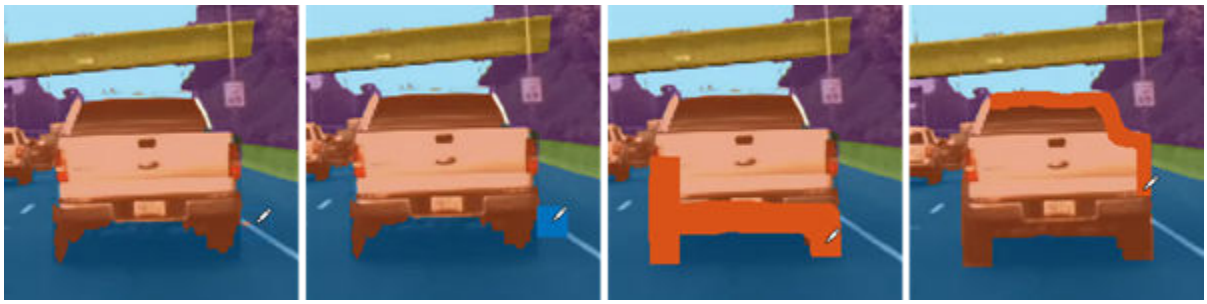
The border of the truck is jagged because **Smart Polygon** labels entire subregions, not individual pixels. The next step shows how to refine the labels along the border of the truck.

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.



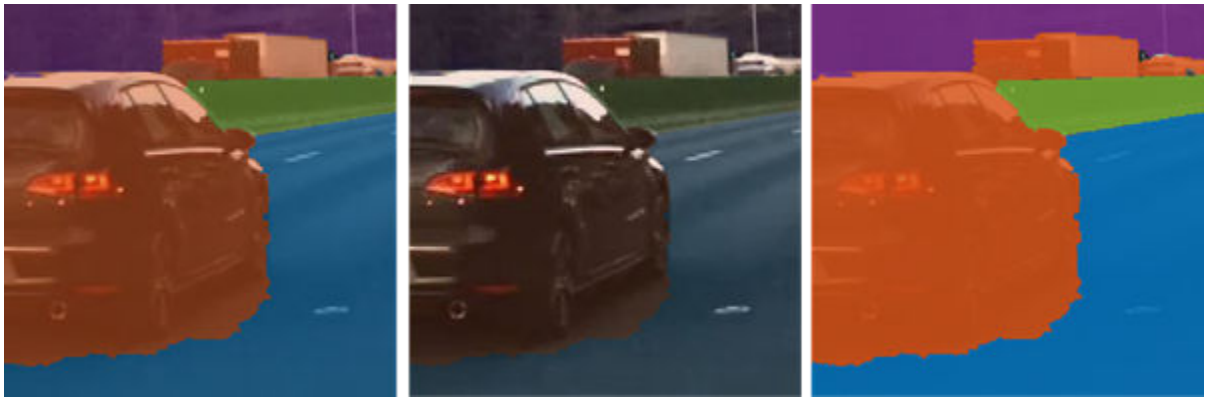
The **Erase** tool removes pixel labels when you draw over the image with the mouse.

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

Image Labeler | **Video Labeler** | **Ground Truth Labeler**

More About

- “Get Started with the Image Labeler” on page 9-30
- “Get Started with the Video Labeler” on page 9-44
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “How Labeler Apps Store Exported Pixel Labels” on page 17-27

Label Objects Using Polygons

The **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** apps enable you to label images using a variety of interactive drawing tools. Use these labels to create ground truth data for training algorithms.

About Polygon Labels

You can draw polygon shapes around objects in your image or video frame, and use the polygons to label the objects. You can also associate polygon labels with one another, enabling you to label distinct objects as instances of the same class. Exported polygon labels preserve the stack order of the polygons across label definitions and are compatible with the ground truth format required by:

- Semantic segmentation networks using an *H*-by-*W* label matrix. Using polygons maintains the order in which the labels are created or reordered (using the `Send to Back` and `Bring to Front` options). Semantic segmentation requires a pixel map of the image with labels. To do this, you must flatten the labels, (losing the layered order of the labels). Imagine turning the opacity of all the polygon labels to maximum and preserving all of the filled areas as a label matrix.
- Instance segmentation networks using an *H*-by-*W*-by-*NumObjects* mask stack.
- Raw polygons in a *NumObjects*-by-1 cell array, where each cell contains *M*-by-2 points.

In this example, you load an image, create polygon ROI labels, modify preferences related to drawing polygons, and postprocess the exported ground truth object data to use for instance segmentation and semantic segmentation training networks.

Load Unlabeled Data

You can load images stored in a datastore, from a folder, or load a previous labeler session. The images must be readable by `imread`. In this example, you are loading data that contains a number of vehicles, including boats, a tanker, and a plane.

Load Data Programmatically

Use this process to programmatically load images from a datastore:

1. Create a datastore from a folder of images. For example, if you had a folder named "stopSignImages" that contained images, you could create a datastore of the images with this code.

   ```
   imageFolder = fullfile(toolboxdir('vision'),'visiondata','stopSignImages');
   imds = imageDatastore(imageFolder);
   ```

2. Load the datastore into the **Image Labeler**:

   ```
   imageLabeler(imds);
   ```

Alternatively, you can load the folder of images directly into **Image Labeler**:

```
imageFolder = fullfile(toolboxdir('vision'),'visiondata','stopSignImages');
imageLabeler(imageFolder);
```

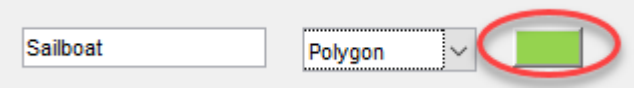
Load Data Directly from the App

Click **Import** to load images or click **Open Session** to open a saved session.

Create Polygon Labels

To follow these steps that use an image of sailboats, a tanker, and an airplane, open the **Image Labeler** app and then import the "`boats.png`" image from the `vision/visiondata` folder.

1. In the **ROI Labels** pane on the left, click **Label**.
2. Select the `Polygon` label type and name it `Sailboat`.
3. Optionally, you can change the label color by clicking the preview color.



4. Click **OK**.

   The **Sailboat** label appears in the **ROI Labels** pane.
5. Repeat steps 1 through 4 to create a **Tanker** label and a **Airplane** label. You can move a label in the list by left-clicking and dragging the label up or down.

Draw Polygon ROI Labels

1. Select the **Sailboat** label, and then using the mouse, draw a polygon ROI around each of the sailboats.
2. Select the **Tanker** label and draw a polygon ROI around the tanker ship.
3. Select the **Airplane** label and draw a polygon ROI around the airplane.

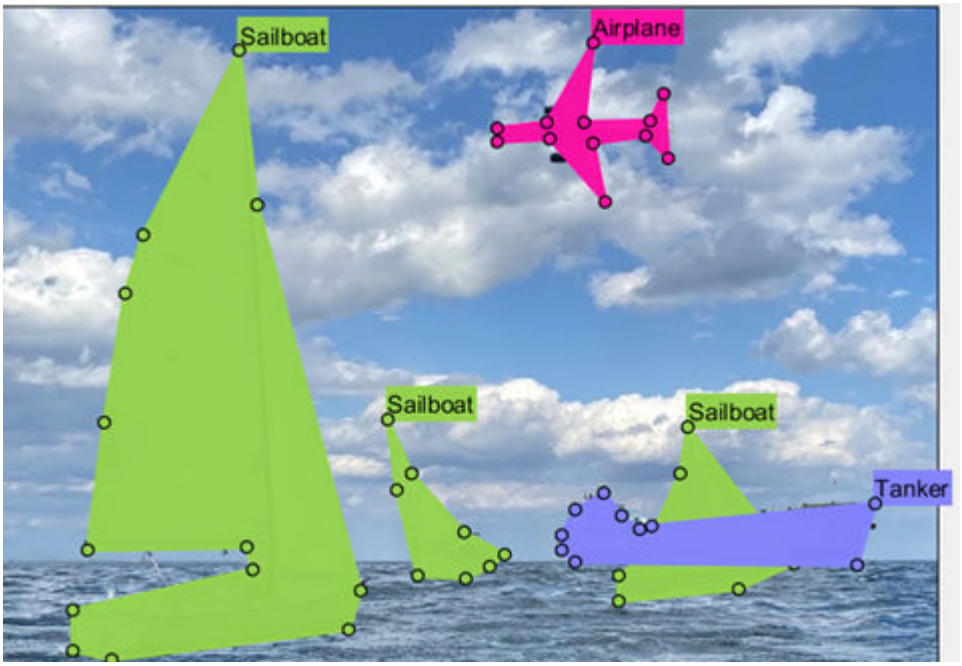
Modify Polygon Preferences and Stacking Order

There are a number of ways to modify the appearance of the ROI labels. You can also set the stacking order for the exported ground truth file. The stacking order indicates whether an object is in front of

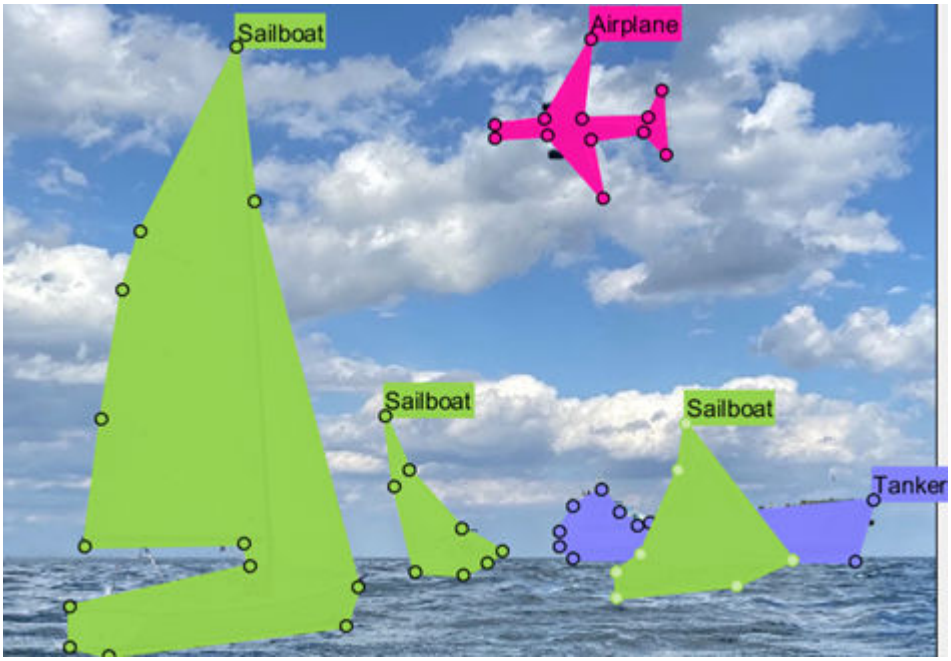
or behind another object, and is required in order to resolve overlapping areas to build a semantic label map.

In the next steps, you modify the color of a label, set the appearance of label names while labeling, modify the opacity of labels, view instances of a class of labels, and set the stacking order.

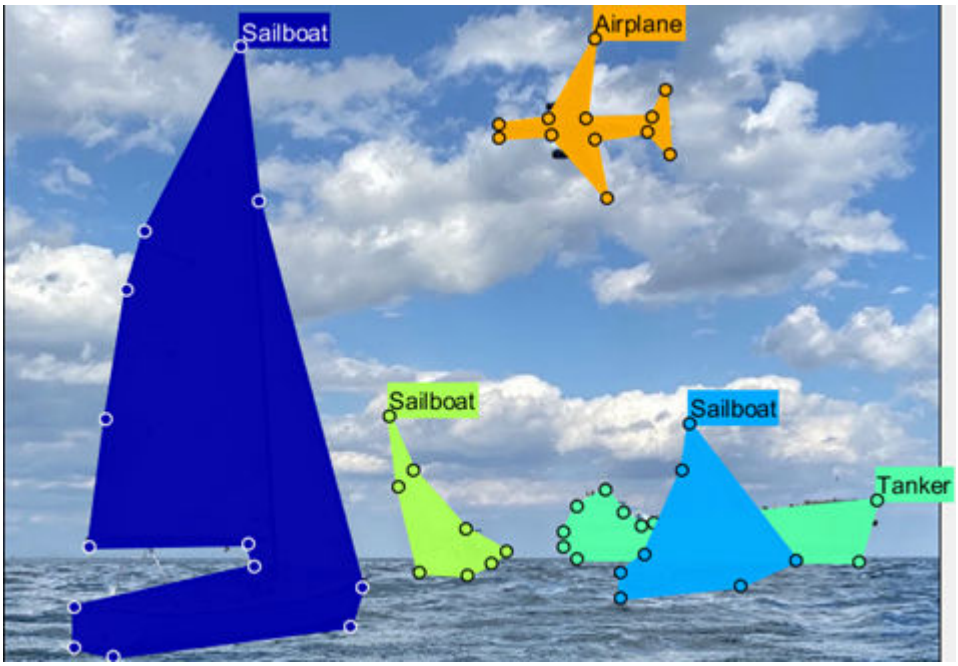
1. To modify the color of the airplane ROI label, right-click the `Airplane` label in the **ROI Labels** pane and select **Edit Label**. Click the color preview and select a color.
2. To show ROI label names while labeling, select **Always** from the **Show ROI Labels** menu in the **View** section.
3. To increase the label opacity, use the **Polygon** slider in the **Label Opacity** section. Increasing the opacity helps to recognize which labels are in front of or behind other labels in the scene.



4. The tanker label is in front of the sailboat label, but it should be behind it. The position affects the stacking order in the exported ground truth. To modify the stacking order, right-click on the tanker ROI label in the scene and select **Send To Back**.



5 To view instances of ROI labels, select `By Instance` from the **ROI Color** drop-down menu in the **View** section.



6 Export the labeled ground truth data by clicking **Export** and selecting **To Workspace**. Name the ground truth data MAT file `gTruth`.

Postprocess Exported Labels for Instance or Semantic Segmentation Networks

You can use the exported, labeled ground truth for training an instance segmentation network or a semantic segmentation network.

Follow these steps to process the polygon data for either semantic segmentation or instance segmentation.

Step	Description	Procedure

1 — Display ground truth data	The exported ground truth object contains the data for five objects and three definitions.	Enter `gTruth` on the MATLAB command-line.
		`>> gTruth`
		`gTruth =`
		`  groundTruth with properties:`
		`DataSource: [1×1 groundTruthDataSource]`
		`LabelDefinitions: [3×5 table]`
		`LabelData: [1×3 table]`

2 — Get polygon data	The LabelData property groups the data by label name. The `Sailboat` label cell array contains three elements, the `Tanker` cell array contains one element, and the `Airplane` cell array contains one element.	Type `gTruth.LabelData`.
		`>> gTruth.LabelData`
		`ans =`
		`  1×3 table`
		`Sailboat  Tanker  Airplane`
		`{3×1 cell}  {1×1 cell}  {1×1 cell}`

3 — Stack ground truth polygon data	The sailboat and the tanker are overlapping. This information (relative ordering of pixels) is lost in this format.
	Use the `gatherLabelData` object function to group the data by label type. This produces one table containing five objects stacked.
		Use the `gatherLabelData` property and store the output.
		`>> out = gatherLabelData(gTruth,[labelType.Polygon],'GroupLa`
		`out =`
		`  1×1 cell array`
		`{1×1 table}`
		Show the contents of the table.
		`>> out{1}.PolygonData`
		`ans =`
		`  1×1 cell array`
		`{5×2 cell}`

Step	Description	Procedure

4 — View ground truth data by depth order	The contents of `PolygonData`, shows the order of the stacked polygons. The order is consistent with the labeling order, including the tanker, which you sent to the back. Now that the labels are flattened, at the base is the tanker, and the sailboat above it overwrites the tanker where they overlap.	Show the polygon data.
		`>> out{1}.PolygonData{1}`
		`ans =`
		`5×2 cell array`
		`{12×2 double} {'Airplane'}`
		`{ 6×2 double} {'Sailboat'}`
		`{ 7×2 double} {'Sailboat'}`
		`{13×2 double} {'Sailboat'}`
		`{ 9×2 double} {'Tanker' }`

Convert Polygon Data for Semantic Segmentation

Step	Description	Procedure

1 — Create a label ID map	Create a map associating class names to class IDs.	`classNames = {'Airplane','Sailboat','Ta`
		`classIds = 1:numel(classNames);`
		`labelIDMap = containers.Map(classNames,`

2 — Extract polygons	Extract the polygon coordinates and polygon labels from the output structure.	`polygons = out{1}.PolygonData{1}(:,1);` `polygonLabels = out{1}.PolygonData{1}(:`

3 — Flatten polygons to a semantic segmentation map	Convert the label names to IDs.
	Convert the polygon coordinates to a semantic segmentation map.
		`polygonLabelIDs = cellfun(@(x)labelIDMa`
		`imageSize = [645 916]; % size(boats_im)`
		`outputIm = poly2label(polygons,polygonL`

Convert Polygon Data for Instance Segmentation

Step	Description	Procedure

Preallocate a mask stack for instance segmentation	Preallocate the mask stack with height and width equal to the image dimensions, and the channel depth the same as the number of polygons.	`polygons = out{1}.PolygonData{1}(:,1);`
		`numPolygons = size(polygons,1);`
		`imageSize = [645 916]; % size(boats_im)`
		`maskStack = false([imageSize(1:2) numPo`

Convert polygons to instance masks	Convert each polygon into a separate mask and insert it into the mask stack.	`for i = 1:numPolygons`
		`maskStack(:,:,i) = poly2mask(polygo`
		`polygons{i}(:,2)`
		`end`

See Also

Apps
Image Labeler | Video Labeler | Ground Truth Labeler

Functions
`gatherLabelData`

Objects
`groundTruth` | `groundTruthMultisignal`

More About

- “Get Started with the Image Labeler” on page 9-30
- “Get Started with the Video Labeler” on page 9-44
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Label Pixels for Semantic Segmentation” on page 9-14

Get Started with the Image Labeler

The **Image Labeler** app provides an easy way to interactively create a variety of shapes to mark as region of interest (ROI) labels. You can create rectangular, polyline, pixel, and polygon ROI labels and scene labels in an image or image sequence.

You can use labeled data to validate or train algorithms such as image classifiers, object detectors, and semantic and instance segmentation networks. Consider your application when choosing a labeling drawing tool to create ROI labels. For more details on how to select the right label type and drawing tool for your application, see “ROI Labels, Sublabels, and Attributes”.

This example gets you started using the app by showing you how to:

- Load unlabeled data
- 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, pixel, or polygon 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

You can load images stored in a datastore, from a folder, or load a previous labeler session. The images must be readable by `imread`.

Load Data Programmatically

To programmatically load images from a datastore:

1. Create a datastore from a folder of images:

   ```
   imageFolder = fullfile(toolboxdir('vision'),'visiondata','bookcovers')
   imds = imageDatastore(imageFolder)
   ```

2. Load the datastore:

   ```
   imageLabeler(imds)
   ```

To programmatically load a folder of images:

```
imageFolder = fullfile(toolboxdir('vision'),'visiondata','bookcovers')
imageLabeler(imageFolder)
```

Load Data Directly from the App

In this example, you launch and load an image interactively from the app. Open the app from the **Apps** tab, under **Image Processing and Computer Vision**. Click **Import** to load images or click **Open Session** to open a saved session.

To load the image used for this example, click **Import** and then select **From File**. Select the `boats.png` image:

```
matlab/toolbox/vision/visiondata/boats.png
```

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 Label

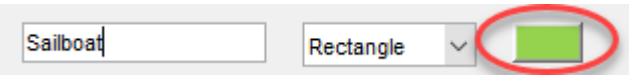
An ROI label is a label that corresponds to a region of interest (ROI) in an image. You can define these ROI label types.

- `Rectangle` — Draw 2-D rectangular bounding box labels around objects in an image, such as vehicles, boats, buildings.
- `Projected cuboid` — Draw 3-D bounding box labels around objects in an image, such as vehicles, boats, buildings.
- `Line` — Draw linear ROIs to label lines, such as lane boundaries.
- `Pixel label` — Draw pixels to label various classes, such as road or sky, for semantic segmentation. For more information about pixel labeling, see “Label Pixels for Semantic Segmentation” on page 9-14.
- `Polygon` — Draw polygon labels around objects. You can label distinct instances of the same class. For more information on drawing polygon ROI labels for instance and semantic segmentation networks, see Label Objects Using Polygons on page 9-23

For more details about these ROI label definitions, see “ROI Labels, Sublabels, and Attributes”.

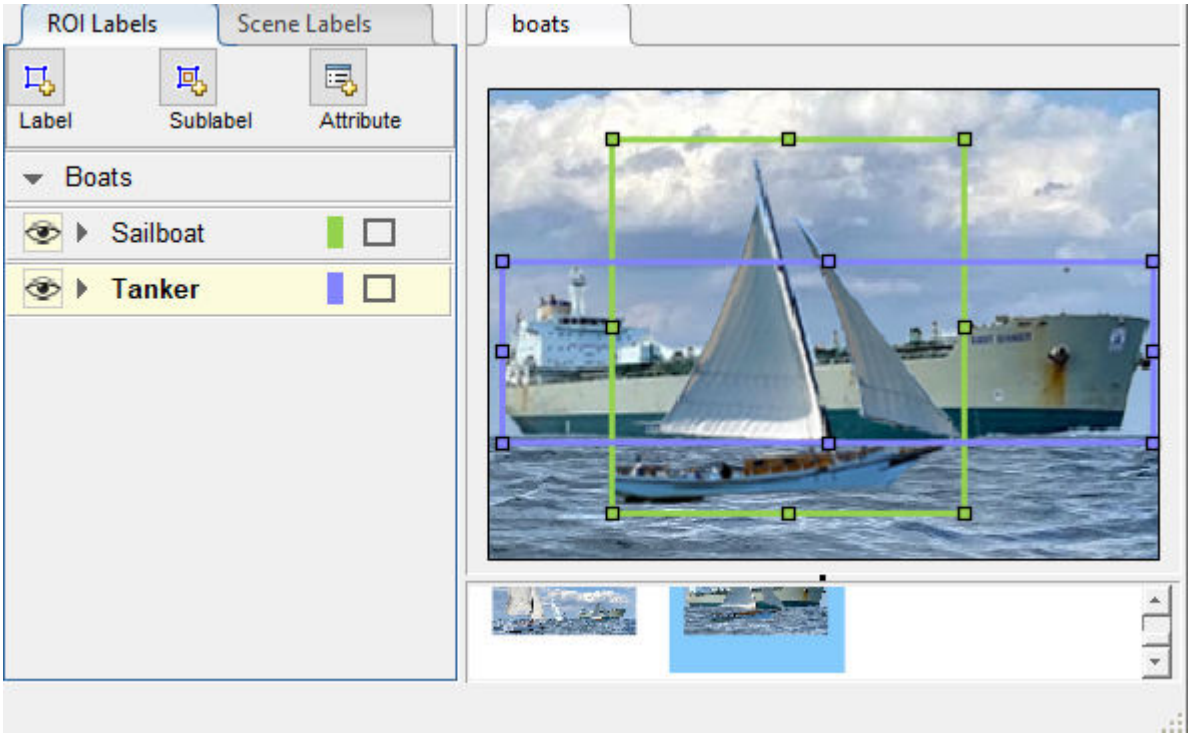
In this example, you define a `Boat` group for labeling types of boats, and then create a `Rectangle` ROI label for a `Sailboat` and a `Tanker`. To control showing the ROI label names during labeling, select **On Hover**, **Always**, or **Never** from the **Show ROI Labels** drop-down menu.

1. In the **ROI Labels** pane on the left, click **Label**.
2. Create a `Rectangle` label type named `Sailboat`.
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 `Boats`
5. Click **OK**.

   The **Boats** group name appears in the **ROI Labels** pane with the label **Sailboat** 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. To add a second type of **Boats** label, select the group **Boats**, then click **Label**. Name the label **Tanker**. Click **OK**.
7. Select the **Sailboat** sublabel, then use the mouse to draw a rectangular ROI around the a sailboat. Select the **Tanker** sublabel to draw an ROI around the tanker ship.



Create Sublabels

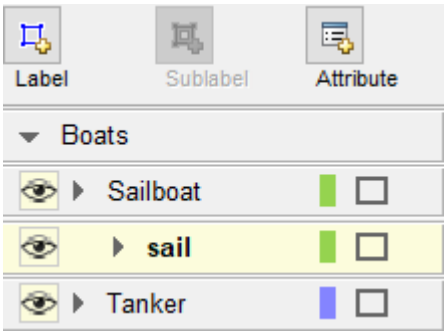
A *sublabel* is a type of ROI label that corresponds to a parent ROI label. Each sublabel must belong to, or be a child of, a specific label defined in the **ROI Labels** pane. For example, in the ocean scene, a sailboat label might have sublabels for sails. For more details about sublabels, see “ROI Labels, Sublabels, and Attributes”.

Define a sublabel for sails.

1. In the **ROI Labels** pane on the left, click the **Sailboat** label.
2. Click **Sublabel**.
3. Create a `Rectangle` sublabel named `sail` and optionally write a description. Click **OK**.

   The **sail** sublabel appears in the **ROI Labels** pane. The sublabel is nested under the selected ROI label, **Sailboat**, and by default has the same color as its parent label.

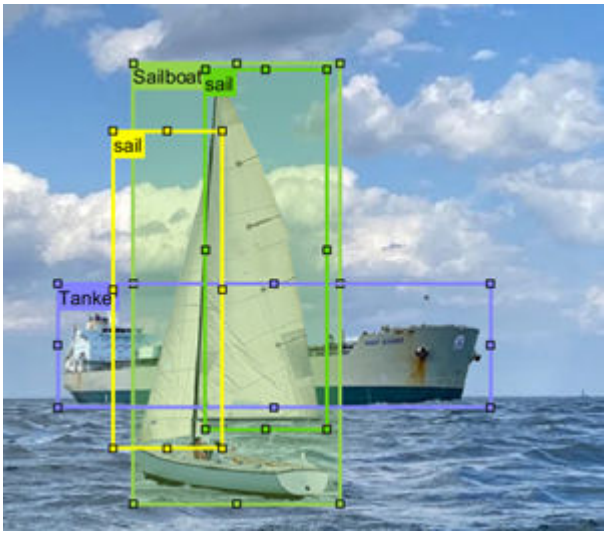
   You can add multiple sublabels under a label. You can also drag-and-drop the sublabels to reorder them in the list and right-click any label for additional edits.



4 In the **ROI Labels** pane, select the **sail** sublabel.

5 In the image frame, select the **Sailboat** label. The label turns yellow when selected. You must select the **Sailboat** label (parent ROI) before you can draw a sublabel for it.

Draw **sail** sublabels for each of the sails.

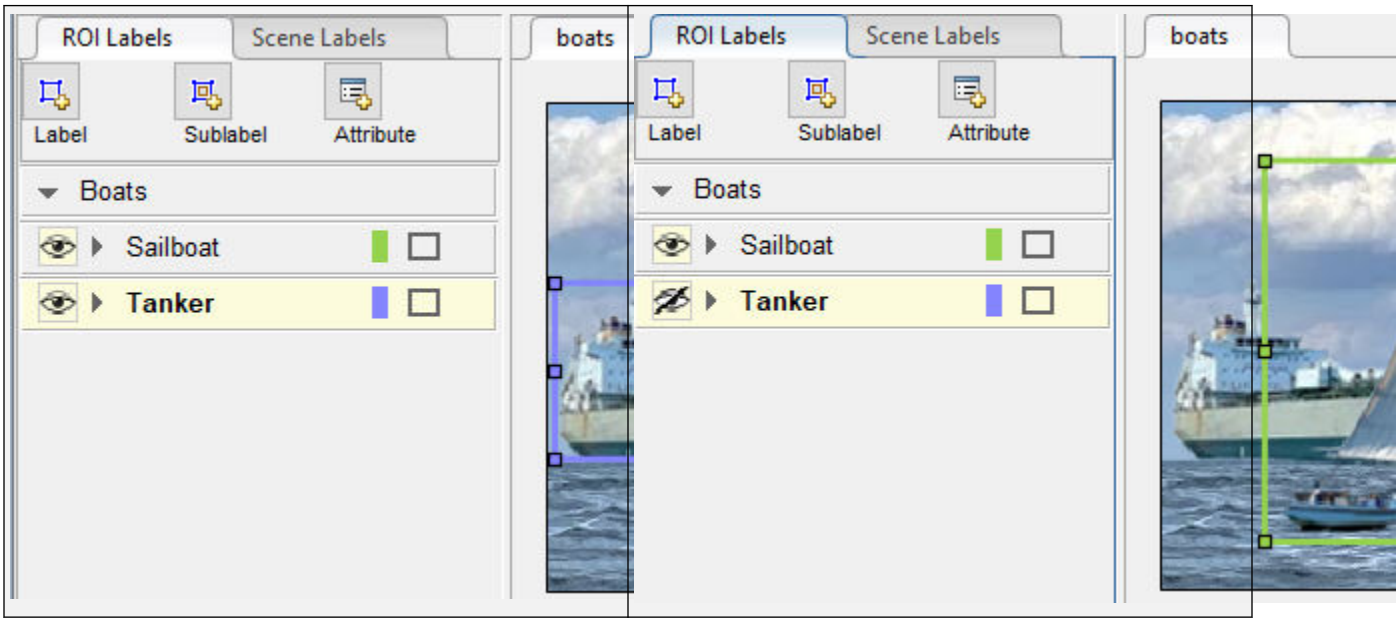


Sublabels cannot have their own sublabels and they can only be used with rectangular or polyline ROI labels.

Show or Hide Labels and Sublabels

You can show or hide the labels or sublabels in a labeled image by using the  icon on the **ROI Labels** pane. The  appears only after you define a label or sublabel. By default, the app displays all the labels and the sublabels.

To hide a label or sublabel, click on the  icon along side the label or sublabel name. The app hides the corresponding label or sublabel and displays the  icon.



Create ROI Attribute

An ROI attribute specifies additional information about an ROI label or sublabel. For example, in a driving scene, attributes might include the type or color of a vehicle. In the ocean scene, attributes might include the type of sailboat or number of sails. You can define ROI attributes of these types.

- `Numeric Value` — Specify a numeric scalar attribute, such as the number of doors on a labeled vehicle or the number of sails on a sailboat.
- `String` — Specify a string scalar attribute, such as the color of a vehicle or boat.
- `Logical` — Specify a logical true or false attribute, such as whether a vehicle is in motion or a boat is making way.
- `List` — Specify a drop-down list attribute of predefined strings, such as make or model of a vehicle or boat.

For more details about these attribute types, see “ROI Labels, Sublabels, and Attributes”.

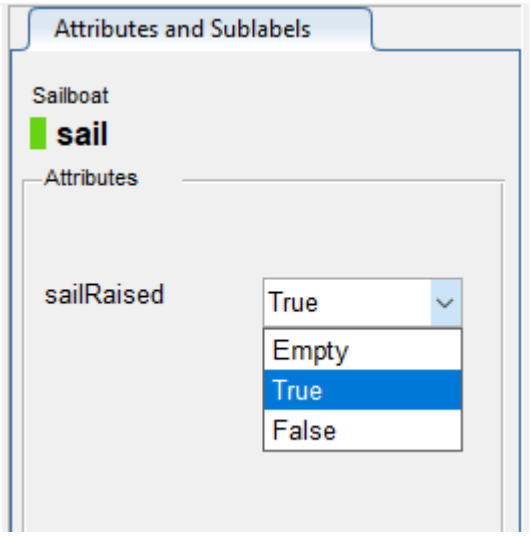
Add an attribute to a Label.

1. In the **ROI Labels** pane on the left, select the **Sailboat** label and click **Attribute**.
2. In the **Attribute Name** box, type `sailboatType`. Set the attribute type to `List`.
3. In the **List Items** section, type different types of sailboats, such as `Monohull`, `Catamaran`, and `Multihull`, each on its own line. Optionally give the attribute a description, and click **OK**. You can hover the information icon that appears next to the attribute field to display the added description.
4. Select a **Sailboat** ROI label. In the **Attributes and Sublabels** pane, select the appropriate **sailboatType** attribute value for that vehicle.

You can also add attributes to sublabels. Add an attribute for the **sail** sublabel to indicate whether it is a foresail or a mainsail.

1. In the **ROI Labels** pane on the left, select the **sail** sublabel and click **Attribute**.

2. In the **Attribute Name** box, type `sailRaised`. Set the attribute type to `Logical`. Leave the **Default Value** set to `Empty`, optionally write a description, and click **OK**.
3. Select a sail in the scene. In the right-hand pane, set the appropriate **isOn** attribute value, or leave the attribute value set to `Empty`.



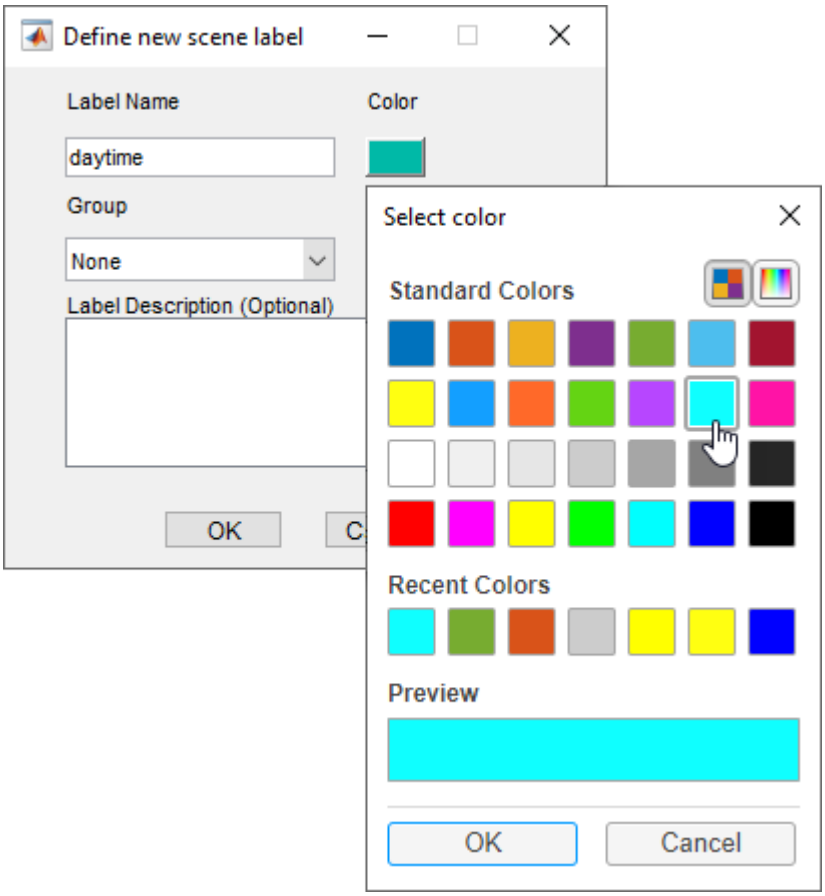
To delete an attribute, right-click an ROI label or sublabel, and select the attribute to delete. Deleting the attribute removes attribute information from all previously created ROI label annotations.

Create Scene Labels

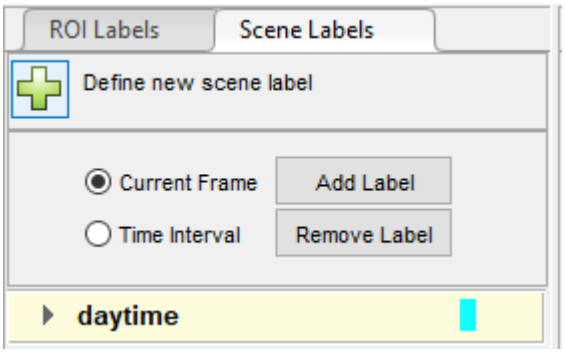
A scene label defines additional information for the entire scene. Use scene labels to describe conditions, such as lighting and weather, or events, such as lane changes or point of sail.

Create a scene label to apply to an image.

1. In the left pane of the app, select the **Scene Labels** next to the **ROI Labels** tab.
2. Click **Define new scene label**, and in the **Label Name** box, enter a scene label named `daytime`.
3. Change the color of the label definition to light blue to reflect the nature of the scene label. Under the **Color** parameter, click the color preview and select the standard light blue colors. Then, click **OK** to close the color selection window.



4 Leave the **Group** parameter set to the default of `None` and click **OK**. The **Scene Labels** pane shows the scene label definition.



5 Click **Apply to Image** to apply the `daytime` label to the scene. A checkmark appears for the scene label.

6 To edit or delete a scene label, right-click on the label and select either **Edit Label** or **Delete Label**.

Label Ground Truth

You can label images manually, use a suitable built-in automation algorithm, create a new algorithm, or import an algorithm.

Label Ground Truth Manually

The ROI and Scene labels are defined for your entire session and all images, but you must draw the labels and sublabels for each image and update the attribute information.

Label Ground Truth Using an Automation Algorithm

To speed up the labeling process, you can use an automation algorithm to label the remainder of your images. Select one of these types of automation algorithms from the app toolstrip **Automate Labeling > Select Algorithm** section.

- Use one of the built-in automation algorithms — Select a suitable algorithm. Follow the steps that appear in the right pane.
- **Add Whole Image Algorithm** — You can create a new automation algorithm or import one. For details on both options, see “Create Automation Algorithm for Labeling” on page 9-2.
- **Add Blocked Image Algorithm** — You can create a new blocked image automation algorithm or import one. For details on both options, see “Label Large Images in the Image Labeler” on page 9-6.

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 toolstrip, 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 9-73. This summary does not show 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 9-84.

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 17-27.

In this example, you export the labeled ground truth to the MATLAB workspace. From the app toolstrip, 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: [1×1 groundTruthDataSource]
    LabelDefinitions: [2×6 table]
           LabelData: [531×3 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: [531×1 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 =
  3×6 table

      Name         Type        LabelColor        Group       Description     Hierarchy
    _________    _________    _____________    ___________    ___________    _____________

    {'Car'  }    Rectangle    {1×3 double}    {'Vehicle'}    {0×0 char}     {1×1 struct}
    {'Truck'}    Rectangle    {1×3 double}    {'Vehicle'}    {0×0 char}     {0×0 double}
    {'Sunny'}    Scene        {1×3 double}    {'Weather'}    {0×0 char}     {0×0 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: [1×1 struct]
          color: [1×1 struct]
       inMotion: [1×1 struct]
        carType: [1×1 struct]
      headlight: [1×1 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: [1×1 struct]
```

Display information about the `carType` attribute.

```
gTruth.LabelDefinitions.Hierarchy{1}.carType
```

```
ans =

  struct with fields:

      ListItems: {3×1 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 9-67.

See Also

Apps
Image Labeler

Objects
`groundTruth` | `imageDatastore` | `groundTruthDataSource` | `labelDefinitionCreator` | `vision.labeler.AutomationAlgorithm`

More About

- “Training Data for Object Detection and Semantic Segmentation” on page 9-84
- “Keyboard Shortcuts and Mouse Actions for Image Labeler” on page 9-63
- “Use Sublabels and Attributes to Label Ground Truth Data” on page 9-80
- “Label Pixels for Semantic Segmentation” on page 9-14
- “Label Objects Using Polygons” on page 9-23
- “Create Automation Algorithm for Labeling” on page 9-2

Choose an App to Label Ground Truth Data

You can use Computer Vision Toolbox, Automated Driving Toolbox™, Lidar Toolbox™, Audio 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, instance 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. The **Image Labeler** can also handle very large images (at least one dimension >8K).
- 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. You can also use **Signal Labeler** to perform audio-specific tasks, such as speech detection and speech-to-text transcription.

This table summarizes the key features of the labeling apps.

Labeling App	Data Sources	Label Support	Automation	Additional Features
Image Labeler	• Image collections • Very large images (at least one dimension >8K)	• Rectangle regions of interest (ROIs) • Projected cuboid (ROIs) • Line ROIs • Pixel ROIs • Polygon ROIs • Sublabels • Attributes • Scenes	• Built-in automation algorithms • Custom automation algorithms • Blocked image automation algorithms	• View visual summary of labeled data

Labeling App	Data Sources	Label Support	Automation	Additional Features
Video Labeler	• Videos • Image sequences • Custom image data sources	• Rectangle ROIs • Projected cuboid (ROIs) • Line ROIs • Pixel ROIs • Polygon ROIs • Sublabels • Attributes • Scenes	• Built-in automation algorithms • Custom automation algorithms • Temporal automation algorithms	• View visual summary of labeled data
Ground Truth Labeler	• Videos • Image sequences • Custom image data sources • Point cloud sequences (PCD or PLY files) • Velodyne® lidar files • Rosbags (requires ROS Toolbox)	• Rectangle ROIs • Projected cuboid (ROIs) • Cuboid ROIs • Line ROIs • Pixel ROIs • Polygon ROIs • Sublabels • Attributes • Scenes	• Built-in automation algorithms, including vehicle and lane detection algorithms and a point cloud temporal interpolation algorithm • Custom automation algorithms • Temporal automation algorithms • Multisignal automation	• 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

Labeling App	Data Sources	Label Support	Automation	Additional Features
Lidar Labeler	• Point cloud sequences (PCD or PLY files) • Velodyne lidar files • LAS/LAZ file sequences • Rosbags (requires ROS Toolbox)	• Cuboid ROIs • Attributes • Scenes	• Built-in automation algorithms, including a lidar object tracker and point cloud temporal interpolator • Custom automation algorithms • Temporal automation algorithms	• 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
Signal Labeler	• Numeric arrays, MATLAB timetables, and `labeledSignalSet` objects in the MATLAB workspace • MAT-files and CSV files • Audio files (WAVE, OGG, FLAC, AU, AIFF, AIFC, MP3, MPEG-4 AAC)	• Time-based ROIs • Time-based points • Attributes • File-level labels • Sublabels	• Built-in peak labeling • Custom automation algorithms • Speech detection • Speech-to-text transcription (requires Audio Toolbox extended functionality for speech2text)	• 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 playback • Inspect audio file information

See Also

More About

- “Get Started with the Image Labeler” on page 9-30
- “Get Started with the Video Labeler” on page 9-44
- “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 Spoken Words in Audio Signals” (Signal Processing 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.

You can use labeled data to validate or train algorithms such as image classifiers, object detectors, and semantic and instance segmentation networks. Consider your application when choosing a labeling drawing tool to create ROI labels. For more details on how to select the right label type and drawing tool for your application, see “ROI Labels, Sublabels, and Attributes”.

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.

ROI and Scene Label Definitions

- An ROI label corresponds to either a rectangular, polyline, pixel, or polygon 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

Programmatically open the app and load a video. Videos must be in a file format readable by `VideoReader`.

```
videoLabeler('visiontraffic.avi')
```

You can also load a video from the app. Open the app from the **Apps** tab, under **Image Processing and Computer Vision**. Click **Import** to load a video or image sequence, or click **Open Session** to load a saved session.

Explore the video. Click the Play button  to play the entire video, or use the slider  to navigate between frames.

To load an image sequence with corresponding timestamps, select **Import** > **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 9-57.

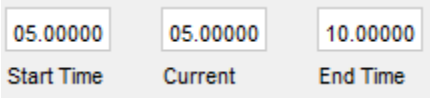
Create Label Definitions

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`.

Set Time Interval

You can label the entire video or start with a portion of the video. In this example, you label a five-second time range 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 range.
3. In the **End Time** box, type `10`.



Optionally, to make adjustments to the time range, click and drag the red interval flags.

The entire app is now set up to focus on this specific time range. 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 range to fill the entire playback section, click **Zoom in Time Range**.

Create ROI Label

An ROI label is a label that corresponds to a region of interest (ROI) in an image. You can define these ROI label types.

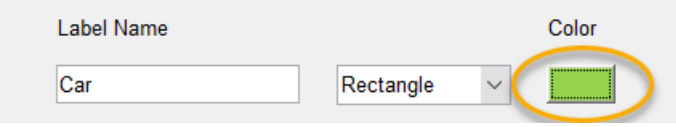
- `Rectangle` — Draw 2-D rectangular bounding box labels around objects in an image, such as vehicles, boats, buildings.
- `Projected cuboid` — Draw 3-D bounding box labels around objects in an image, such as vehicles, boats, buildings.
- `Line` — Draw linear ROIs to label lines, such as lane boundaries.
- `Pixel label` — Draw pixels to label various classes, such as road or sky, for semantic segmentation. For more information about pixel labeling, see “Label Pixels for Semantic Segmentation” on page 9-14.
- `Polygon` — Draw polygon labels around objects. You can label distinct instances of the same class. For more information on drawing polygon ROI labels for instance and semantic segmentation networks, see “Label Objects Using Polygons” on page 9-23

For more details about these ROI label definitions, see “ROI Labels, Sublabels, and Attributes”.

In this example, you define a `vehicle` group for labeling types of vehicles, and then create two `Rectangle` ROI labels, `Car` and `Truck`. After creating labels, 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.



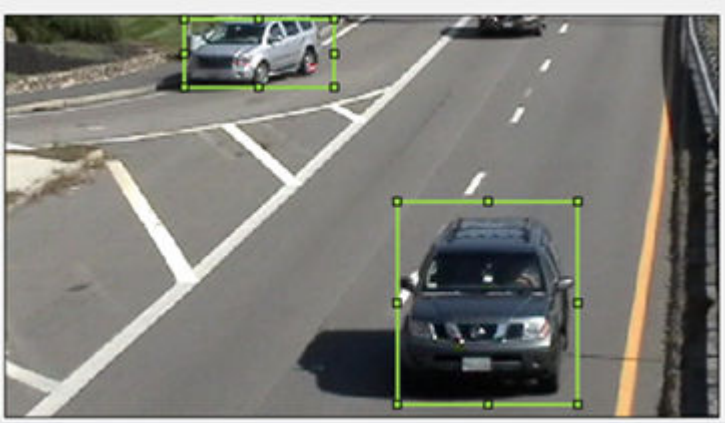
4 From the `Group` drop-down menu, select `New Group` and name the group `Vehicle`

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 To add the 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. For more details about sublabels, see “ROI Labels, Sublabels, and Attributes”.

Define a sublabel for headlights.

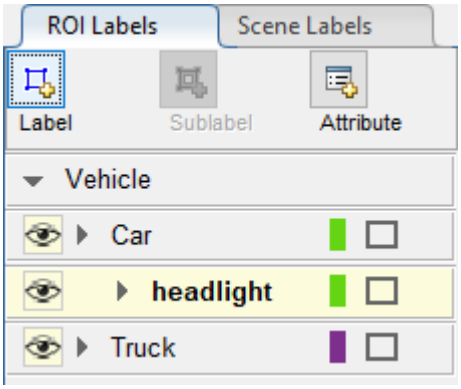
1 In the **ROI Labels** pane on the left, click the **Car** label.

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.

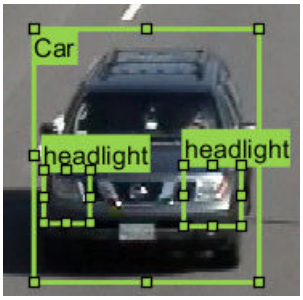


4 In the **ROI Labels** pane, select the **headlight** sublabel.

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.

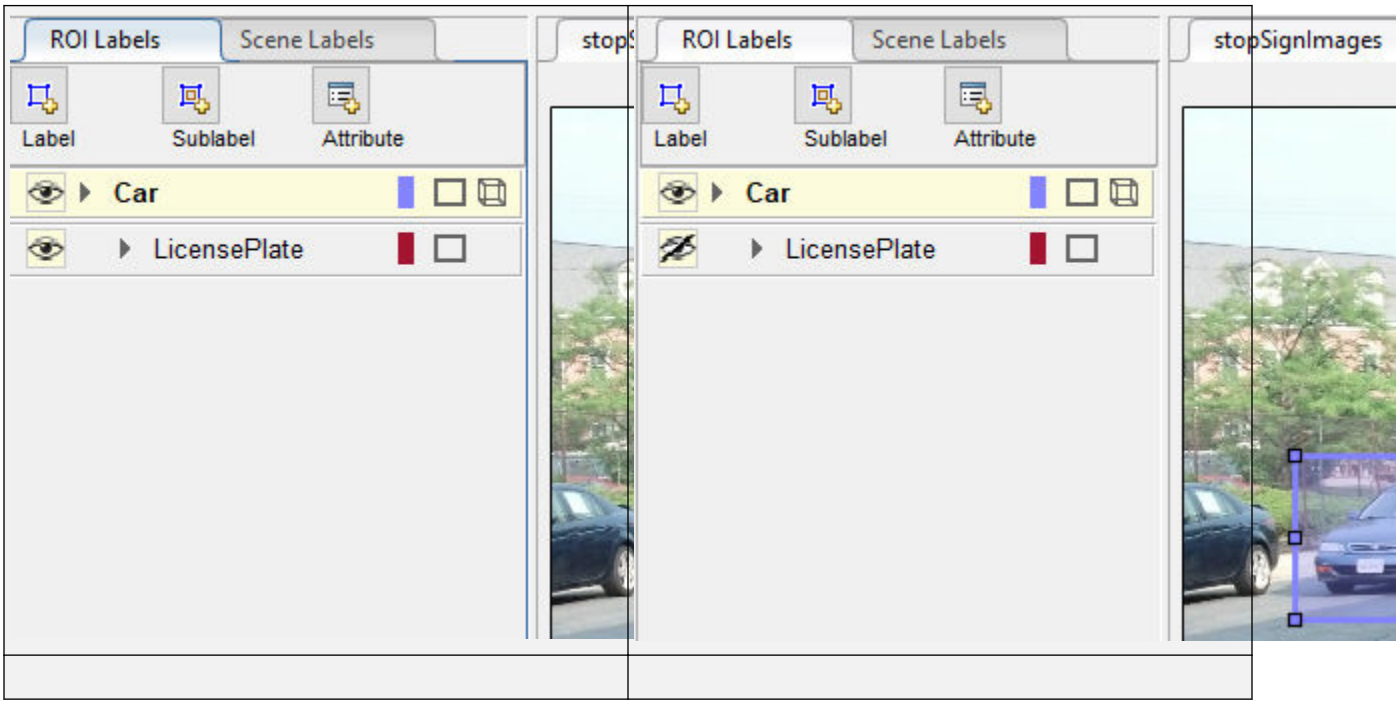


Sublabels can only be used with rectangular or polyline ROI labels and cannot have their own sublabels. For more details on working with sublabels, see “Use Sublabels and Attributes to Label Ground Truth Data” on page 9-80.

Show or Hide Labels and Sublabels

You can show or hide the labels or sublabels in a labeled video by using the  icon on the **ROI Labels** pane. The  appears only after you define a label or sublabel. By default, the app displays all the labels and the sublabels.

To hide a label or sublabel, click on the  icon along side the label or sublabel name. The app hides the corresponding label or sublabel and displays the  icon.



Create Attributes

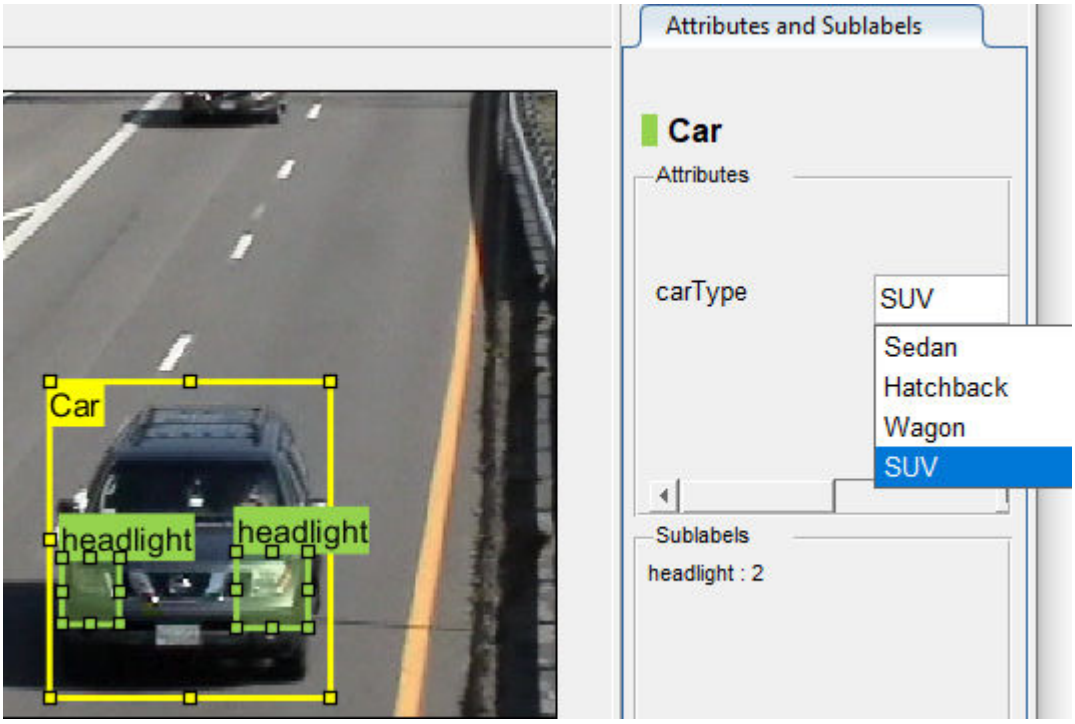
An ROI attribute specifies additional information about an ROI label or sublabel. For example, in a driving scene, attributes might include the type or color of a vehicle. In the ocean scene, attributes might include the type of sailboat or number of sails. You can define ROI attributes of these types.

- `Numeric Value` — Specify a numeric scalar attribute, such as the number of doors on a labeled vehicle or the number of sails on a sailboat.
- `String` — Specify a string scalar attribute, such as the color of a vehicle or boat.
- `Logical` — Specify a logical true or false attribute, such as whether a vehicle is in motion or a boat is making way.
- `List` — Specify a drop-down list attribute of predefined strings, such as make or model of a vehicle or boat.

For more details about these attribute types, see “ROI Labels, Sublabels, and Attributes”.

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`, `SUV`, and `Wagon`, each on its own line. Optionally give the attribute a description, and click **OK**. You can hover the information icon that appears next to the attribute field to display the added description.
4. Select a **Car** ROI label in the image frame. 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.

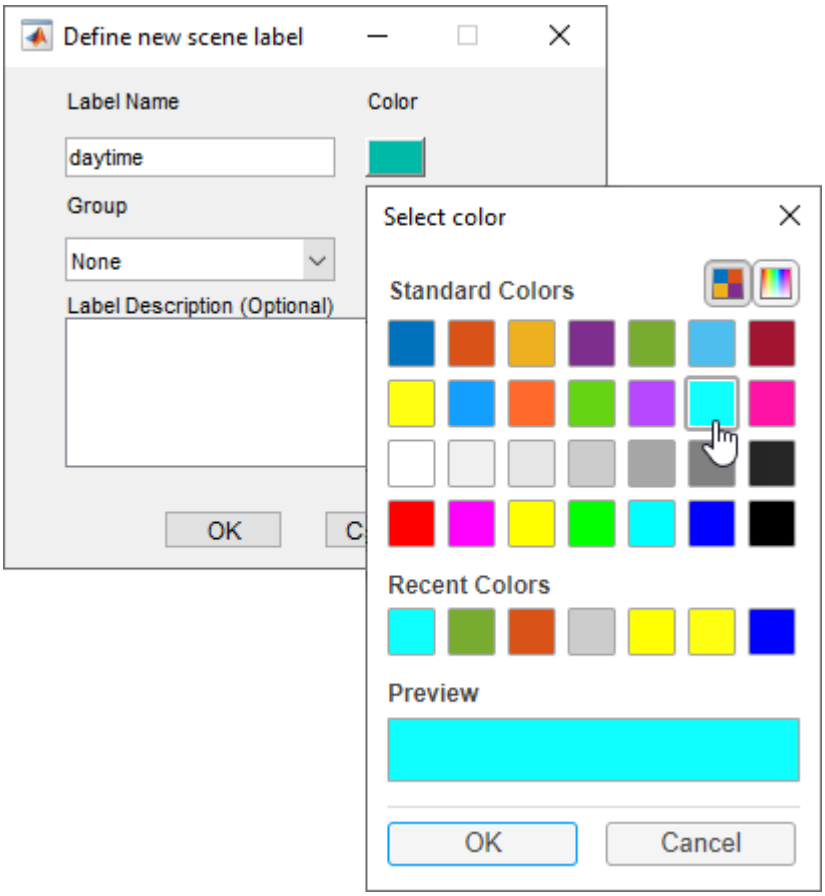
To delete an attribute, right-click an ROI label or sublabel, and select the attribute to delete. Deleting the attribute removes attribute information from all previously created ROI label annotations.

Create Scene Labels

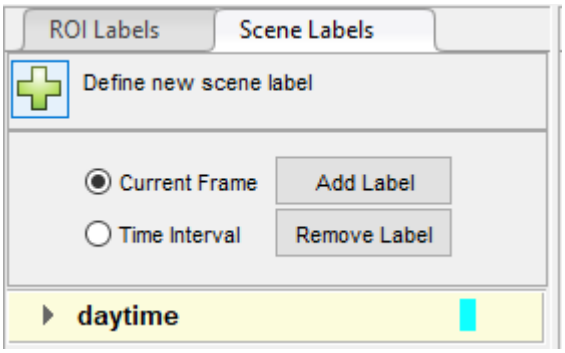
A scene label defines additional information for the entire scene. Use scene labels to describe conditions, such as lighting and weather, or events, such as lane changes.

Create a scene label to apply to an image.

1. In the left pane of the app, select the **Scene Labels** tab next to the **ROI Labels** tab.
2. Click **Define new scene label**, and in the **Label Name** box, enter a scene label named `daytime`.
3. Change the color of the label definition to light blue to reflect the nature of the scene label. Under the **Color** parameter, click the color preview and select the standard light blue colors. Then, click **OK** to close the color selection window.



4 Leave the **Group** parameter set to the default of `None` and click **OK**. The **Scene Labels** pane shows the scene label definition.



5 Click **Apply to Image** to apply the `daytime` label to the scene. A checkmark appears for the scene label.

6 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 range.

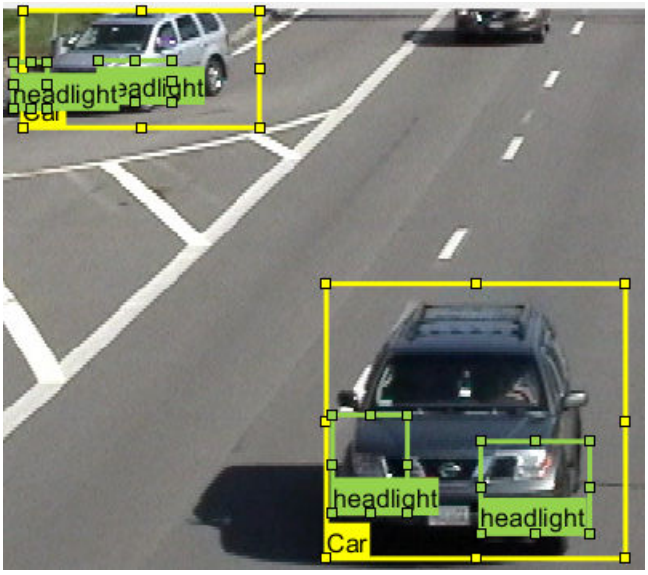
Advance frame by frame and draw the label and sublabel ROIs manually. Also update the attribute information for these ROIs.

Label Ground Truth Using Automation Algorithm

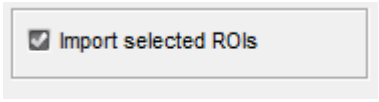
To speed up the labeling process, you can use an automation algorithm within the app. You can either define your own automation algorithm, see “Create Automation Algorithm for Labeling” on page 9-2 and “Temporal Automation Algorithms” on page 9-77, or use a built-in automation algorithm. In this example, 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 toolstrip, 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 9-73. 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 9-84.

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 17-27.

In this example, you export the labeled ground truth to the MATLAB workspace. From the app toolstrip, 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: [1×1 groundTruthDataSource]
    LabelDefinitions: [2x6 table]
           LabelData: [531×3 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: [531×1 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 =
  3×6 table

      Name         Type        LabelColor        Group       Description     Hierarchy
    _________    _________    ____________    ___________    ___________    ____________

    {'Car'  }    Rectangle    {1×3 double}    {'Vehicle'}    {0×0 char}     {1×1 struct}
    {'Truck'}    Rectangle    {1×3 double}    {'Vehicle'}    {0×0 char}     {0×0 double}
    {'Sunny'}    Scene        {1×3 double}    {'Weather'}    {0×0 char}     {0×0 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: [1×1 struct]
          color: [1×1 struct]
       inMotion: [1×1 struct]
        carType: [1×1 struct]
      headlight: [1×1 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: [1×1 struct]
```

Display information about the `carType` attribute.

```
gTruth.LabelDefinitions.Hierarchy{1}.carType
```

```
ans =

  struct with fields:

      ListItems: {3×1 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 range of 5–10 seconds was labeled.

```
labelData = gTruth.LabelData;
head(labelData)
```

```
ans =

  8×3 timetable

      Time           Car            Truck         sunny
    ___________    ___________    ___________    _____

    5.005 sec      [1×2 struct]    [1×0 struct]    true
    5.0384 sec     [1×2 struct]    [1×0 struct]    true
    5.0717 sec     [1×2 struct]    [1×0 struct]    true
    5.1051 sec     [1×2 struct]    [1×0 struct]    true
    5.1385 sec     [1×2 struct]    [1×0 struct]    true
    5.1718 sec     [1×2 struct]    [1×0 struct]    true
```

```
    5.2052 sec    [1×2 struct]    [1×0 struct]    true
    5.2386 sec    [1×2 struct]    [1×0 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 =

  8×3 timetable

      Time           Car            Truck        sunny
    __________    ____________    ____________    _____

    5.005 sec     [1×2 struct]    [1×0 struct]    true
    5.0384 sec    [1×2 struct]    [1×0 struct]    true
    5.0717 sec    [1×2 struct]    [1×0 struct]    true
    5.1051 sec    [1×2 struct]    [1×0 struct]    true
    5.1385 sec    [1×2 struct]    [1×0 struct]    true
    5.1718 sec    [1×2 struct]    [1×0 struct]    true
    5.2052 sec    [1×2 struct]    [1×0 struct]    true
    5.2386 sec    [1×2 struct]    [1×0 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 range. Your bounding box positions might differ from the ones shown here.

```
gTruthInterval(1,:).Car{1}.Position % [x y width height], in pixels
```

```
ans =

  1×4 single row vector

  415.8962   82.4737  130.8474  129.3805


ans =

  1×4 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 9-67.

See Also

Apps
Video Labeler

Objects
`labelDefinitionCreator` | `groundTruth` | `groundTruthDataSource` | `vision.labeler.mixin.Temporal` | `vision.labeler.AutomationAlgorithm`

More About

- “Use Custom Image Source Reader for Labeling” on page 9-57
- “Keyboard Shortcuts and Mouse Actions for Video Labeler” on page 9-59
- “Use Sublabels and Attributes to Label Ground Truth Data” on page 9-80
- “Label Pixels for Semantic Segmentation” on page 9-14
- “Create Automation Algorithm for Labeling” on page 9-2
- “View Summary of Ground Truth Labels” on page 9-73
- “Share and Store Labeled Ground Truth Data” on page 9-67
- “Training Data for Object Detection and Semantic Segmentation” on page 9-84

Use Custom Image Source Reader for Labeling

In this section...
“Create Custom Reader Function” on page 9-57
“Import Data Source into Video Labeler App” on page 9-57
“Import Data Source into Ground Truth Labeler App” on page 9-58

The **Video Labeler** and **Ground Truth Labeler** 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

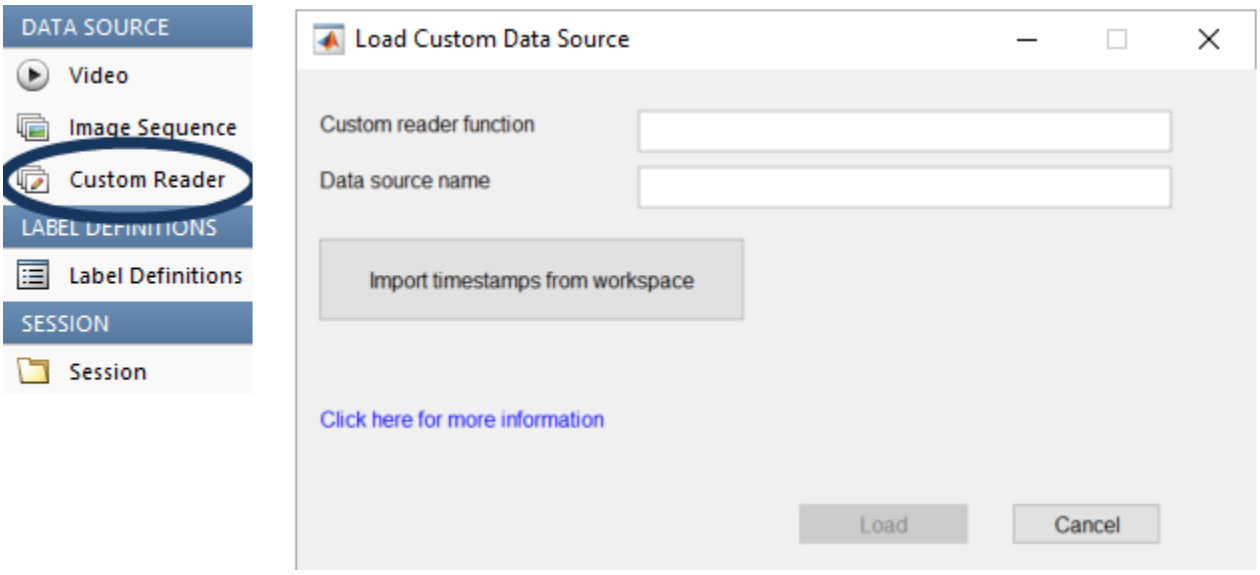
To import a custom data source into the **Video Labeler** app, first create a `groundTruthDataSource` object. This object stores the data source files and timestamps. Specify the name of the data source, the custom reader function handle that reads the data, and the timestamps by using this syntax.

```
gtSource = groundTruthDataSource(sourceName,readerFcn,timestamps)
```

To load this object into the app, at the MATLAB command prompt, enter this code.

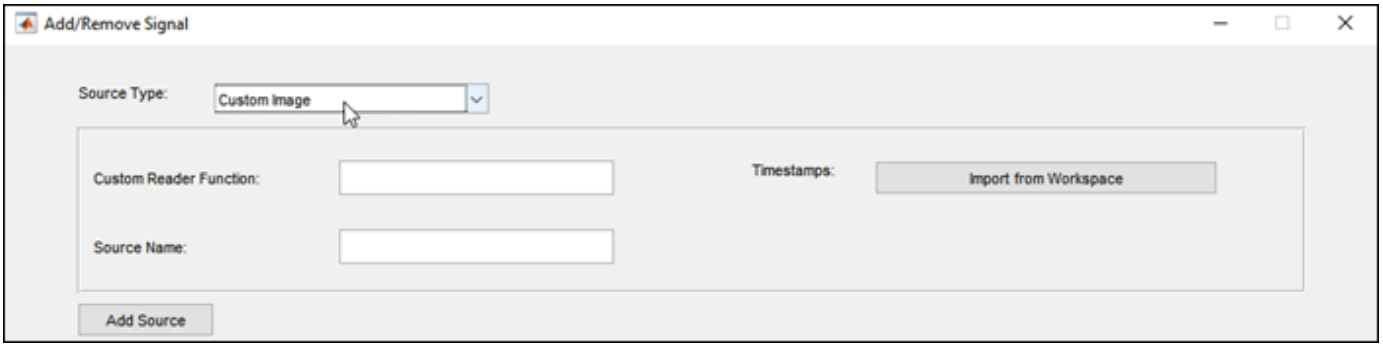
```
videoLabeler(gtSource)
```

Alternatively, on the toolstrip of the **Video Labeler** app, select **Import > 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 toolstrip, select **Import** > **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` | `groundTruthMultisignal` | `groundTruthDataSource`

More About

- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Get Started with the Video Labeler” on page 9-44

Keyboard Shortcuts and Mouse Actions for Video Labeler

Note On Macintosh platforms, use the **Command (⌘)** key instead of **Ctrl**.

Label Definitions

Task	Action
Navigate through ROI labels and their groups in the **ROI Label Definition** pane.	Up or Down arrow
Navigate through scene labels and their groups in the **Scene Label Definition** pane,	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 range 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	• PC: **End** • Mac: Hold **Fn** and press the right arrow
Go to the first frame	• PC: **Home** • Mac: Hold **Fn** and press the left arrow
Navigate through time range 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 • 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 9-80.	**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 9-44

Keyboard Shortcuts and Mouse Actions for Image Labeler

Note On Macintosh platforms, use the **Command (⌘)** key instead of **Ctrl**.

Label Definitions

Task	Action
Navigate through ROI labels and their groups in the **ROI Label Definition** pane.	Up or Down arrow
Navigate through scene labels and their groups in the **Scene Label Definition** pane,	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	• PC: **Page Up** and **Page Down** • Mac: Hold **Fn** and press the up and down arrows
Go to the first image	• PC: **Home** • Mac: Hold **Fn** and press the left arrow
Go to the last image	• PC: **End** • Mac: Hold **Fn** and press the right arrow
Select all images from the current image to the first image	• PC: **Shift+Home** • Mac: Hold **Fn+Shift** and press the left arrow
Select all images from the current image to the last image	• 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 • 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 9-80.	**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 toolstrip, 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	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

Image Labeler

More About

- “Get Started with the Image Labeler” on page 9-30

Share and Store Labeled Ground Truth Data

The **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** 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:

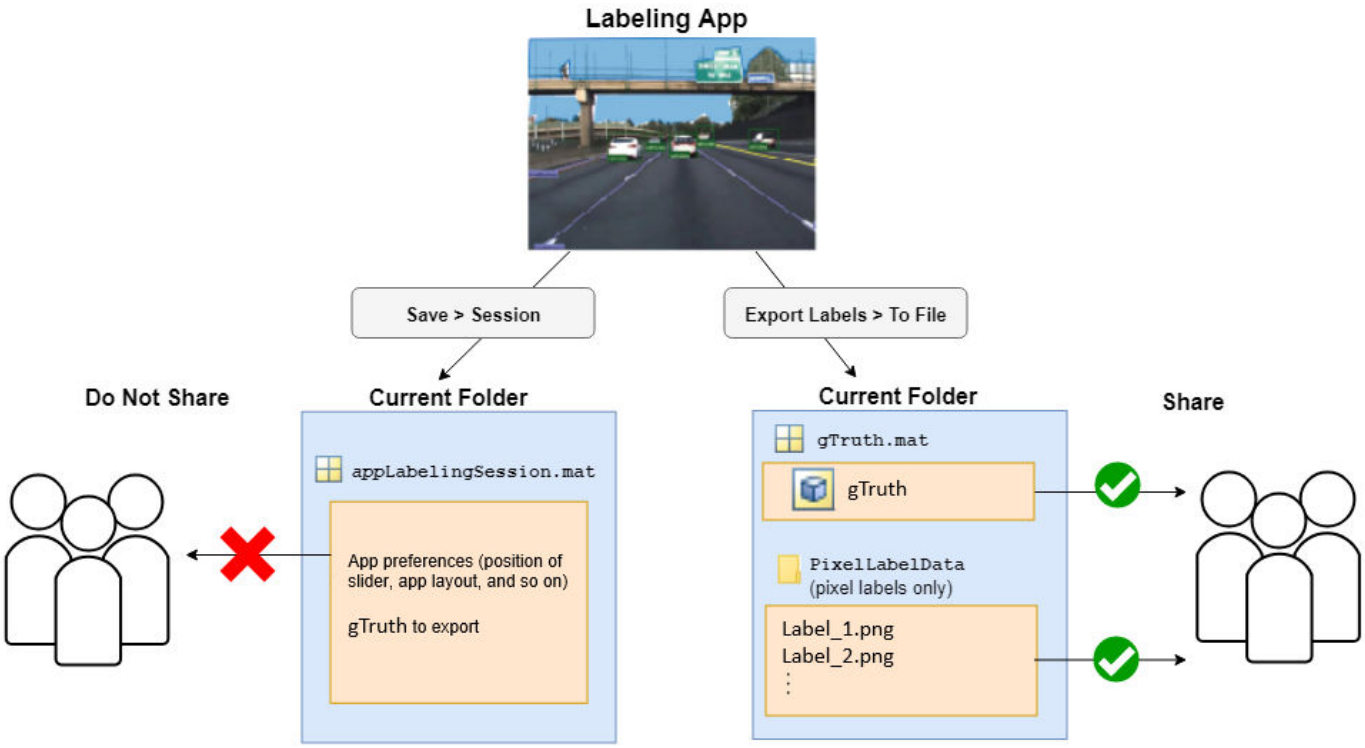
- Other labeling colleagues, who can use it to continue labeling
- Algorithm developers, who can use it to train algorithms, such as an object detector or semantic segmentation network
- Validation engineers, who can use it to validate algorithms

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.

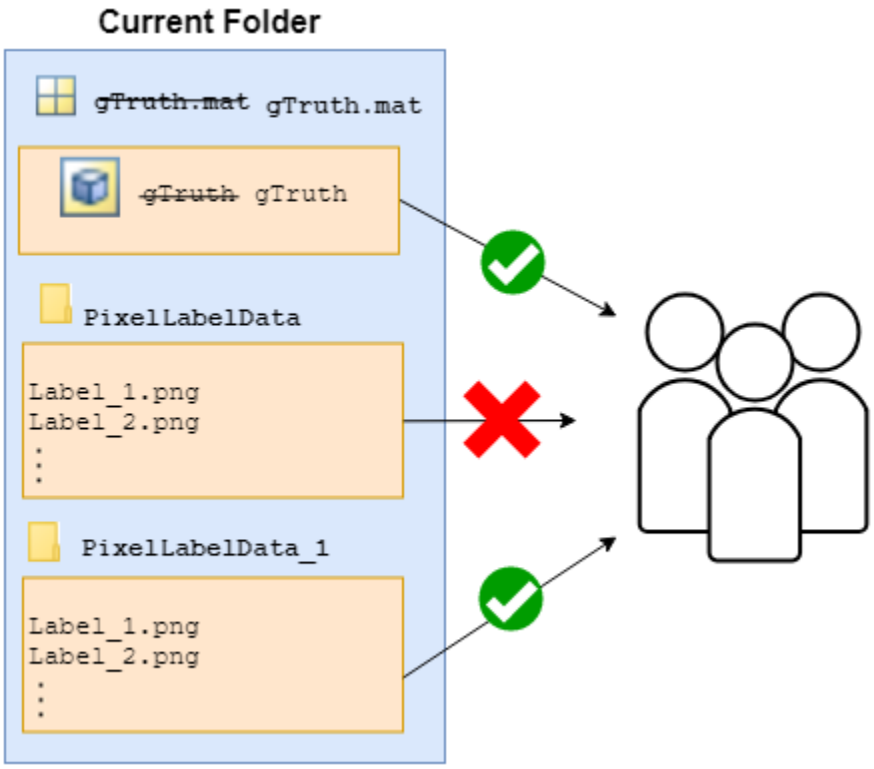
If the exported ground truth data contains pixel labels, the app also generates a `PixelLabelData` folder containing the pixel label data. The label data table stored in the ground truth object references the path to this folder. Share this folder along with the ground truth object.

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	• `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	• `groundTruth` object MAT-file • `PixelLabelData` folder (pixel labels only) • Video source file (if not in a shared location)
Image sequence	• `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	• `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	• `groundTruthMultisignal` object MAT-file • `PixelLabelData` folder (pixel labels only) • Video source file (if not in a shared location)
Image sequence	• `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	• `groundTruthMultisignal` object MAT-file • `PixelLabelData` folder (pixel labels only) • Data source files (if not in a shared location) • Custom reader function
Point cloud sequence	• `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	• `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	• `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.

Extract Labeled Video Scenes

You can extract labeled video scenes and corresponding labels from a `groundTruth` or `groundTruthMultisignal` object exported by the **Video Labeler** or **Ground Truth Labeler** apps in a form that can be loaded using a datastore. These labeled video scenes can be used for training, validation, and evaluation. Extracted videos scenes enable you to employ video labels on specific

durations of a long video and apply deep learning techniques to detect anomalies or recognize a specific activity in a video.

- Use the `sceneTimeRanges` function to obtain labeled scene time ranges from a ground truth object.
- Use the `writeVideoScenes` function to take a ground truth object array and labeled scene time ranges information from the `sceneTimeRanges` function to write video scenes to a folder. This data can be directly used with datastore based deep learning training and evaluation workflows.

See Also

Apps
Video Labeler | **Ground Truth Labeler** | **Image Labeler**

Objects
`groundTruth` | `groundTruthDataSource` | `groundTruthMultisignal`

Functions
`changeFilePaths (groundTruth)` | `changeFilePaths (groundTruthMultisignal)` | `writeVideoScenes` | `sceneTimeRanges`

More About

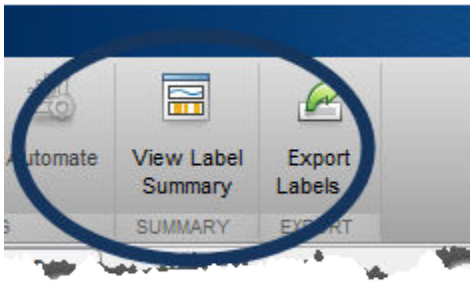
- “How Labeler Apps Store Exported Pixel Labels” on page 17-27

View Summary of Ground Truth Labels

In this section...
“View Label Summary” on page 9-73 “Compare Selected Labels” on page 9-75

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 9-40.

You can view and compare the distribution of ROI and scene labels by clicking **View Label Summary** on the app toolstrip.

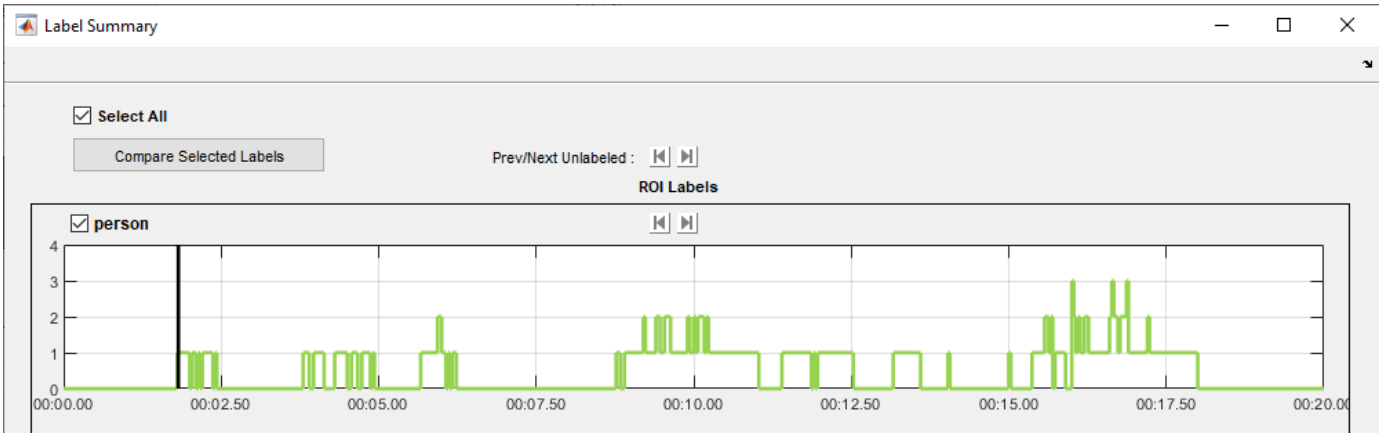


View Label Summary

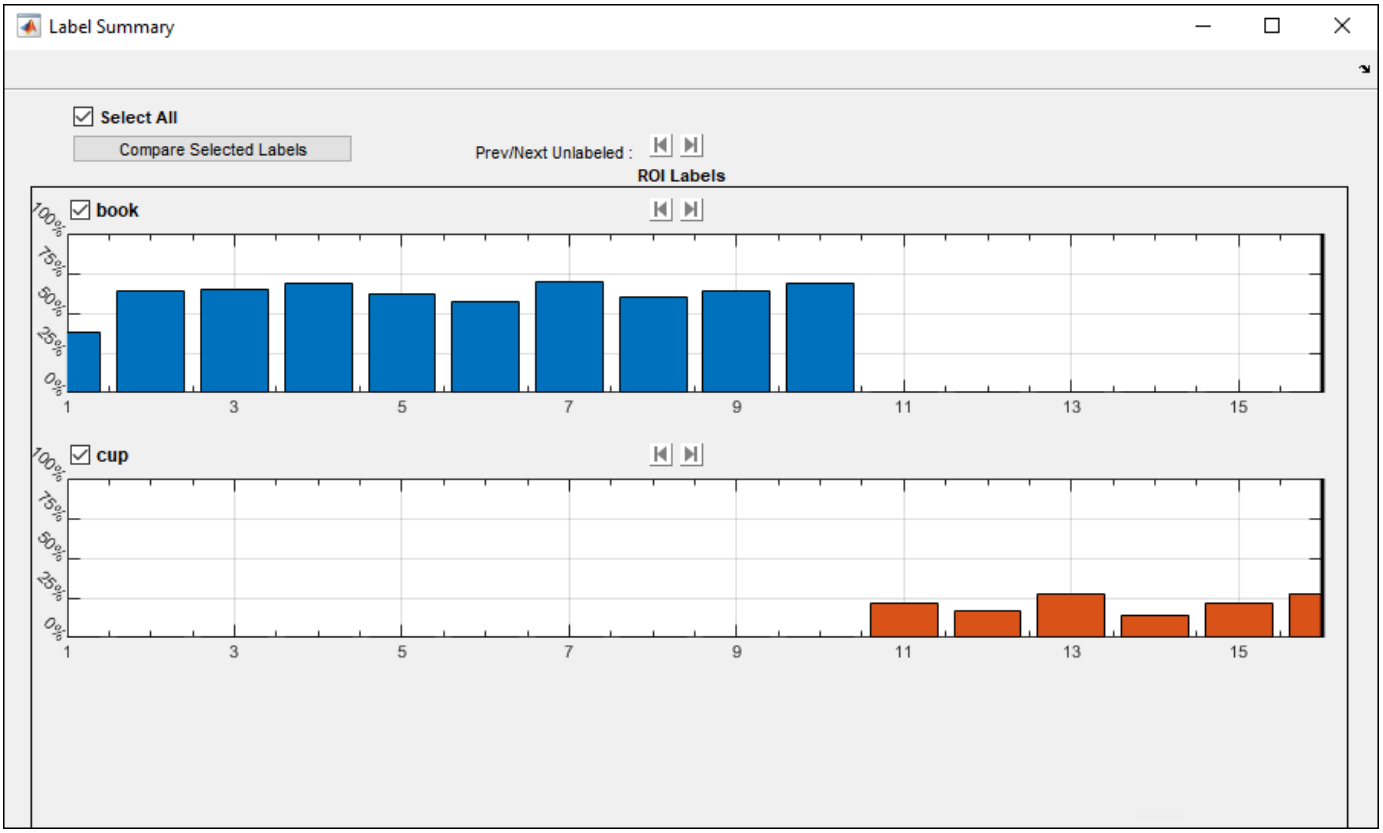
Clicking **View Label Summary** opens dockable distribution graphs for the ROI and scene labels.

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.

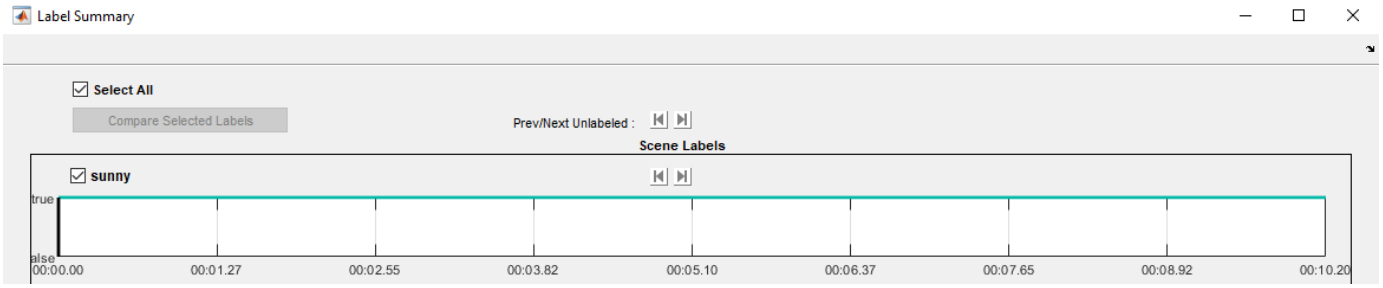
For all ROI labels except pixels, the *y*-axis displays the number of ROIs at each timestamp or for each image. The visual summary does not include information about sublabels or label attributes.



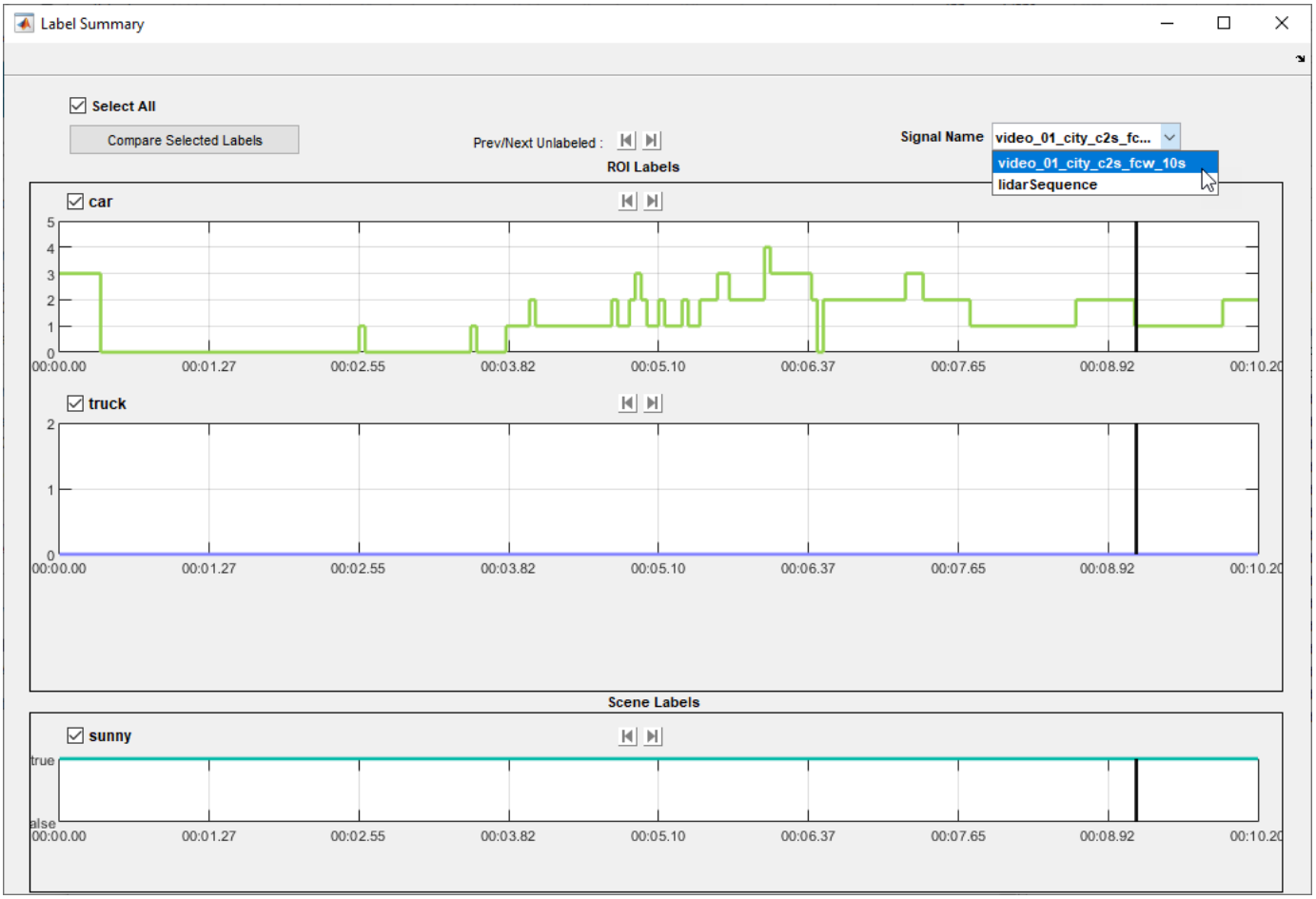
For pixel ROI labels, the *y*-axis displays the percentage of the frame that is labeled for each pixel label.



For scene labels, the graph displays the presence or absence of a scene label at each timestamp or for each image in a collection.

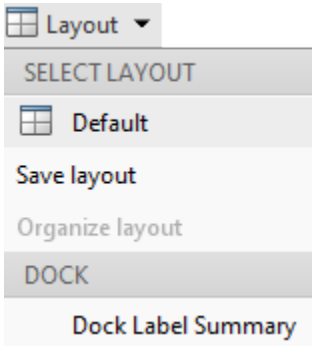


In the **Ground Truth Labeler** app, you can view labels by signal. From **Signal Name**, select a signal to view a summary of the labels for that signal.



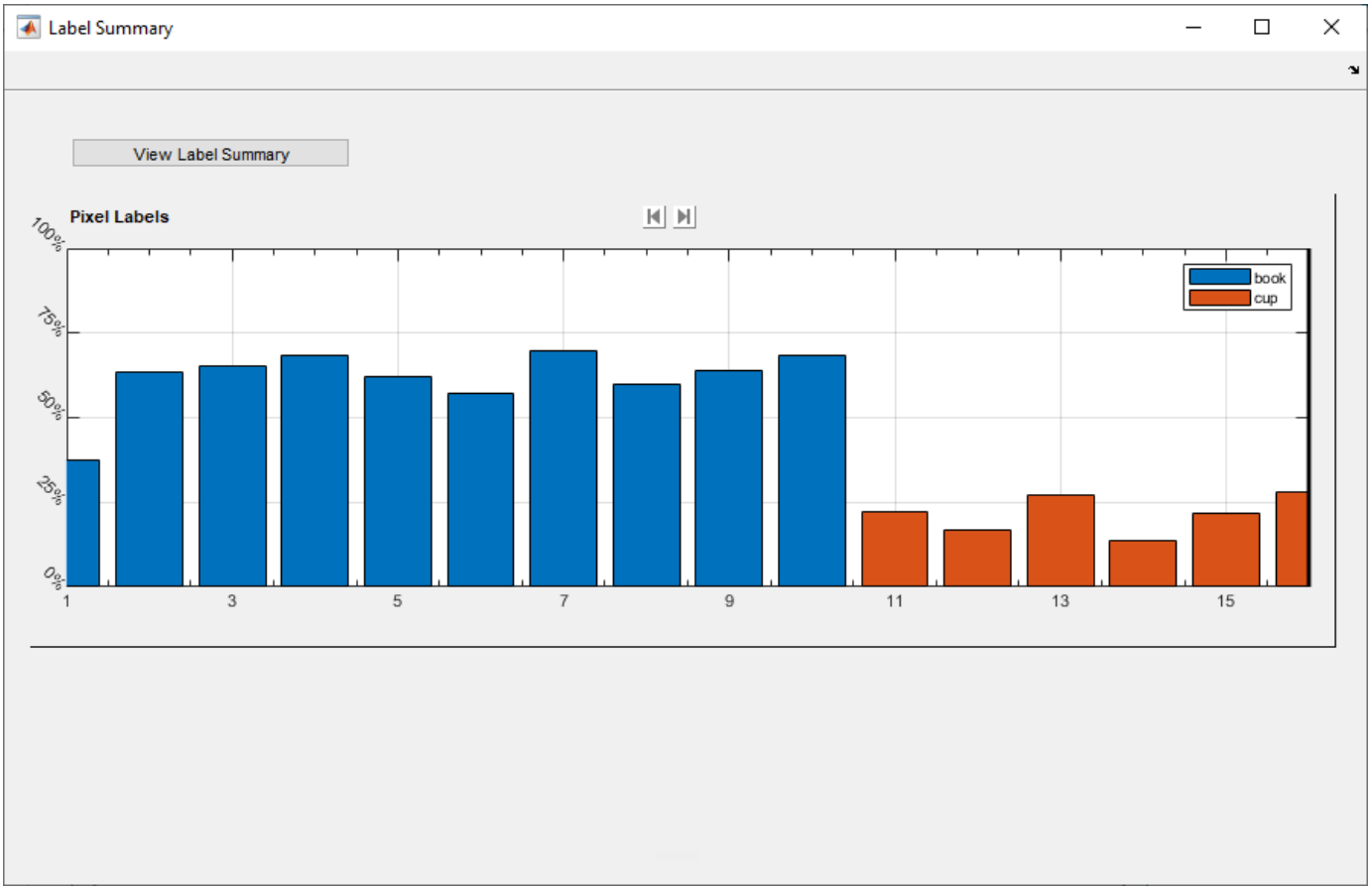
Use the graphs to examine the occurrence of labels over time or in relation to each other. Drag the black vertical line in any graph to move to a different timestamp or image in a collection.

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 9-40
- “Get Started with the Image Labeler” on page 9-30
- “Get Started with the Video Labeler” on page 9-44
- “Get Started with the Ground Truth Labeler” (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
Direction of automation: Forward Reverse	Run automation from: Start time to End time Current time to End time	
	Run automation from: Start time to End time Current time to End time	
Direction of automation: Forward Reverse	Run automation from: End time to Start time Current time to Start time	
	Run automation from: End time to Start time Current time to Start time	

See Also

Apps
Lidar Labeler | Image Labeler | Ground Truth Labeler

Functions
`vision.labeler.AutomationAlgorithm` | `vision.labeler.mixin.Temporal`

See Also

Blocked Image Automation Algorithms

The labeling apps enable you to create and import a custom automation algorithm to automatically label your data. Automation algorithms can be time-independent or time-dependent.

The **Image Labeler** app supports blocked image algorithms.

Create Blocked Image Automation Algorithm

To create a blocked image 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.BlockedImageAutomation` classes, as shown by the class definition of the template:

```
classdef MyCustomAlgorithm < vision.labeler.AutomationAlgorithm && vision.labeler.mixin.BlockedImageAutomation
```

Blocked image algorithms must inherit from both of these classes. Inheriting from the blocked image mixin class enables you to access properties such as `BatchSize`, `BorderSize` and `InclusionThreshold` to design blocked image based algorithms. For more details on enabling blocked image properties, see the `vision.labeler.mixin.BlockedImageAutomation` 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 Blocked Image Automation Algorithm

To run your blocked image algorithm from the labeling app, 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.

See Also

Apps

Image Labeler

Functions

`vision.labeler.AutomationAlgorithm` | `vision.labeler.mixin.BlockedImageAutomation`

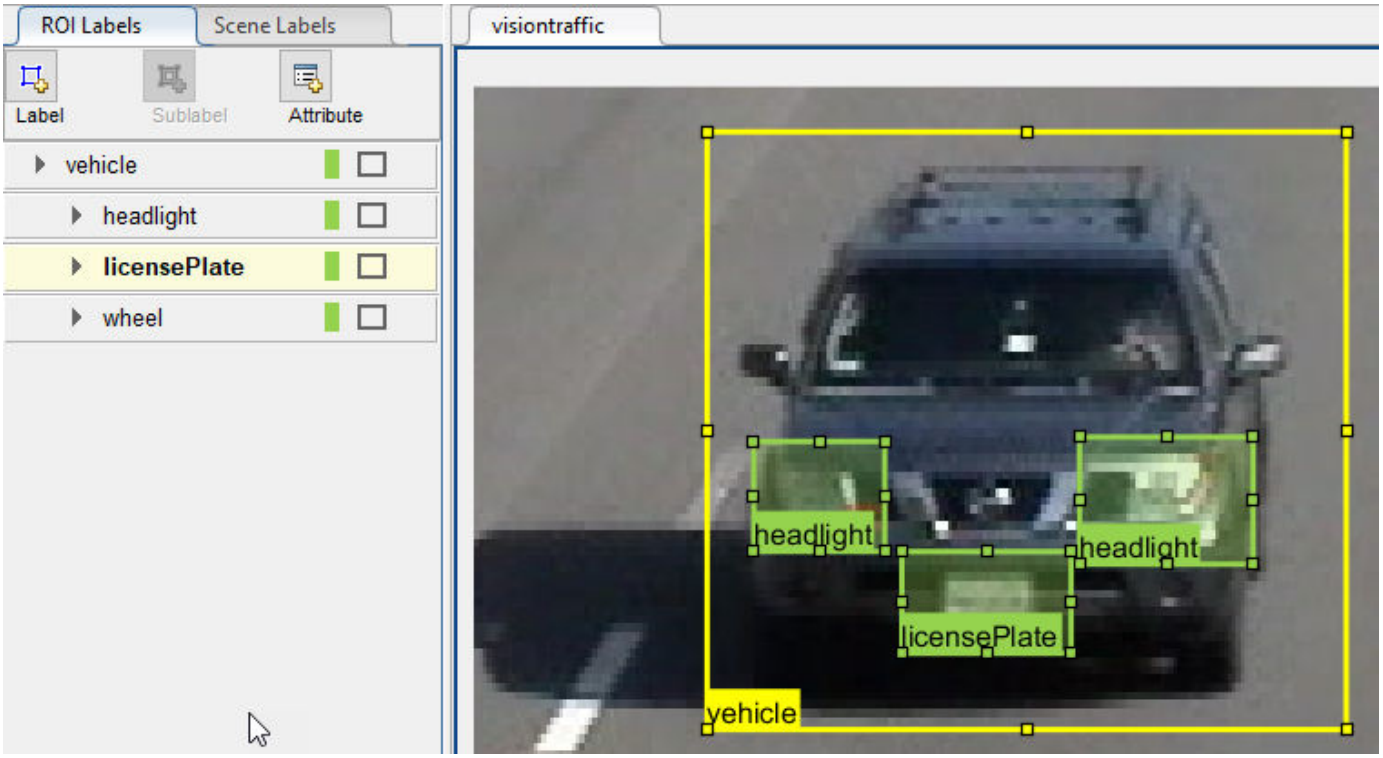
Related Examples

- “Get Started with the Image Labeler” on page 9-30

Use Sublabels and Attributes to Label Ground Truth Data

In the **Image Labeler**, **Video Labeler**, and **Ground Truth Labeler** 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**:

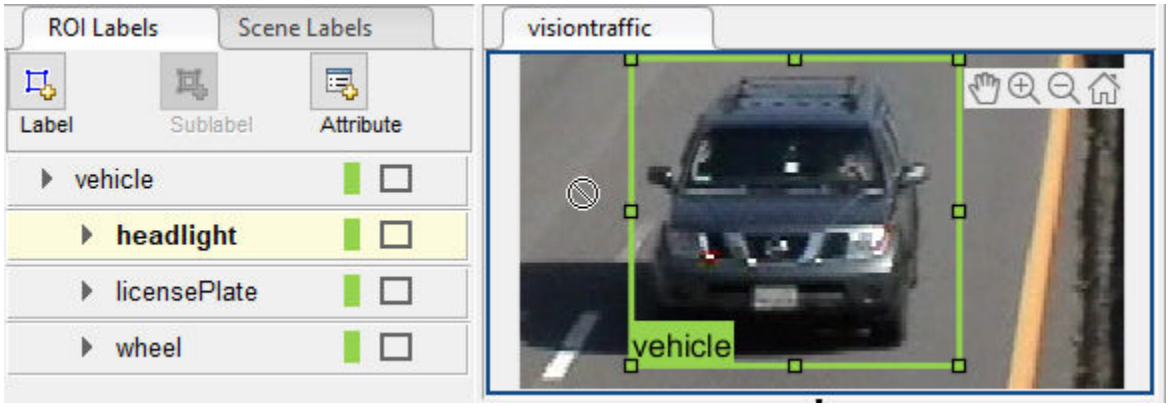
- A **wheel** is a good candidate for a *sublabel*. A wheel is part of a vehicle, and you can draw a label around a wheel.
- **Vehicle color** is a good candidate for an *attribute*. You cannot draw a label around the color of a vehicle.
- **Vehicle type** (car, truck, and so on) is a good candidate 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

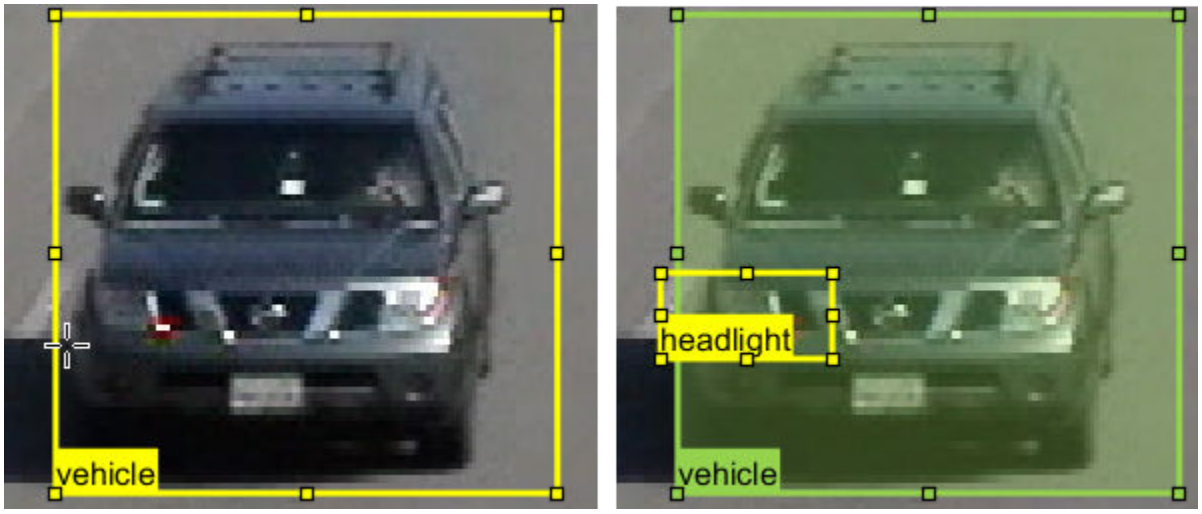
Within each frame, each sublabel that you draw must be associated with a parent label. Therefore, before you can draw a sublabel on a frame, you must:

1. From the **ROI Labels** 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.

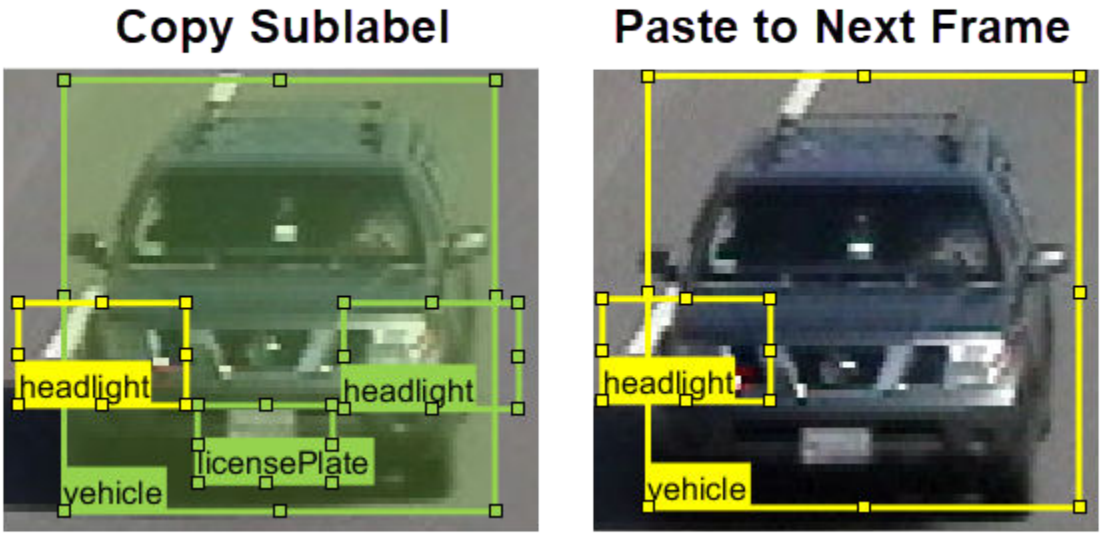


Notice that sublabels do not have to be completely enclosed within the parent label. You can drag sublabels outside the bounds of the parent label and the parent-child relationship remains unchanged.

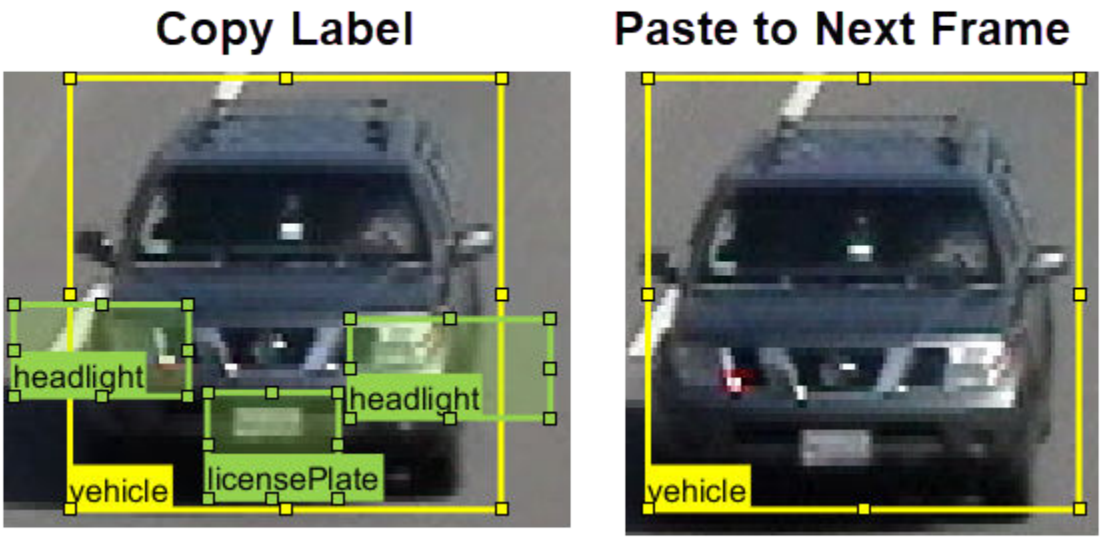
Copy and Paste Sublabels

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.



If you copy a parent label, however, the associated sublabels are not copied over.



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 Labels** 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
Image Labeler | Video Labeler | Ground Truth Labeler

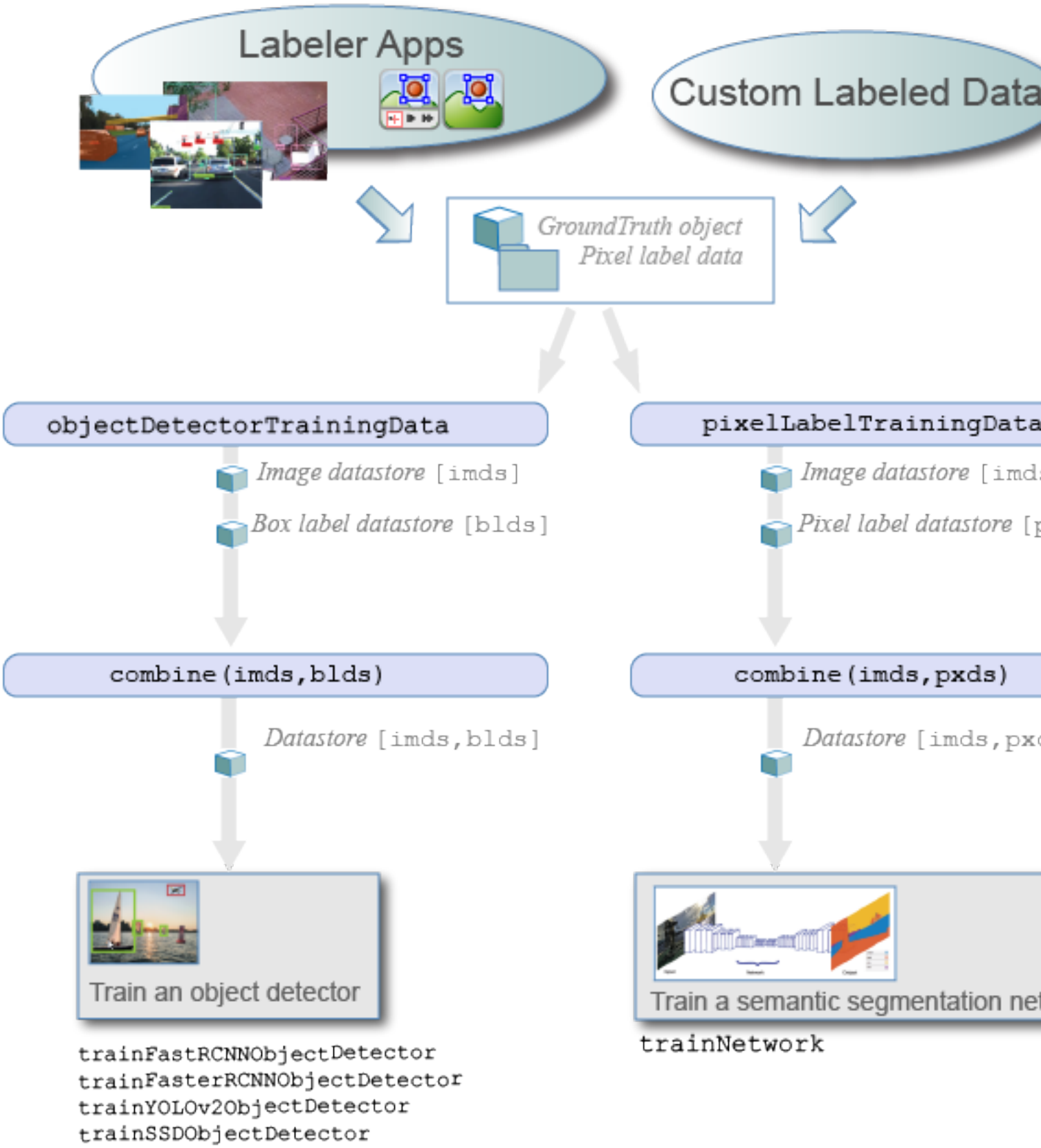
More About

- “Get Started with the Image Labeler” on page 9-30
- “Get Started with the Video Labeler” on page 9-44
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Label Pixels for Semantic Segmentation” on page 9-14
- “Automate Attributes of Labeled Objects” (Automated Driving 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.


Labeler Apps
Custom Labeled Data
GroundTruth object
Pixel label data
objectDetectorTrainingData
Image datastore [imds]
Box label datastore [blds]
combine(imds,blds)
Datastore [imds,blds]
Train an object detector
trainFastRCNNObjectDetector
trainFasterRCNNObjectDetector
trainYOLOv2ObjectDetector
trainSSDObjectDetector
pixelLabelTrainingData
Image datastore [imds
Pixel label datastore [p
combine(imds,pxds)
Datastore [imds,pxd
Train a semantic segmentation net
trainNetwork

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 9-30
   - **Video Labeler** — “Get Started with the Video Labeler” on page 9-44

   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 9-2.
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 9-67.
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”. 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 17-62.

See Also

Apps
Image Labeler | Video Labeler

Functions
`semanticseg` | `objectDetectorTrainingData` | `trainRCNNObjectDetector` | `trainRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `trainACFObjectDetector` | `trainYOLOv2ObjectDetector` | `trainSSDObjectDetector` | `pixelLabelTrainingData`

Objects
`groundTruth` | `groundTruthDataSource`

More About

- “Get Started with the Image Labeler” on page 9-30
- “Get Started with the Video Labeler” on page 9-44
- “Create Automation Algorithm for Labeling” on page 9-2
- “Getting Started with Object Detection Using Deep Learning” on page 17-24
- “Getting Started with Semantic Segmentation Using Deep Learning” on page 17-62
- “Getting Started with Point Clouds Using Deep Learning” on page 12-2
- “Anchor Boxes for Object Detection” on page 17-32

Featured Examples

- “Localize and Read Multiple Barcodes in Image” on page 10-2
- “Monocular Visual Odometry” on page 10-22
- “Detect and Track Vehicles Using Lidar Data” on page 10-35
- “Semantic Segmentation Using Dilated Convolutions” on page 10-54
- “Define Custom Pixel Classification Layer with Tversky Loss” on page 10-59
- “Track a Face in Scene” on page 10-66
- “Create 3-D Stereo Display” on page 10-71
- “Measure Distance from Stereo Camera to a Face” on page 10-72
- “Reconstruct 3-D Scene from Disparity Map” on page 10-73
- “Visualize Stereo Pair of Camera Extrinsic Parameters” on page 10-76
- “Remove Distortion from an Image Using the Camera Parameters Object” on page 10-79

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)
```

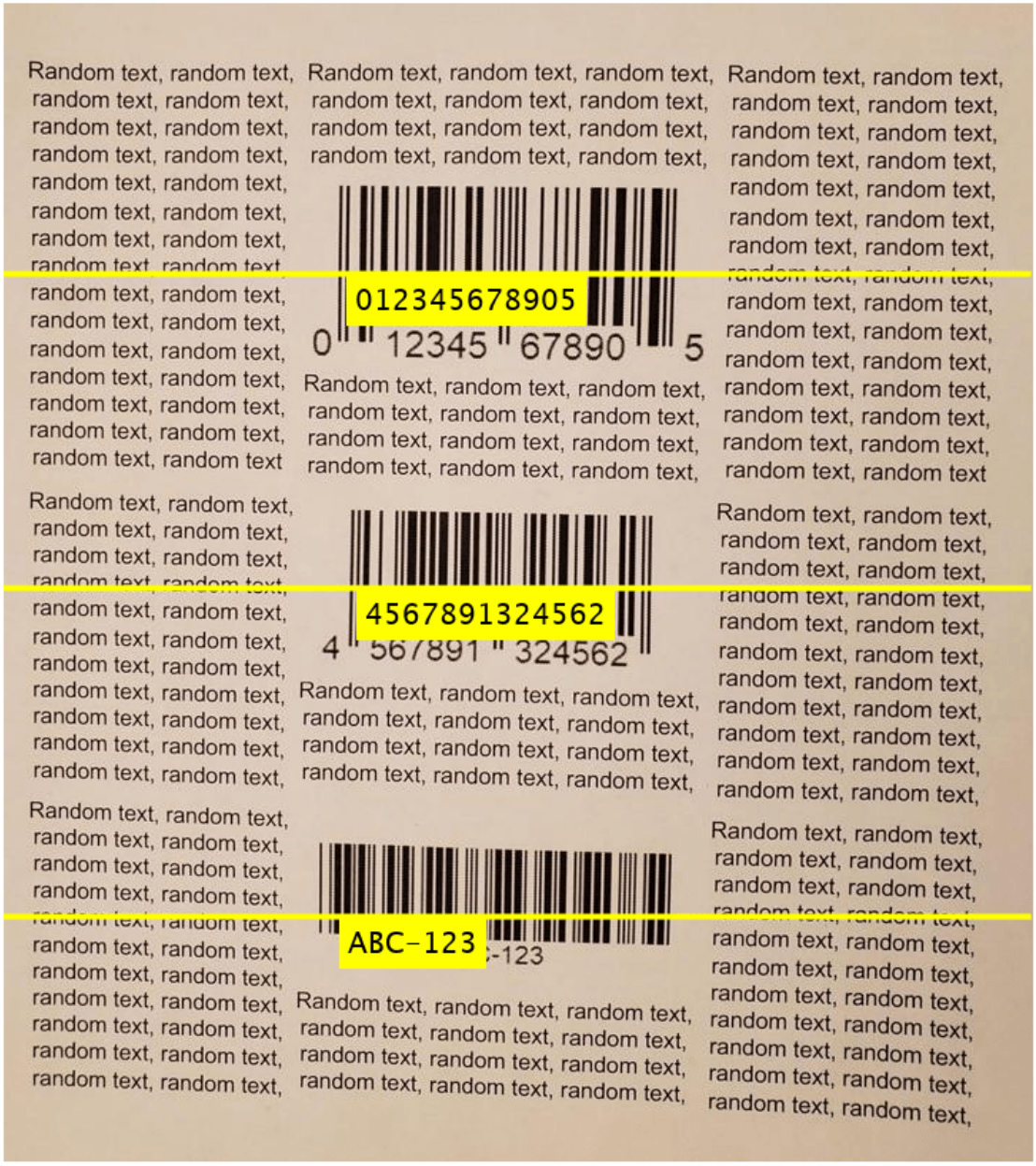
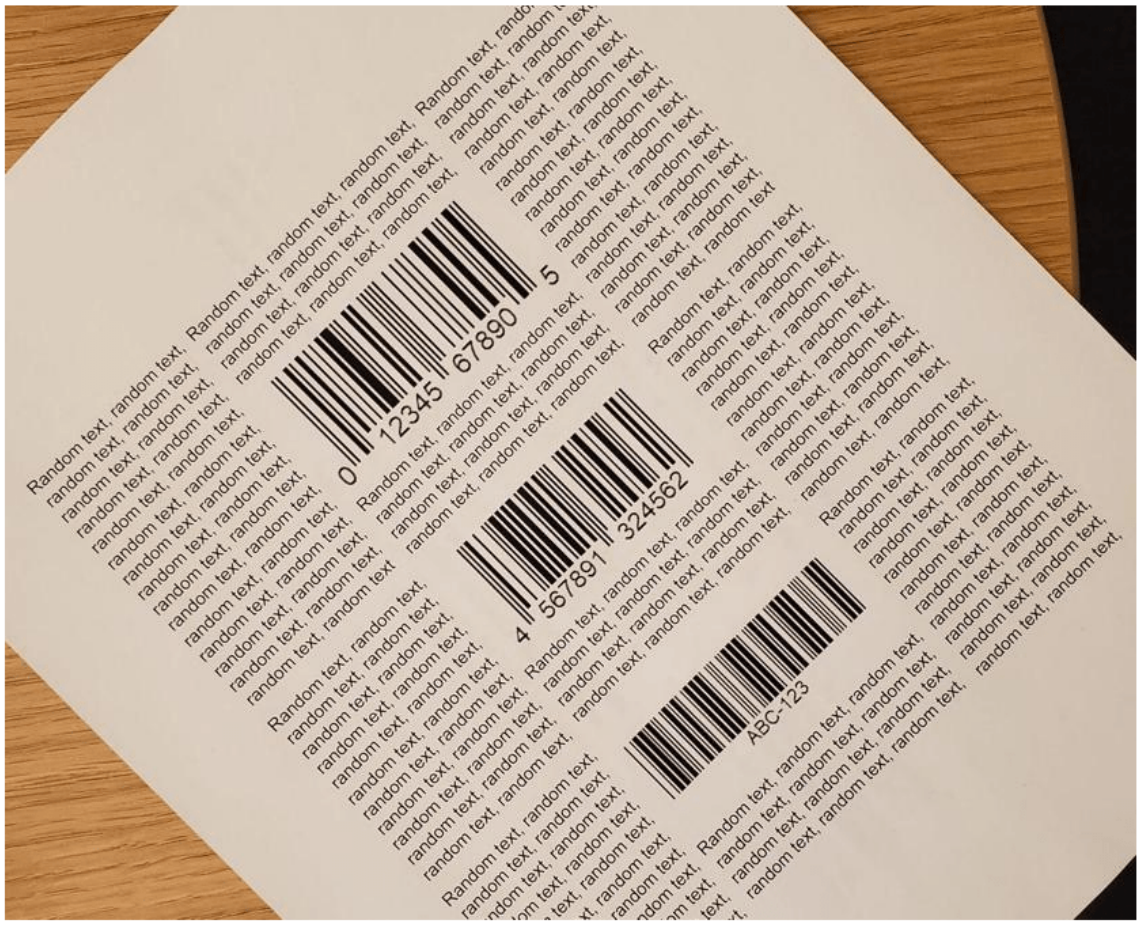


Image analysis to determine ROIs

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 = im2gray(I);

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



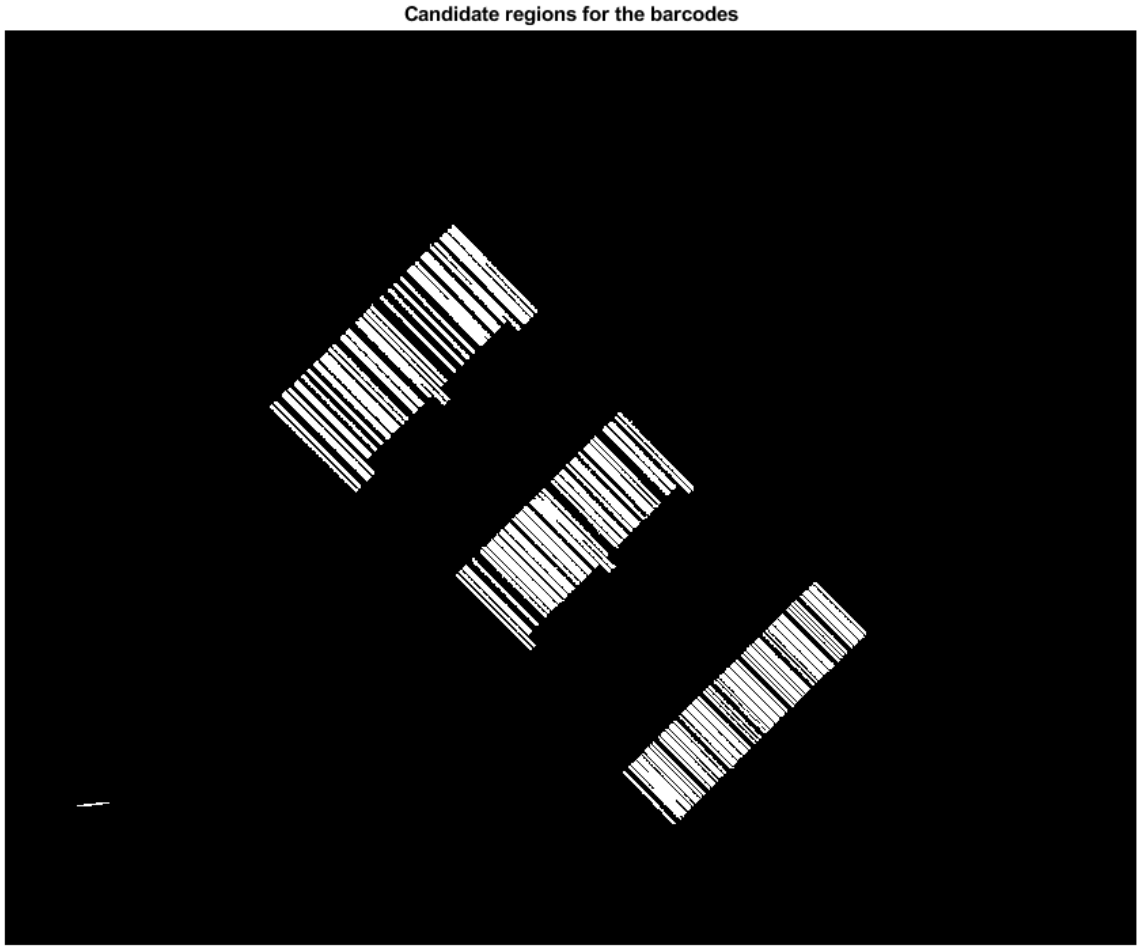

```
% Pass the unprocessed image to the readBarcode function.
readBarcode(Igray, '1D')

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.

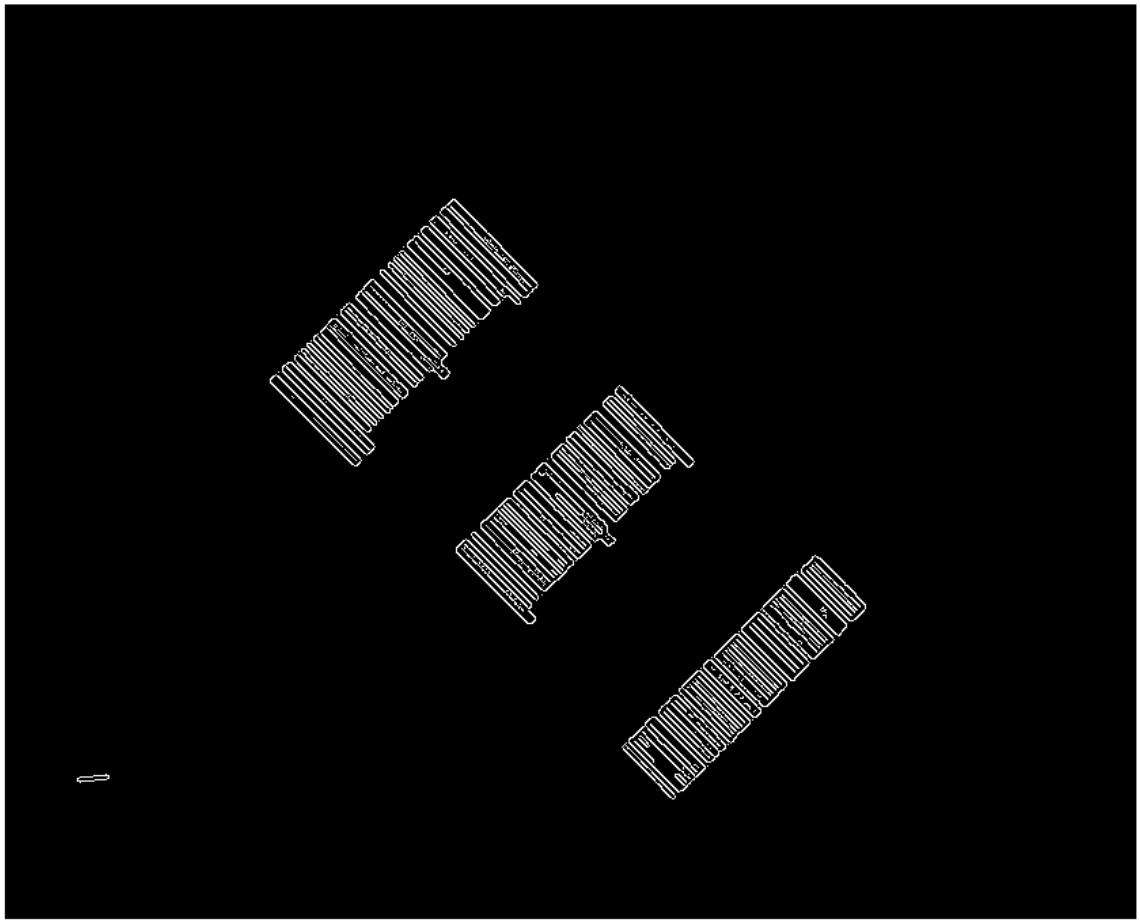
```
% Detect MSER features.
[~, cc] = detectMSERFeatures(Igray);

% 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

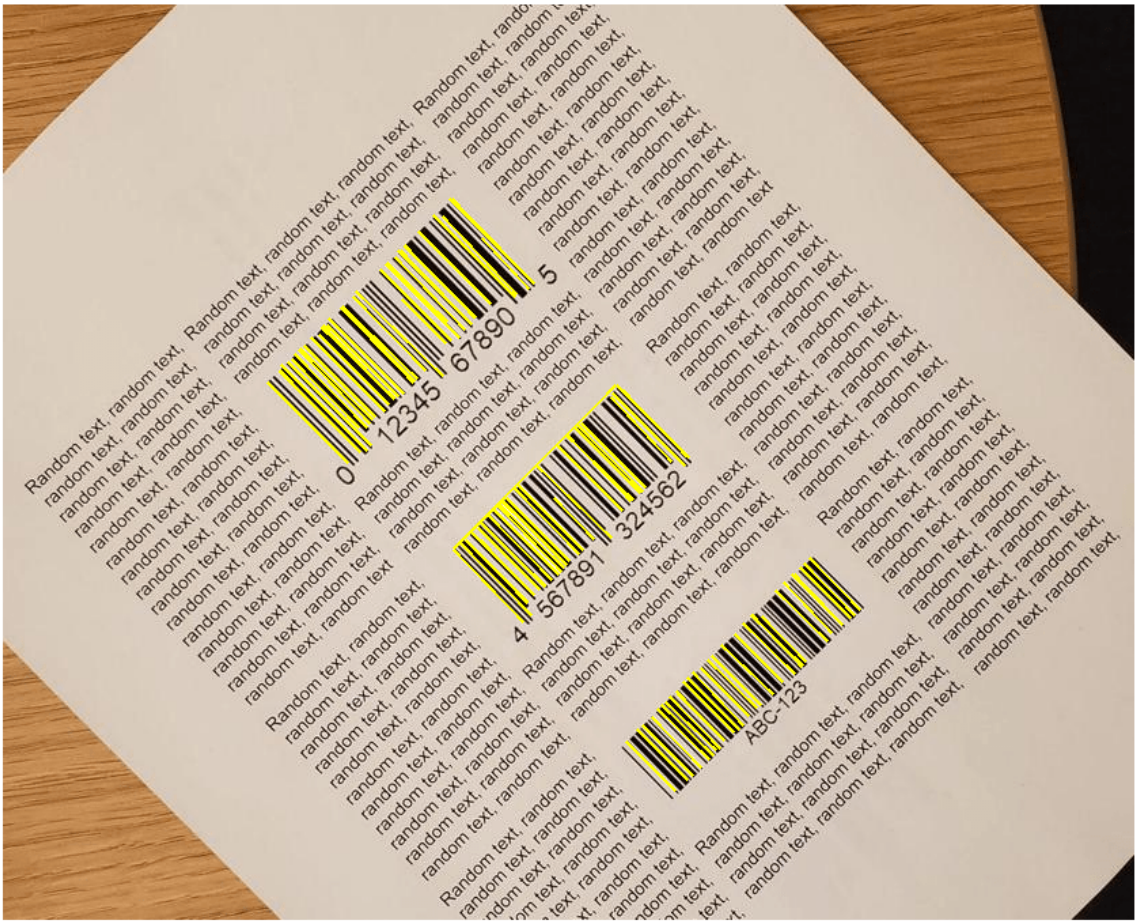
Detect prominent edges in the image using the `edge` function. Then use the hough transform to find lines of interest. The lines represent possible candidates for the vertical bars in the barcode.

```
% 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.
Ihoughlines = 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))';
```

```
Ihoughlines = insertShape(Ihoughlines, 'Line', [startPts, endPts], ...
    'LineWidth', 2, 'Color', 'green');

% Display the original image overlayed with the detected lines.
Ibarlines = imoverlay(I, ~Ihoughlines(:,:,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

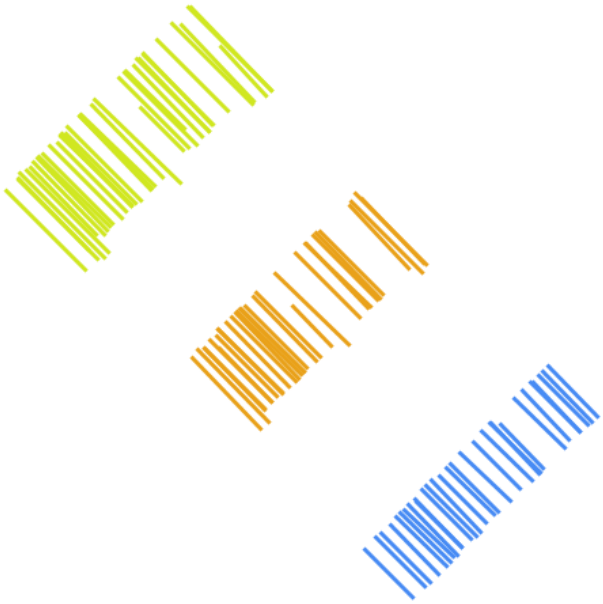
While it is common practice to directly use the lines (that were obtained using the Hough transform) to localize the barcode, this method uses the lines to further detect the perpendicular bisectors for each of the lines. The bisector lines are represented as points in cartesian space, which makes them 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 identity 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 Learni
end
```



Method 2: Segmentation based workflow

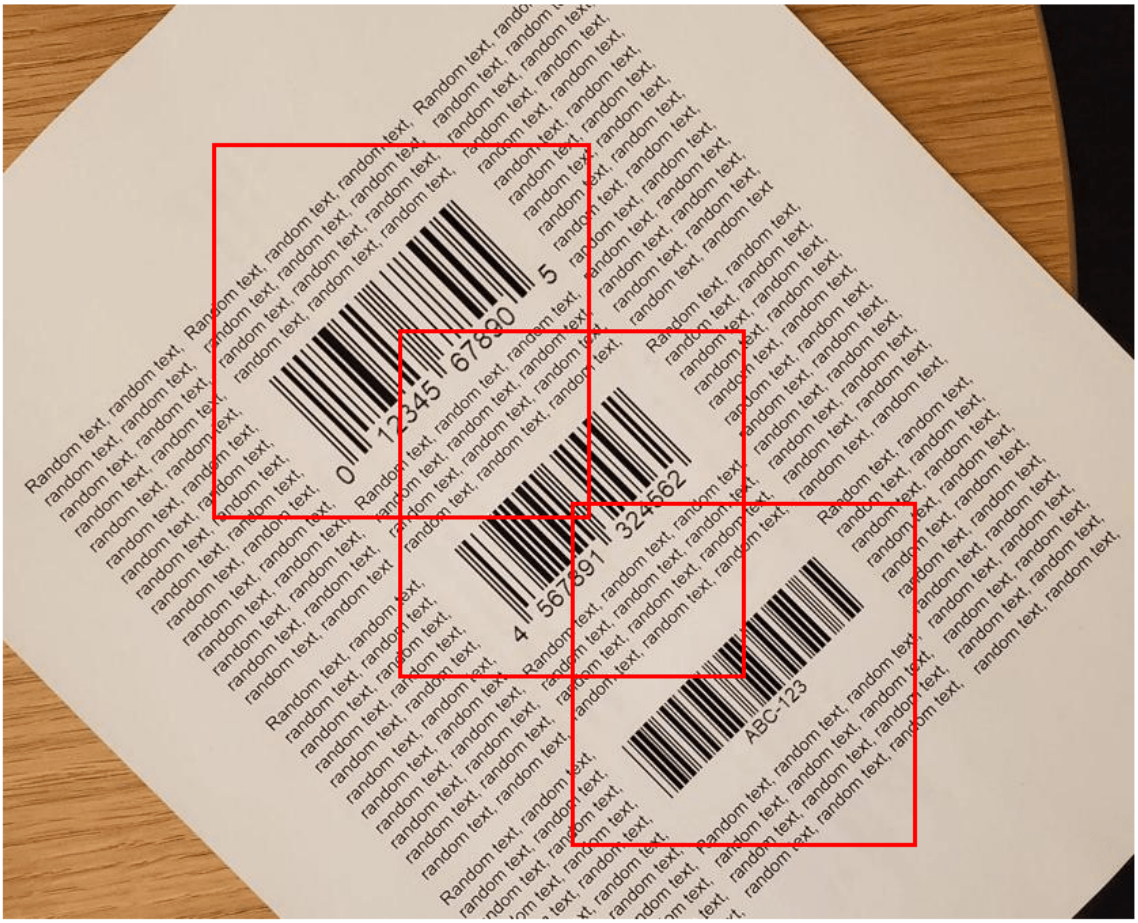
Having removed the background noise and variation, the detected vertical bars are grouped into individual barcodes using morphological operations, like `imdilate`. The example uses the `regionprops` function to determine the bounding box and orientation for each of the barcodes. The results are used to crop the individual barcodes from the original image and to orient them to be roughly horizontal.

```
if ~useClustering
    [boundingBox, orientation, Idilated] = segmentationLocalization(Ihoughlines);

    % 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, Iclusters] = 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(Ihoughlines)

%-------------------------------------------------------------------------
% Use image dilation to separate the barcodes
%-------------------------------------------------------------------------

% Create binary image with the detected lines.
Ibw = ~Ihoughlines(:,:,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.

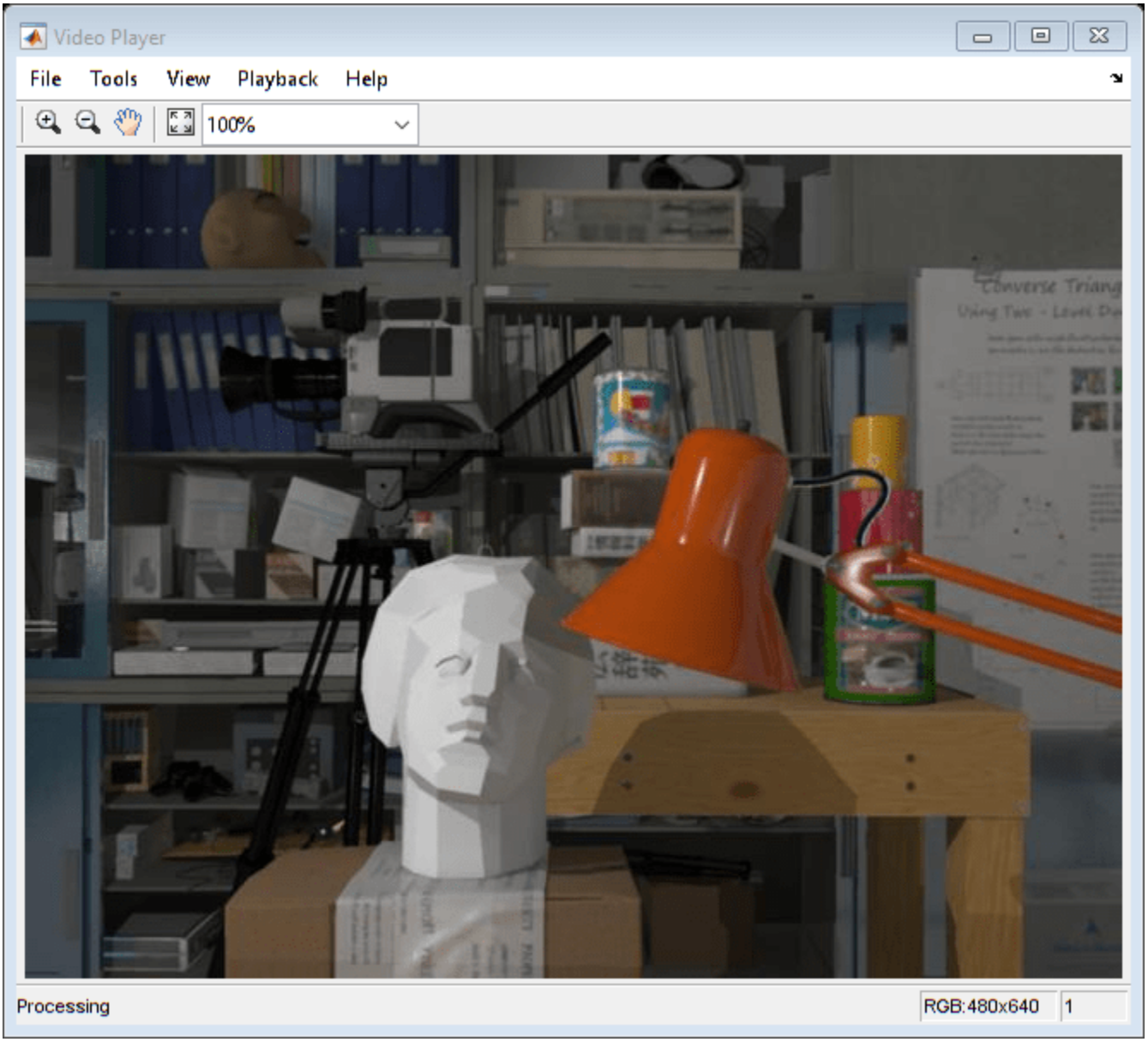
```
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', 'NewTsukuba'));

% 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.

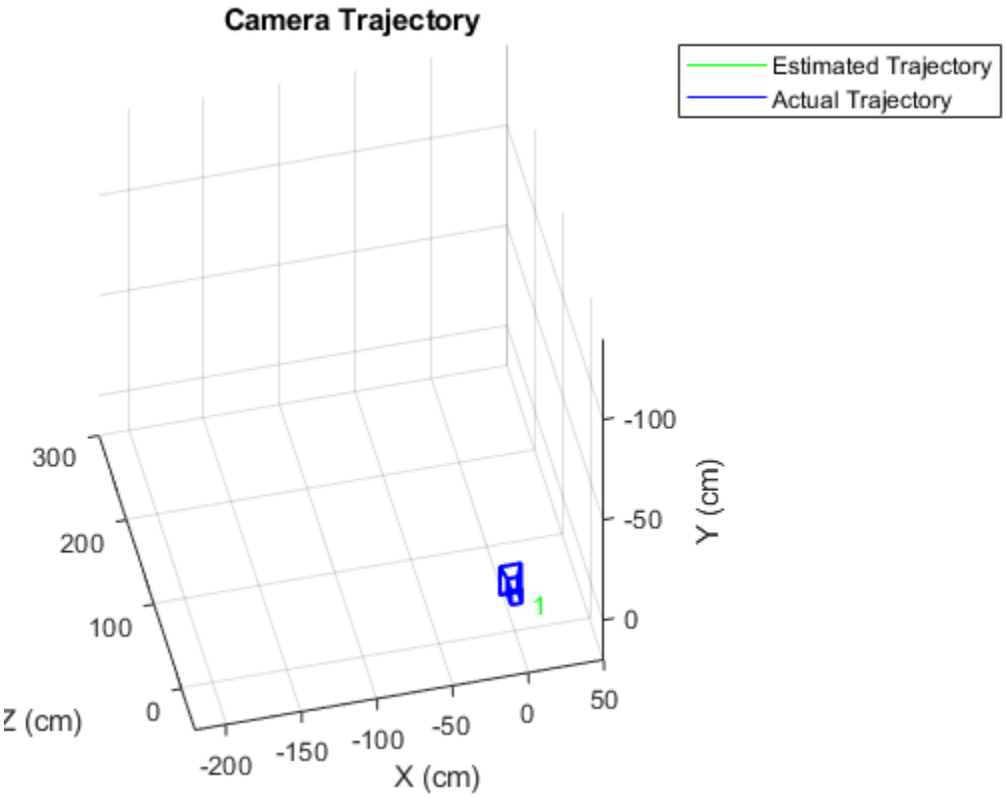
```
prevI = undistortImage(im2gray(Irgb), intrinsics);

% 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-');
```

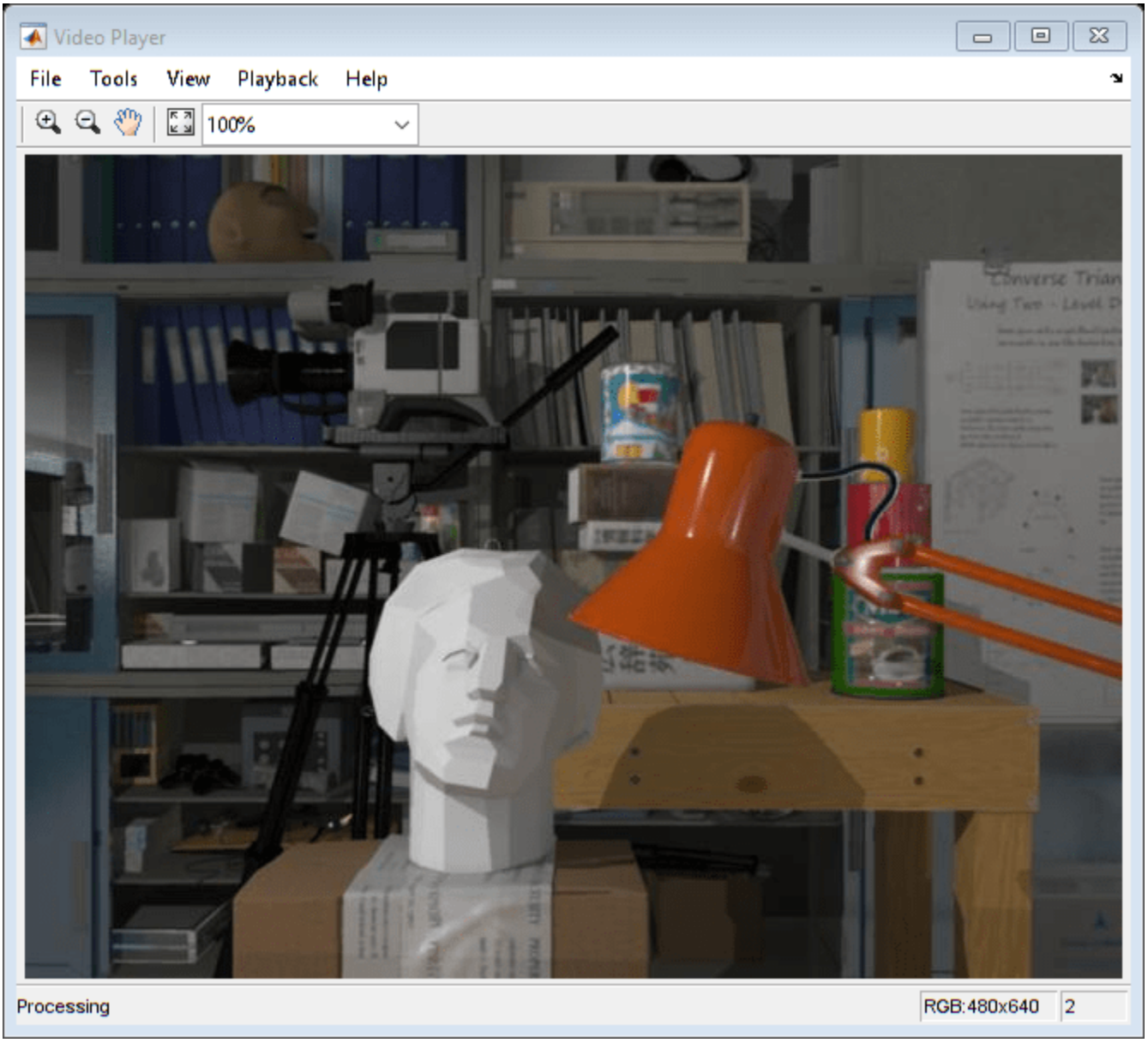
```
trajectoryActual    = plot3(0, 0, 0, 'b-');

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(im2gray(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, :);
```

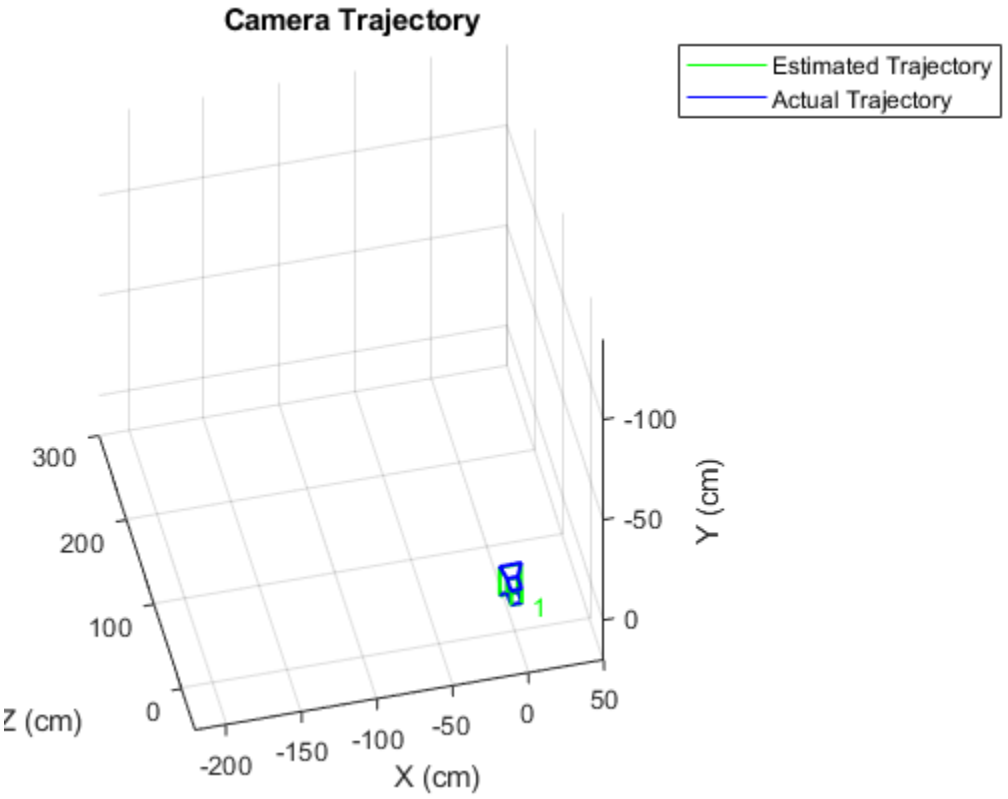
```
% Add the current view to the view set.
vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);

% 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;
```

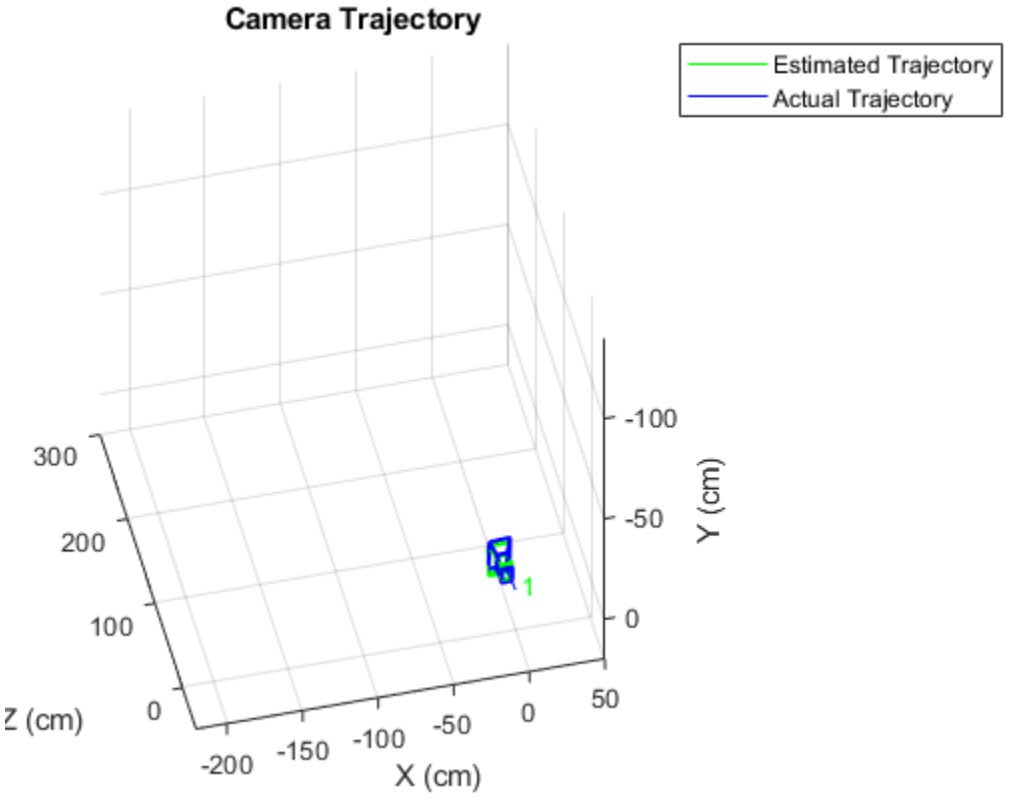
Bootstrap Estimating Camera Trajectory Using Global Bundle Adjustment

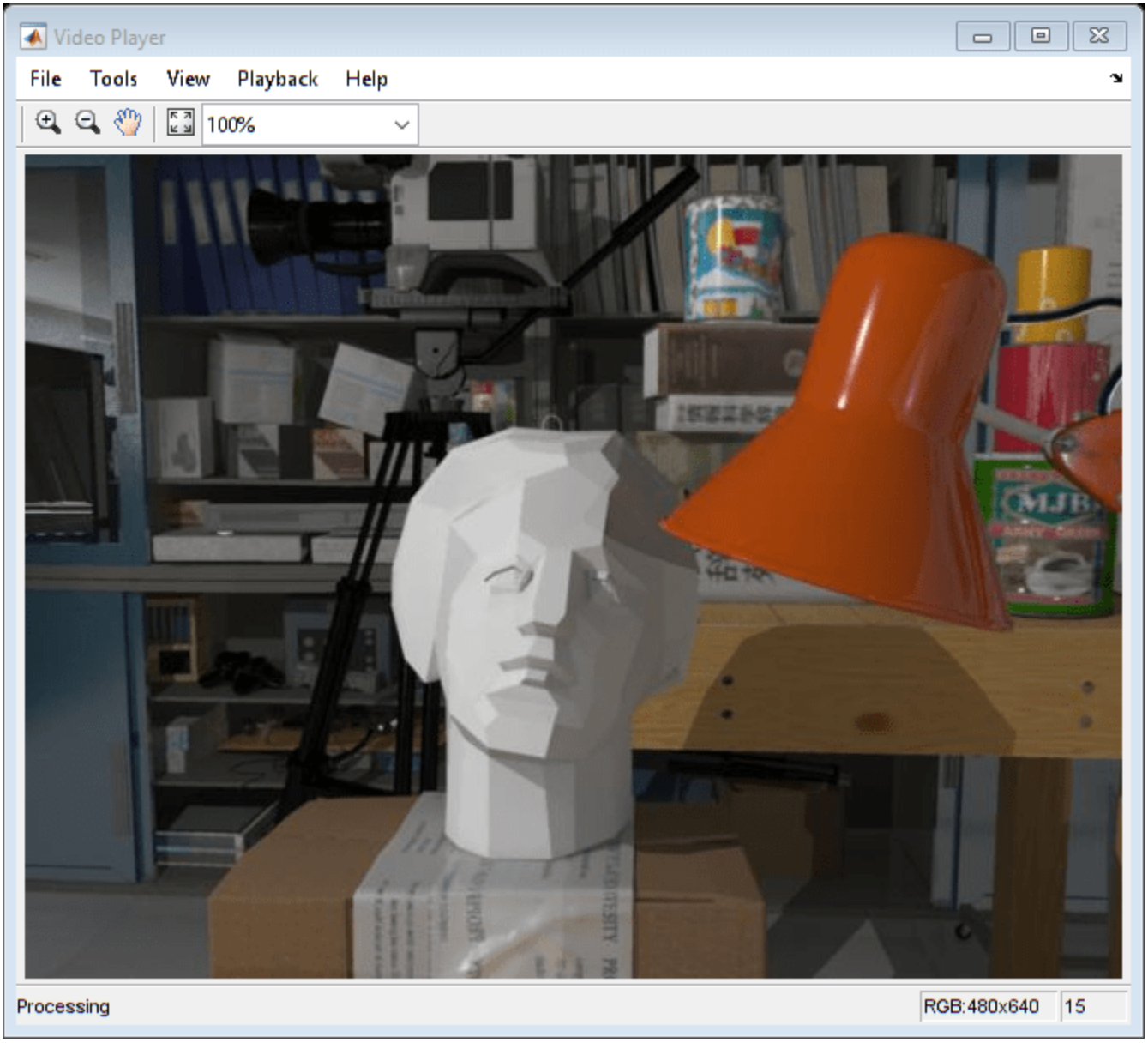
Find 3D-to-2D correspondences between world points triangulated from the previous two views and image points from the current view. Use `helperFindEpipolarInliers` to find the matches that satisfy the epipolar constraint, and then use `helperFind3Dto2DCorrespondences` to triangulate 3-D points from the 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(im2gray(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-axes 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
```





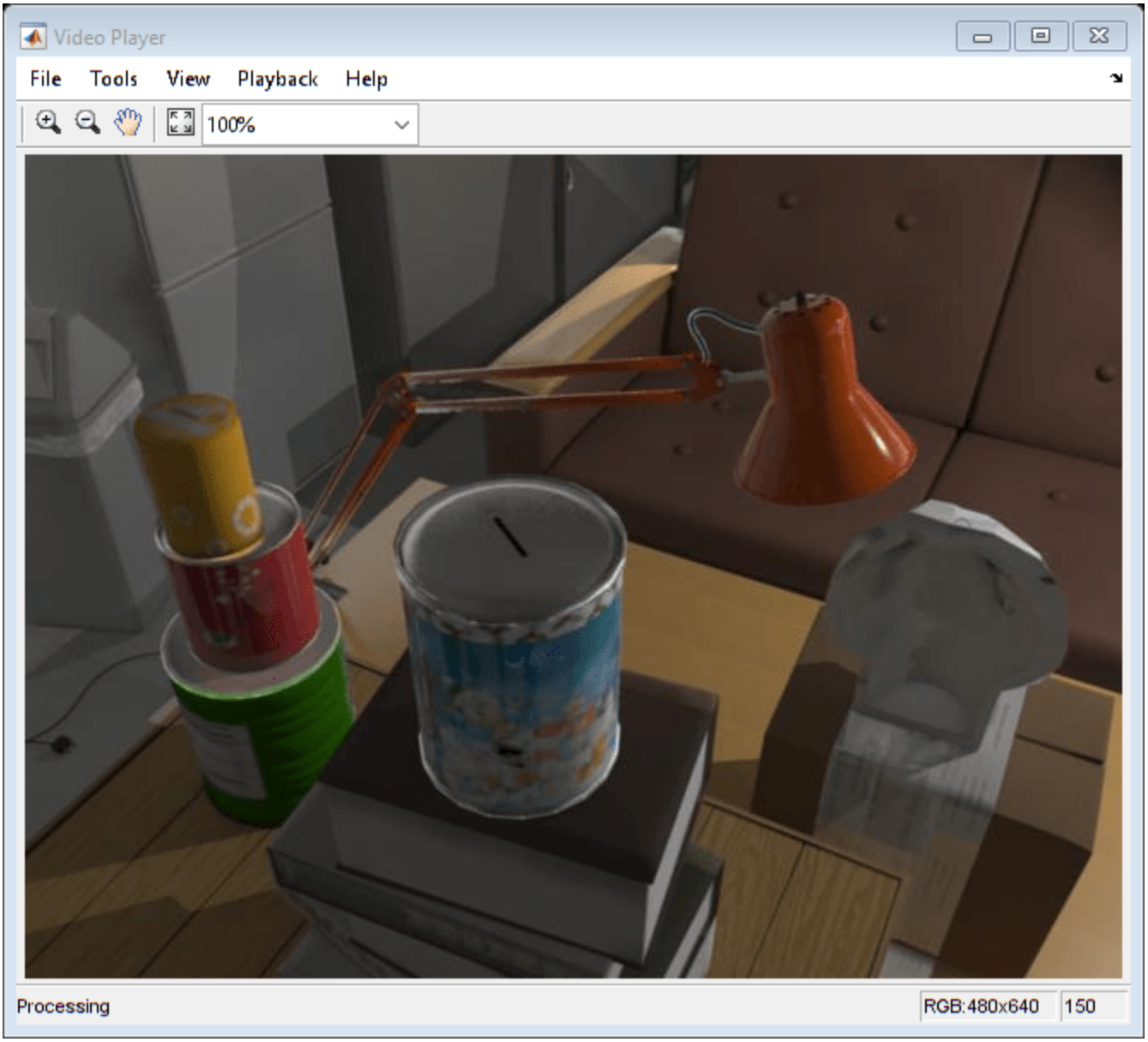
Estimate Remaining Camera Trajectory Using Windowed Bundle Adjustment

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.

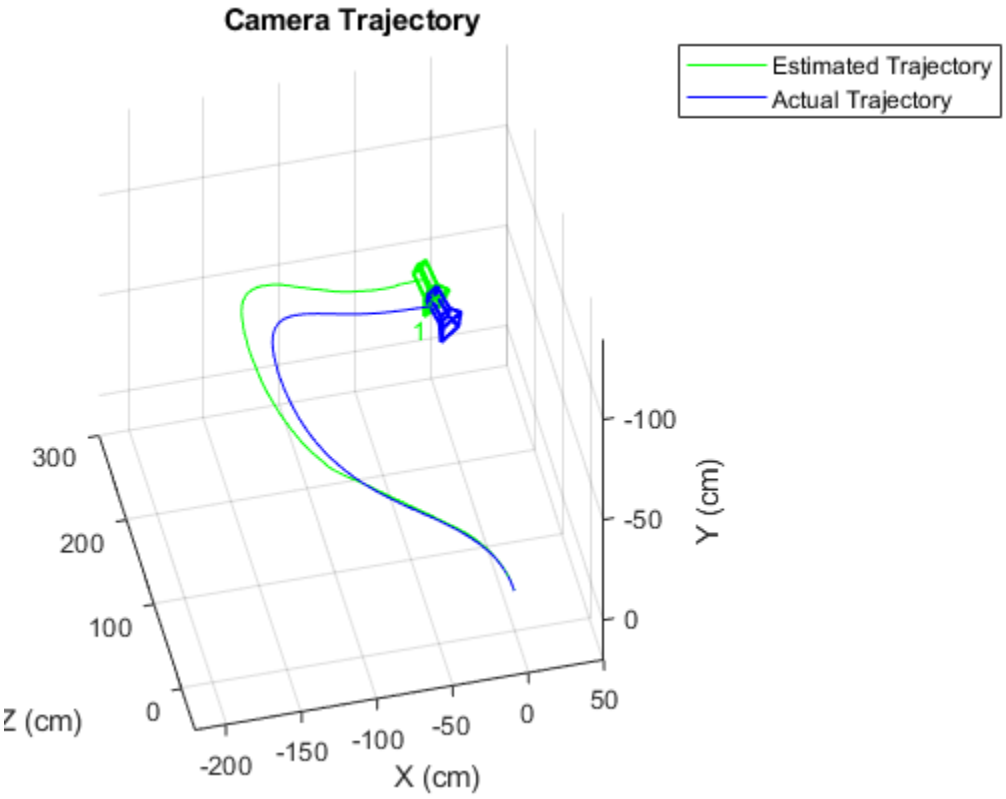
```
for viewId = 16:numel(images.Files)
    % Read and display the next image
    Irgb = readimage(images, viewId);
    step(player, Irgb);

    % Convert to gray scale and undistort.
```

```
I = undistortImage(im2gray(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

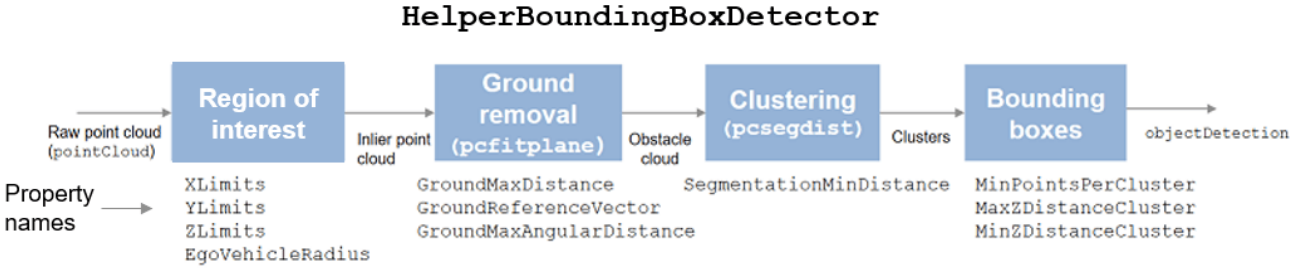
Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrians. For more details about segmentation of lidar 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/TrackVehiclesUsingLidarExampleDa
    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, and 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:

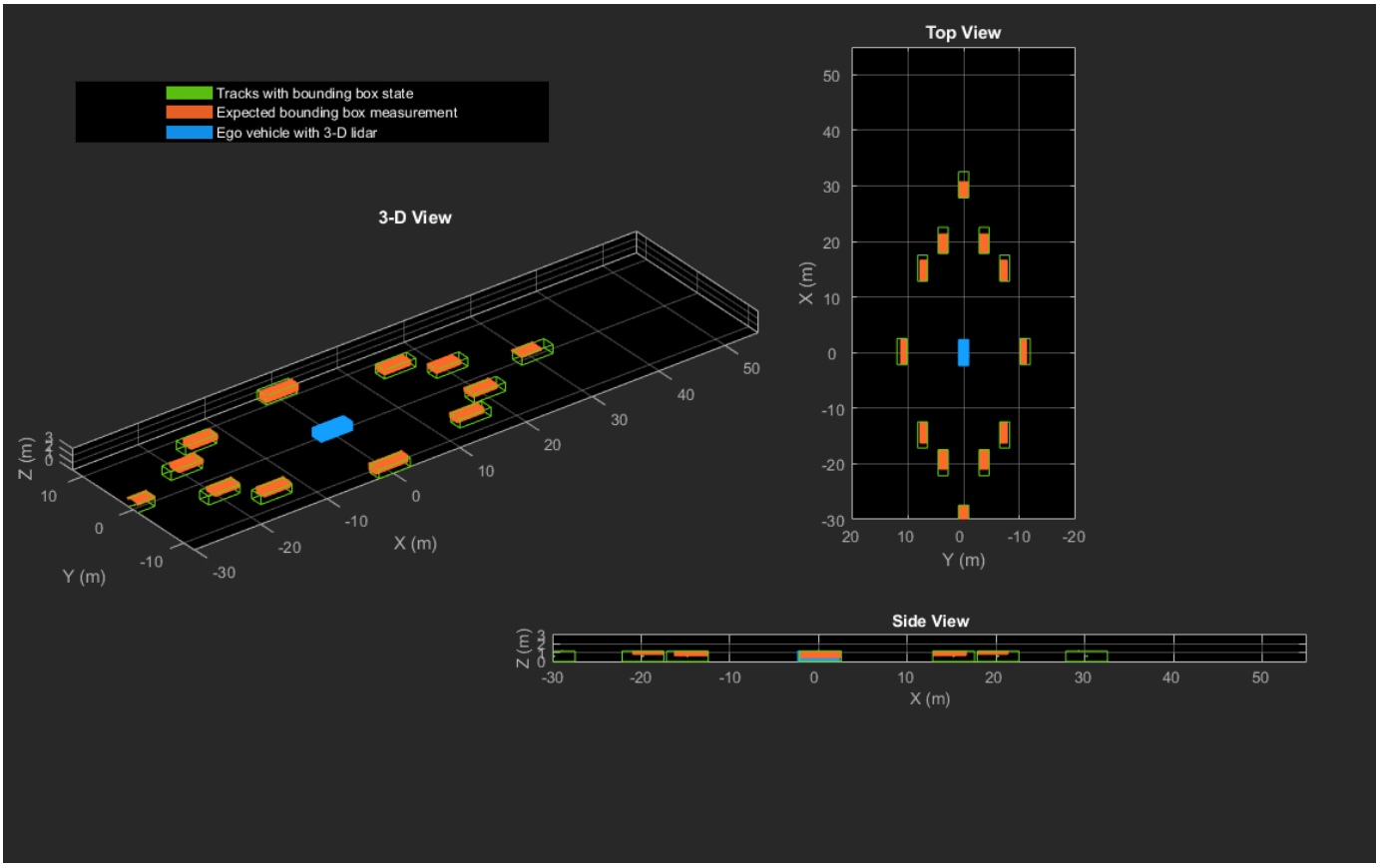
$$x_{cv} = [x\ \dot{x}\ y\ \dot{y}\ z\ \dot{z}\ \theta\ l\ w\ h]$$

$$x_{ct} = [x\ \dot{x}\ y\ \dot{y}\ \dot{\theta}\ z\ \dot{z}\ \theta\ l\ w\ h]$$

For information about their state transition, refer to the `helperConstvelCuboid` and `helperConstturnCuboid` functions used in this example.

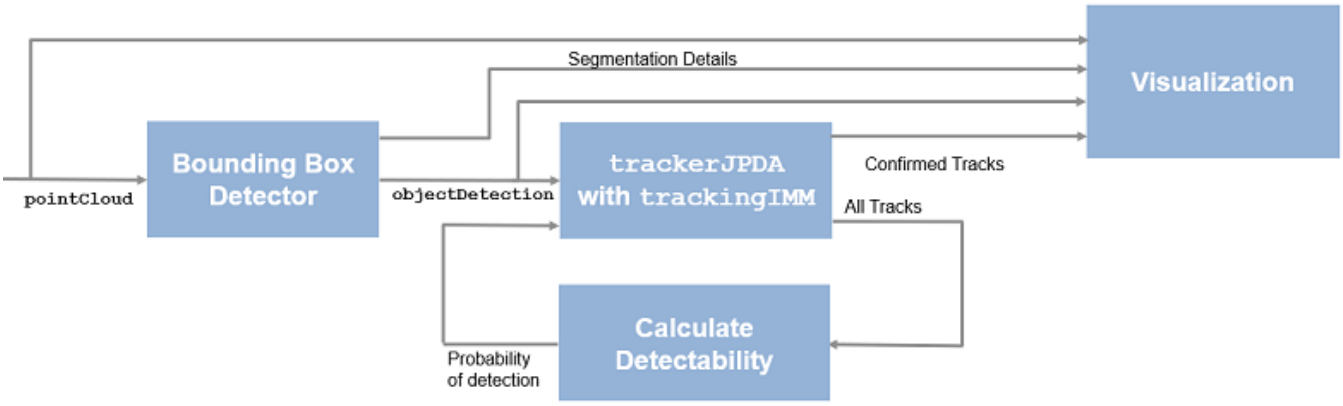
The `helperCvmeasCuboid` and `helperCtmeasCuboid` measurement models describe how the sensor perceives the constant velocity and constant turn-rate states respectively, and they return bounding box measurements. Because the state contains information about the size of the target, the measurement model 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.



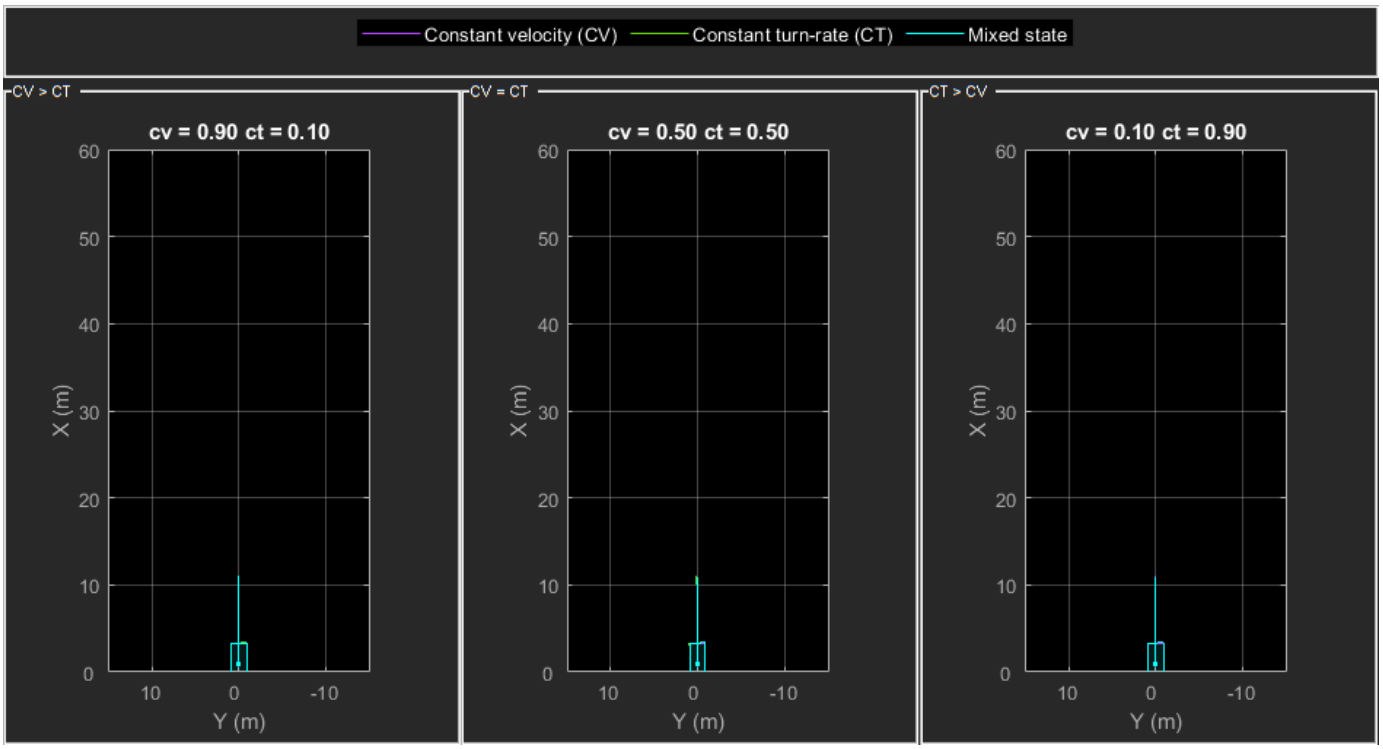
Set Up Tracker and Visualization

The image below shows the complete workflow to obtain a list of tracks from a pointCloud input.

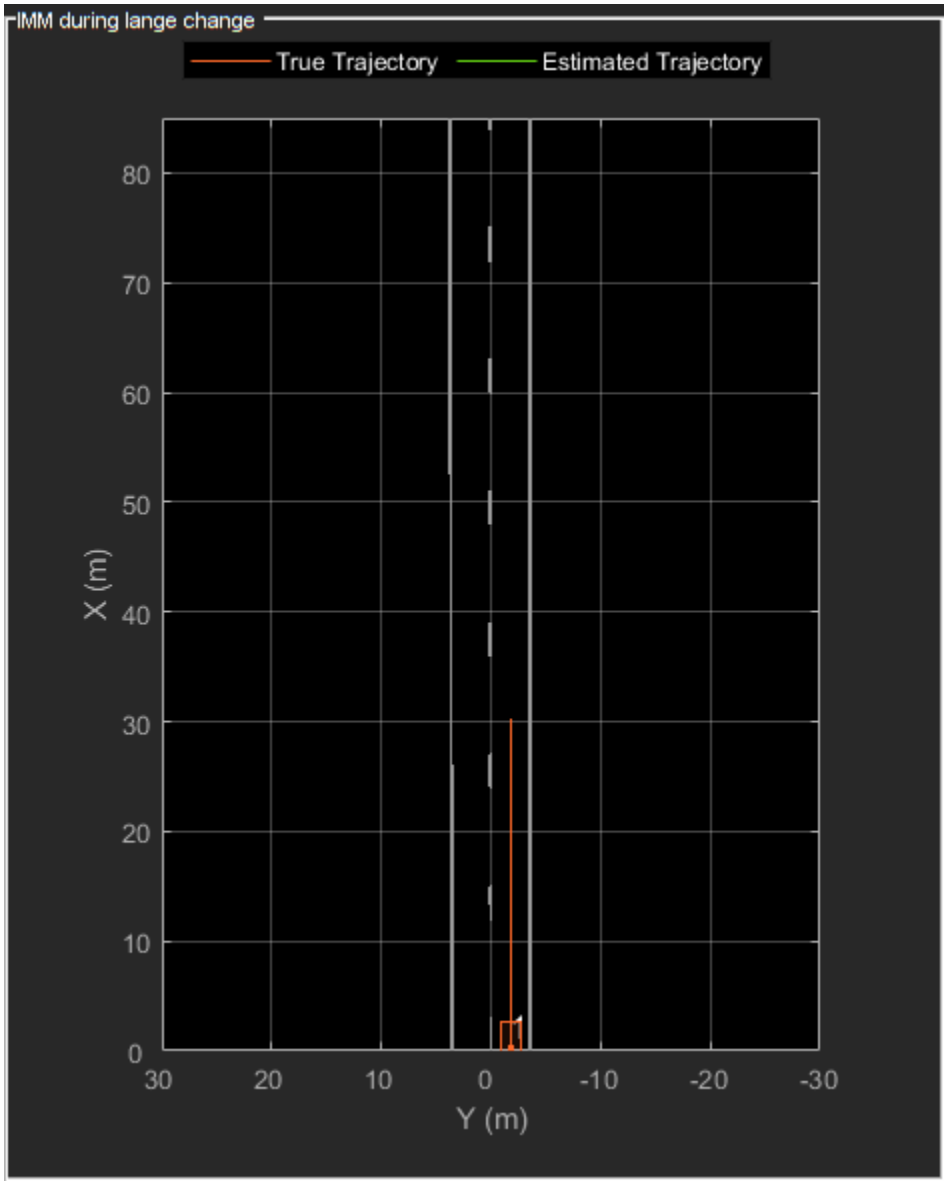


Now, set up the tracker and the visualization used in the example.

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.

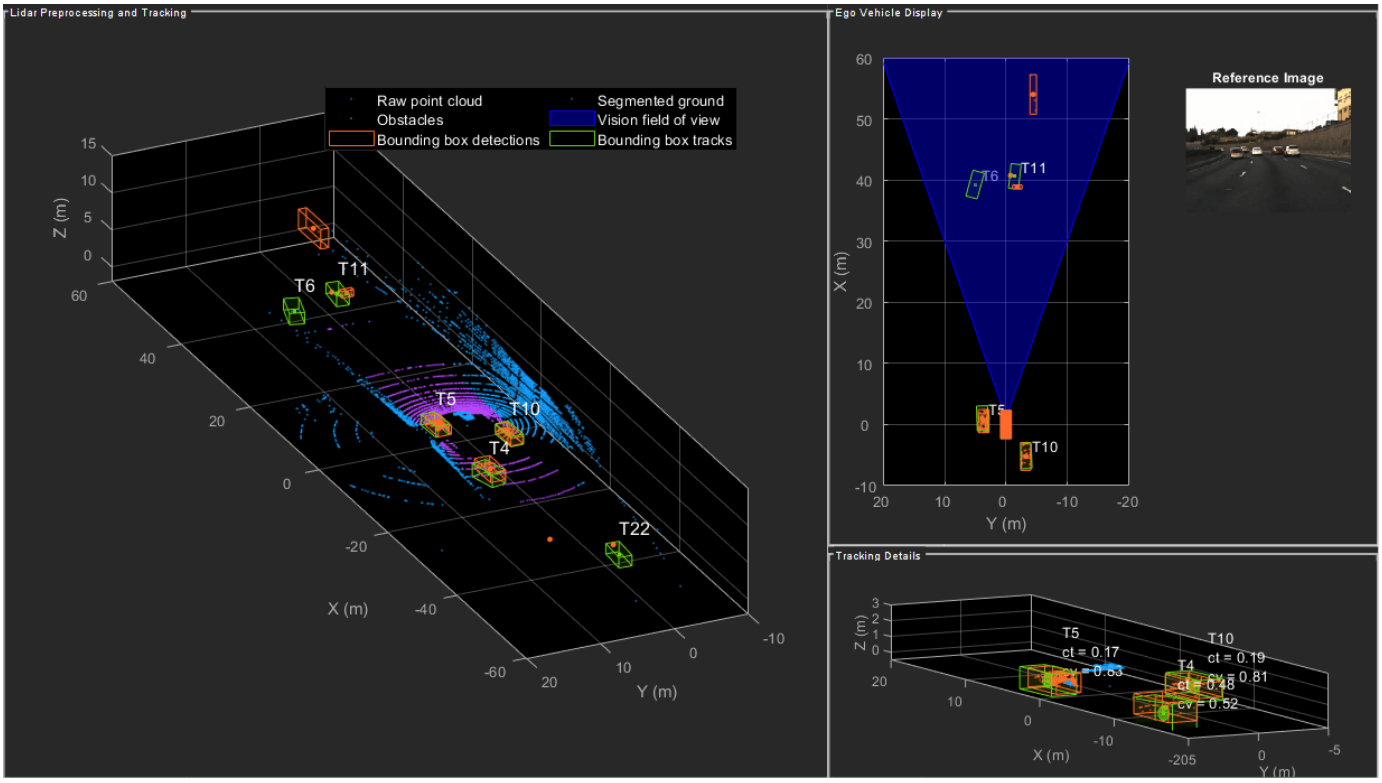
```
% Create display
displayObject = HelperLidarExampleDisplay(imageData{1},...
    'PositionIndex',[1 3 6],...
    'VelocityIndex',[2 4 7],...
    'DimensionIndex',[9 10 11],...
    'YawIndex',8,...
    'MovieName','',...  % Specify a movie name to record a movie.
    'RecordGIF',false); % Specify true 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
    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 wh
    % -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)));
```

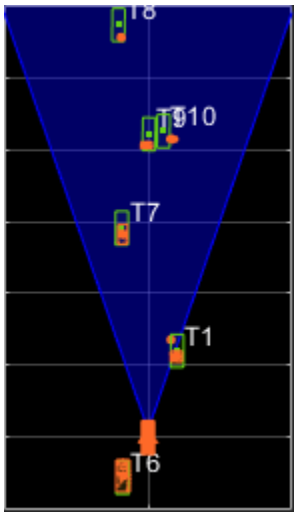
```
   1
```

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

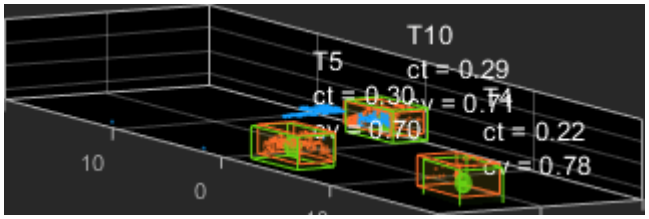
Now, analyze different events in the scenario and understand how the combination of lidar measurement model, joint probabilistic data association, and interacting multiple model filter, helps achieve a good estimation of the vehicle tracks.

Track Maintenance



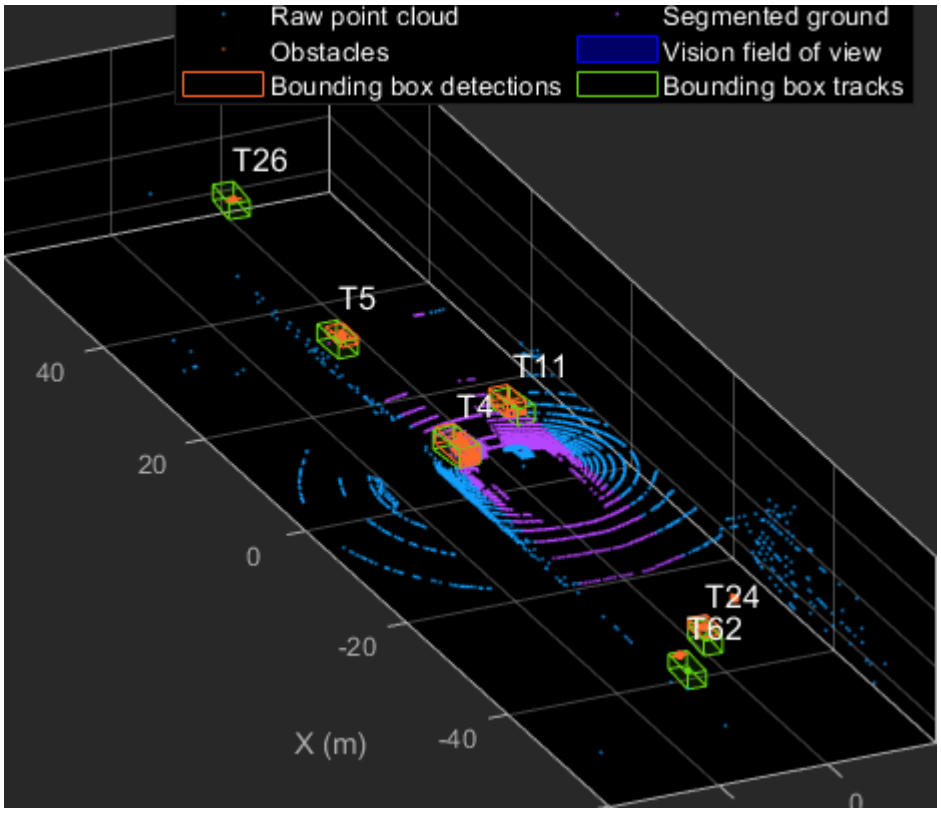
The animation above shows the simulation between time = 3 seconds and time = 16 seconds. Notice that tracks such as T10 and T6 maintain their IDs and trajectory during the time span. However, track T9 is lost because the tracked vehicle was missed (not detected) for a long time by the sensor. Also, notice 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 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
% uncertainity.
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 fot 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,currentPo
            % Crop point cloud
            [pcSurvived,survivedIndices,croppedIndices] = cropPointCloud(currentPointCloud,obj.XL
            % 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,minZDistanc
    % 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;
```

```
            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
    % pcfitplane.
    [~,groundIndices,outliers] = pcfitplane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDis
    ptCloudOut = select(ptCloudIn,outliers);
    obstacleIndices = currentIndices(outliers);
    groundIndices = currentIndices(groundIndices);
end

function [ptCloudOut,indices,croppedIndices] = cropPointCloud(ptCloudIn,xLim,yLim,zLim,egoVehicle
    % 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(curre

    % Use the same starting seed as MATLAB to reproduce results in SIL
    % simulation.
    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

    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
```

helperCalcDetectability

The function calculates 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
[~,~,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).

See Also

Related Examples

- “Track Vehicles Using Lidar: From Point Cloud to Track List” (Lidar Toolbox)
- “Detect, Classify, and Track Vehicles Using Lidar” (Lidar Toolbox)

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 17-62.

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.

```
ds = combine(imdsTrain,pxdsTrain);
tbl = countEachLabel(pxdsTrain)
```

```
tbl=2×3 table
         Name          PixelCount    ImagePixelCount
    _______________    __________    _______________

    {'triangle'  }         10326       2.048e+05
```

```
    {'background'}    1.9447e+05       2.048e+05
```

The majority of pixel labels are for 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(ds,layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:02 |       91.62% |       1.6825 |          0.0010 |
|      17 |          50 |       00:00:35 |       88.56% |       0.2393 |          0.0010 |
|      34 |         100 |       00:01:08 |       92.08% |       0.1672 |          0.0010 |
|      50 |         150 |       00:01:48 |       93.17% |       0.1472 |          0.0010 |
|      67 |         200 |       00:02:27 |       94.15% |       0.1313 |          0.0010 |
|      84 |         250 |       00:03:06 |       94.47% |       0.1167 |          0.0010 |
|     100 |         300 |       00:03:44 |       95.04% |       0.1100 |          0.0010 |
|=========================================================================================|
Training finished: Max epochs completed.
```

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);
```

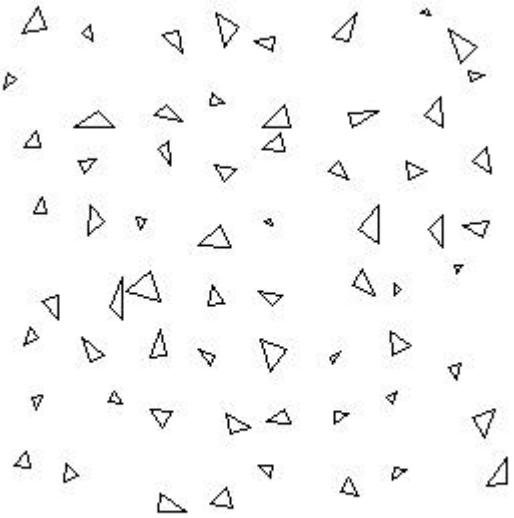
```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 100 images.
* Finalizing... Done.
* Data set metrics:

    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

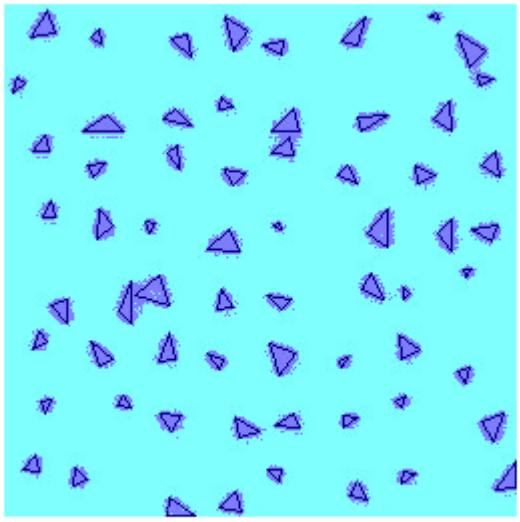
Read and display the test image `triangleTest.jpg`.

```
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 10-0]. The Tversky index TI_c between one image Y and the corresponding ground truth T is given by

$$\mathrm{TI}_c = \frac{\sum_{m=1}^{M} Y_{cm} T_{cm}}{\sum_{m=1}^{M} Y_{cm} T_{cm} + \alpha \sum_{m=1}^{M} Y_{cm} T_{\bar{c}m} + \beta \sum_{m=1}^{M} Y_{\bar{c}m} T_{cm}}$$

- 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

$$L = \sum_{c=1}^{C} 1 - \mathrm{TI}_c$$

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: 1.3687 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)];
```

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 datastore `combine`.

```
ds = combine(imds,pxds);
```

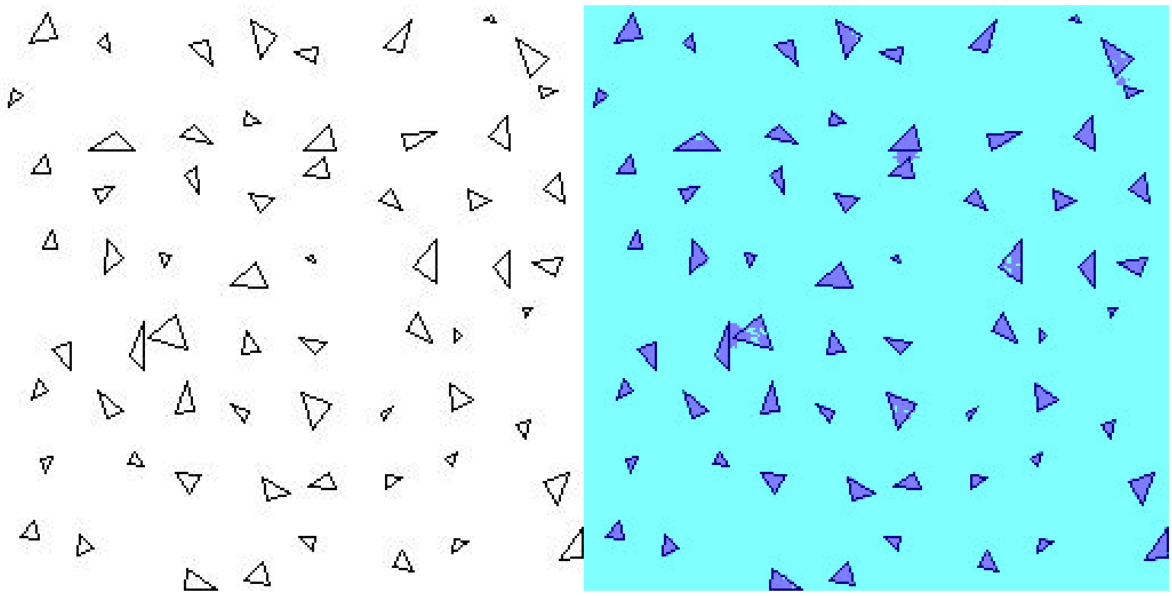
Set the training options and train the network.

```
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  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:05 |       50.32% |       1.2933 |          0.0010 |
|      13 |          50 |       00:01:02 |       98.82% |       0.0985 |          0.0010 |
|      25 |         100 |       00:01:50 |       99.32% |       0.0545 |          0.0005 |
|      38 |         150 |       00:02:49 |       99.37% |       0.0472 |          0.0005 |
|      50 |         200 |       00:03:57 |       99.48% |       0.0401 |          0.0003 |
|      63 |         250 |       00:05:05 |       99.48% |       0.0379 |          0.0001 |
|      75 |         300 |       00:06:16 |       99.54% |       0.0348 |          0.0001 |
|      88 |         350 |       00:07:53 |       99.51% |       0.0351 |      6.2500e-05 |
|     100 |         400 |       00:09:16 |       99.56% |       0.0330 |      6.2500e-05 |
|========================================================================================|
Training finished: Max epochs completed.
```

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.

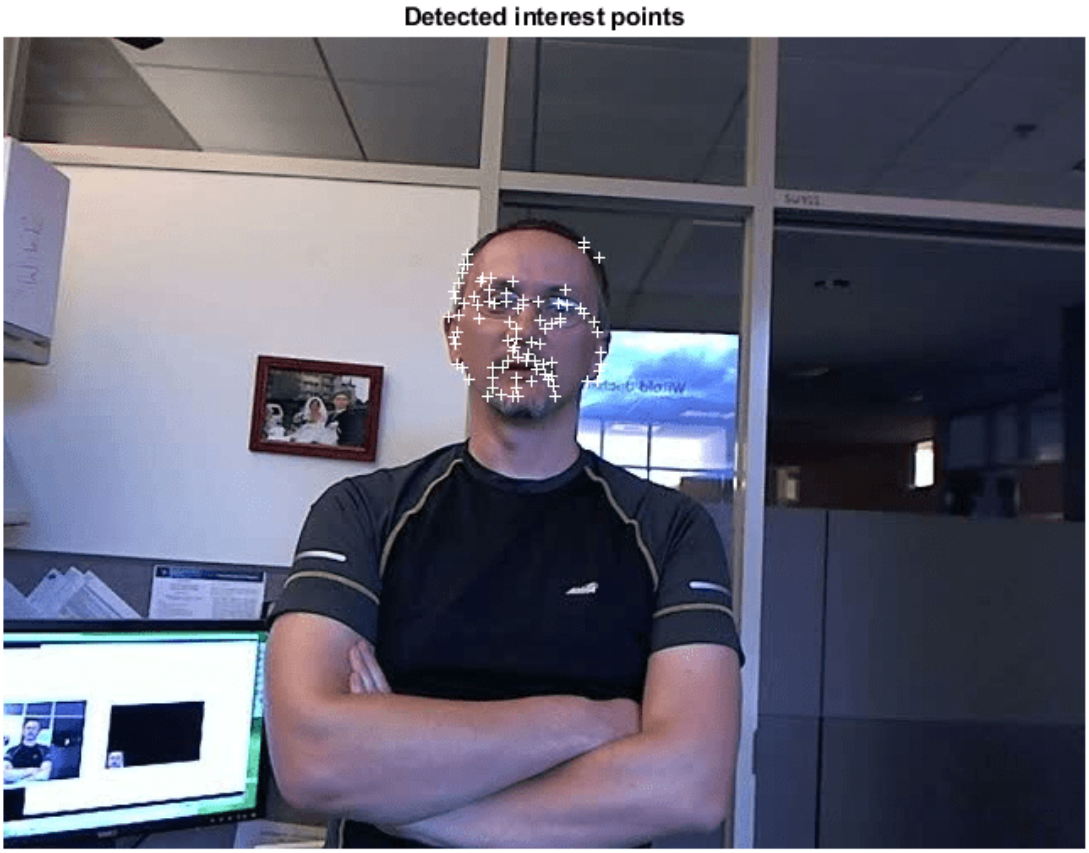
```
objectImage = insertShape(objectFrame,'Rectangle',objectRegion,'Color','red');
figure;
imshow(objectImage);
title('Red box shows object region');
```

Detect interest points in the object region.

```
points = detectMinEigenFeatures(im2gray(objectFrame),'ROI',objectRegion);
```

Display the detected points.

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



Create a tracker object.

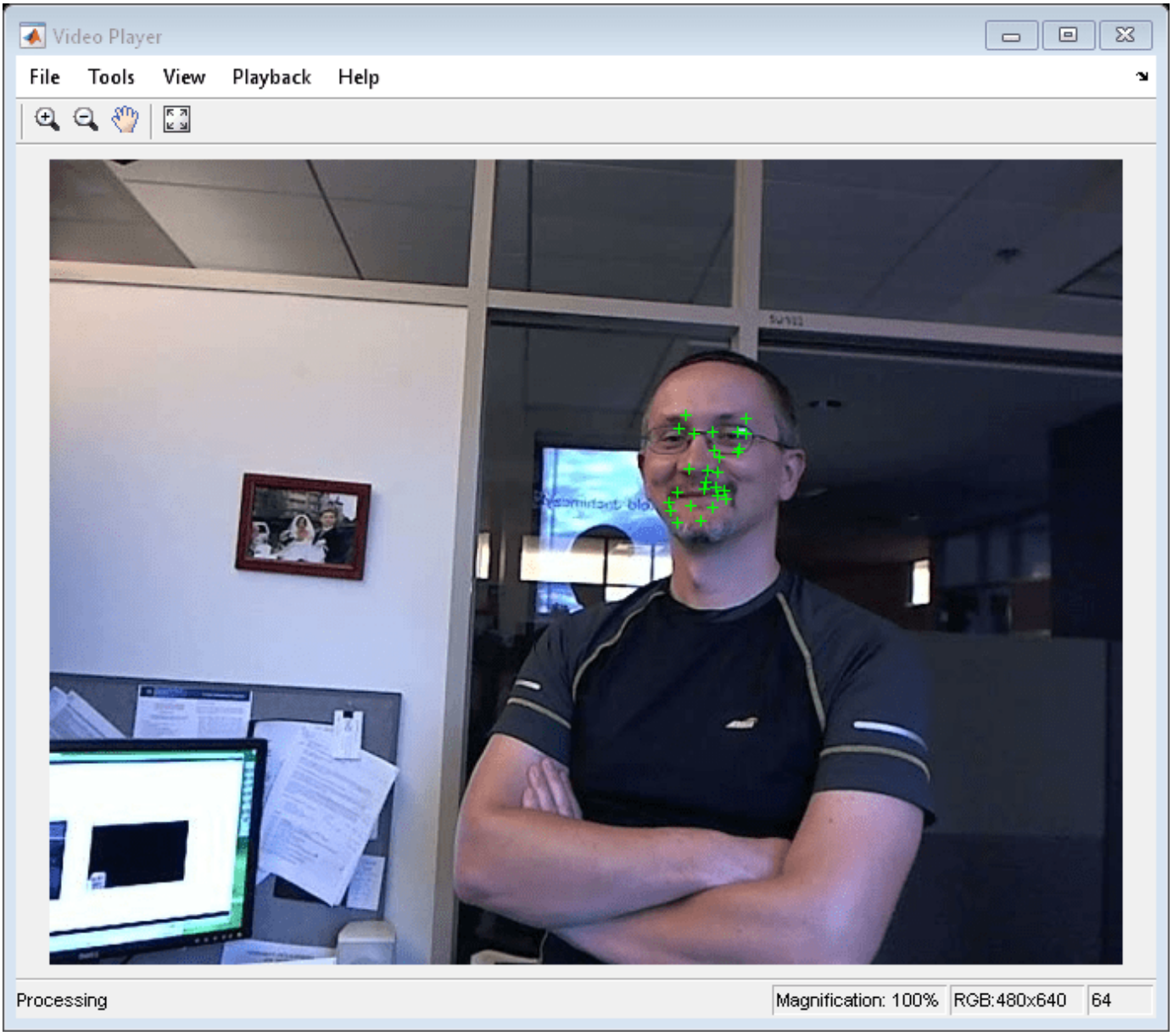
```
tracker = vision.PointTracker('MaxBidirectionalError',1);
```

Initialize the tracker.

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

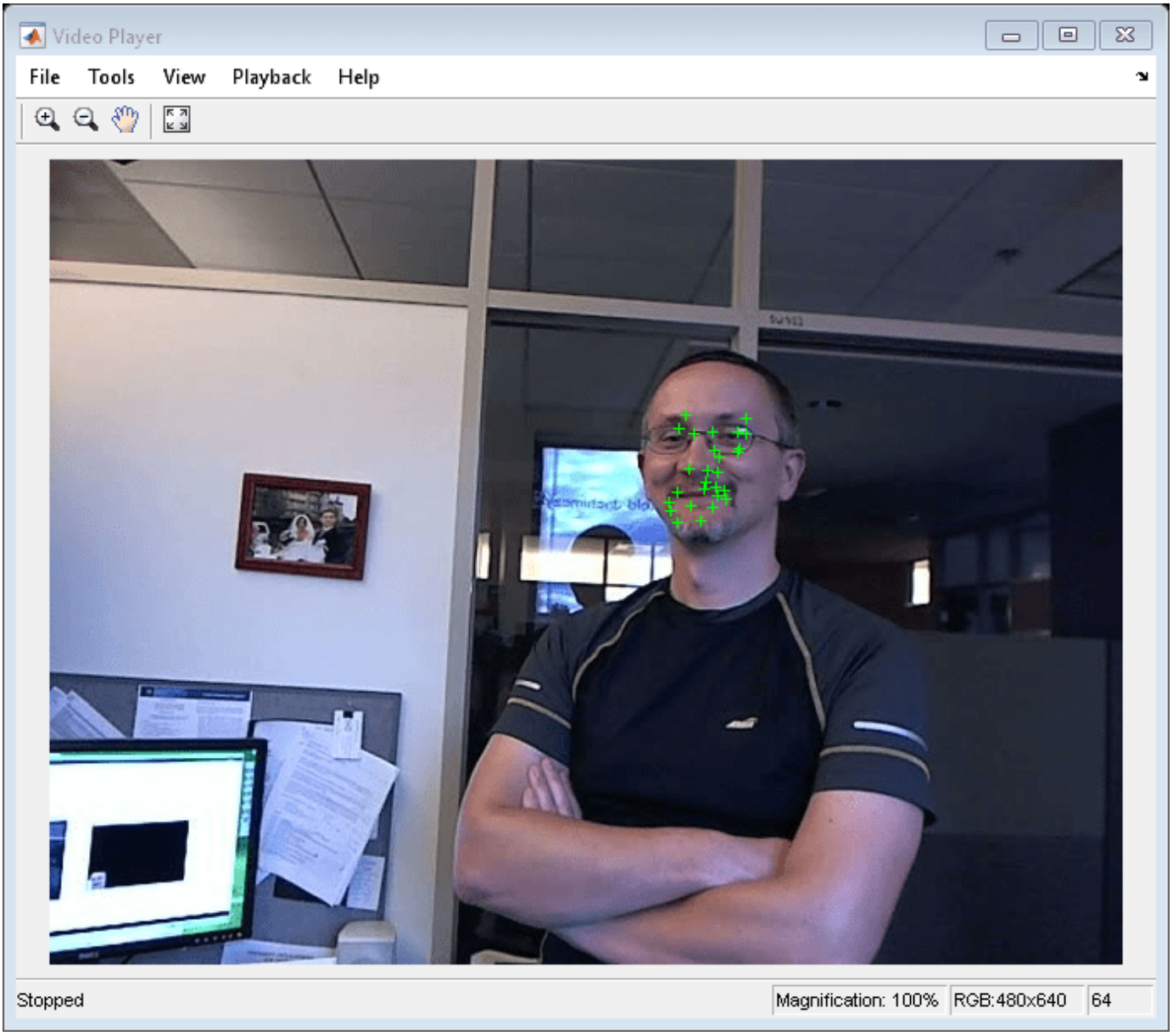
Read, track, display points, and results in each video frame.

```
while hasFrame(videoReader)
      frame = readFrame(videoReader);
      [points,validity] = tracker(frame);
      out = insertMarker(frame,points(validity, :),'+');
      videoPlayer(out);
end
```



Release the video player.

```
release(videoPlayer);
```


Video Player
File
Tools
View
Playback
Help
Stopped
Magnification: 100%
RGB:480x640
64

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, reprojectionMatrix] = 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(im2gray(J1),im2gray(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,reprojectionMatrix);
```

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','circleGrid','stereo');
leftImages = imageDatastore(fullfile(imageDir,'left'));
rightImages = imageDatastore(fullfile(imageDir,'right'));
leftImageFileNames = leftImages.Files(2:2:10);
rightImageFileNames = rightImages.Files(2:2:10);
```

Define the circle grid pattern dimensions, and detect the pattern in the images.

```
patternDims = [4 11];
imagePoints = detectCircleGridPoints(leftImageFileNames,...
        rightImageFileNames,patternDims);
```

Specify the world coordinates for the circle grid keypoints. Center distance is in millimeters.

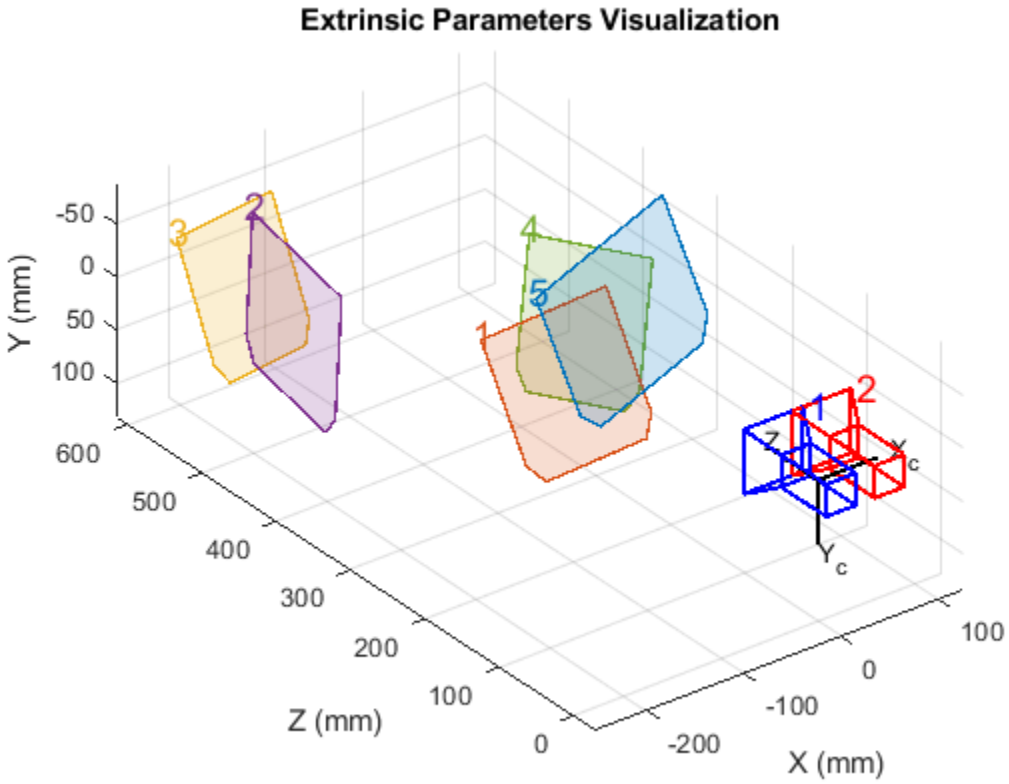
```
centerDistance = 36.5;
worldPoints = generateCircleGridPoints(patternDims,centerDistance);
```

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);
```

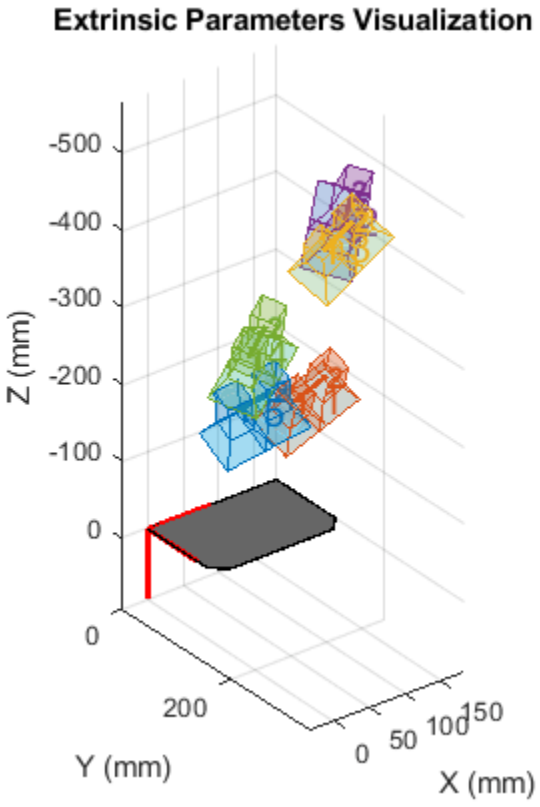
Visualize pattern locations.

```
figure
showExtrinsics(cameraParams)
```



Visualize camera locations.

```
figure
showExtrinsics(cameraParams,'patternCentric')
```


Extrinsic Parameters Visualization
-500
-400
-300
-200
-100
0
Z (mm)
0
200
Y (mm)
0
50
100
150
X (mm)

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',radialDisto
```

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)');
```

Structure from Motion and Visual SLAM

- “Choose SLAM Workflow Based on Sensor Data” on page 11-2
- “Implement Visual SLAM in MATLAB” on page 11-8

Choose SLAM Workflow Based on Sensor Data

You can use Computer Vision Toolbox, Navigation Toolbox™, and Lidar Toolbox for Simultaneous Localization and Mapping (SLAM). SLAM is widely used in applications including automated driving, robotics, and unmanned aerial vehicles (UAV). To learn more about SLAM, see What is SLAM?.

Choose SLAM Workflow

To choose the right SLAM workflow for your application, consider what type of sensor data you are collecting. MATLAB support SLAM workflows that use images from a monocular or stereo camera system, or point cloud data including 2-D and 3-D lidar data.

This table summarizes the key features available for SLAM.

Sensor Data	Features	Topics	Examples	Toolbox	Code Generation
Monocular images	• Feature detection, extraction, and matching • Triangulation and bundle adjustment • Data management for key frames and map points • Loop closure detection using bag of features • Similarity pose graph optimization	• “Implement Visual SLAM in MATLAB” on page 11-8 • “Visual Simultaneous Localization and Mapping (vSLAM)”	• “Monocular Visual Simultaneous Localization and Mapping” on page 1-72 • “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” on page 1-24	• Computer Vision Toolbox	• Computer Vision Toolbox C/C++ Code Generation • Computer Vision Toolbox GPU Code Generation

Sensor Data	Features	Topics	Examples	Toolbox	Code Generation
Stereo images	• Stereo image rectification • Feature detection, extraction, and matching • Reconstruction from disparity, triangulation, and bundle adjustment • Data management for key frames and map points • Loop closure detection using bag of features • Pose graph optimization	• “Visual Simultaneous Localization and Mapping (vSLAM)”	• “Stereo Visual Simultaneous Localization and Mapping” on page 1-102 • “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” on page 1-24	• Computer Vision Toolbox	• Computer Vision Toolbox C/C++ Code Generation • Computer Vision Toolbox GPU Code Generation

Sensor Data	Features	Topics	Examples	Toolbox	Code Generation
RGB-D images	• Feature detection, extraction, and matching • Reconstruction from depth images, triangulation, and bundle adjustment • Data management for key frames and map points • Loop closure detection using bag of features • Pose graph optimization	• "Visual Simultaneous Localization and Mapping (vSLAM)"	• "Visual SLAM with an RGB-D Camera" on page 1-2	• Computer Vision Toolbox	• Computer Vision Toolbox C/C++ Code Generation • Computer Vision Toolbox GPU Code Generation

Sensor Data	Features	Topics	Examples	Toolbox	Code Generation
2-D lidar scans	• Occupancy map building • Vehicle pose estimation • Pose graph optimization • SLAM algorithm tuning • SLAM Map Builder app	• “Mapping” (Navigation Toolbox) • “SLAM” (Navigation Toolbox) • “Navigation and Mapping” (Lidar Toolbox)	• “Build Map from 2-D Lidar Scans Using SLAM” (Lidar Toolbox) • “Implement Simultaneous Localization And Mapping (SLAM) with Lidar Scans” (Navigation Toolbox) • “Implement Online Simultaneous Localization And Mapping (SLAM) with Lidar Scans” (Navigation Toolbox)	• Navigation Toolbox • Lidar Toolbox	• Navigation Toolbox C/C++ Code Generation • Navigation Toolbox GPU Code Generation • Lidar Toolbox C/C++ Code Generation

Sensor Data	Features	Topics	Examples	Toolbox	Code Generation
Point cloud data	• Point cloud processing • Registration • Data management for map building • Loop closure detection with global features • Pose graph optimization • Localization in a known map	• “Register Point Clouds and Create Maps” • “Implement Point Cloud SLAM in MATLAB” on page 12-4	• “Build a Map from Lidar Data Using SLAM” on page 5-55 • “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment ” on page 5-2 • “Build Occupancy Map from 3-D Lidar Data using SLAM” (Automated Driving Toolbox)	• Computer Vision Toolbox	• Computer Vision Toolbox C/C ++ Code Generation • Computer Vision Toolbox GPU Code Generation

Sensor Data	Features	Topics	Examples	Toolbox	Code Generation
3-D lidar scans	Feature-based: • Registration • Loop closure detection • Localization in a known map	• "Navigation and Mapping" (Lidar Toolbox)	• "Aerial Lidar SLAM Using FPFH Descriptors" (Lidar Toolbox) • "Build Map and Localize Using Segment Matching" (Lidar Toolbox) • "Feature-Based Map Building from Lidar Data" (Lidar Toolbox) • "Build a Map with Lidar Odometry and Mapping (LOAM) Using Unreal Engine Simulation" (Lidar Toolbox)	• Lidar Toolbox	• Lidar Toolbox C/C++ Code Generation

Implement Visual SLAM in MATLAB

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 visual SLAM include augmented reality, robotics, and autonomous driving. For a general description on why SLAM matters and how it works for different applications, see What is SLAM?

Visual SLAM algorithms are broadly classified into two categories, depending on how they estimate the camera motion. The indirect, feature-based method uses feature points of images to minimize the reprojection error. The direct method uses the overall brightness of images to minimize the photometric error. The Computer Vision Toolbox algorithms provide functions for performing feature-based visual SLAM. The workflow consists of map initialization, tracking, local mapping, loop detection, and drift correction.

Note The workflow described in this overview applies to images taken by a pinhole camera. To use the visual SLAM workflow with images taken by a fisheye camera, convert the fisheye camera into a virtual pinhole camera using the `undistortFisheyeImage` function.

Terms Used in Visual SLAM

Visual SLAM literature uses these common terms:

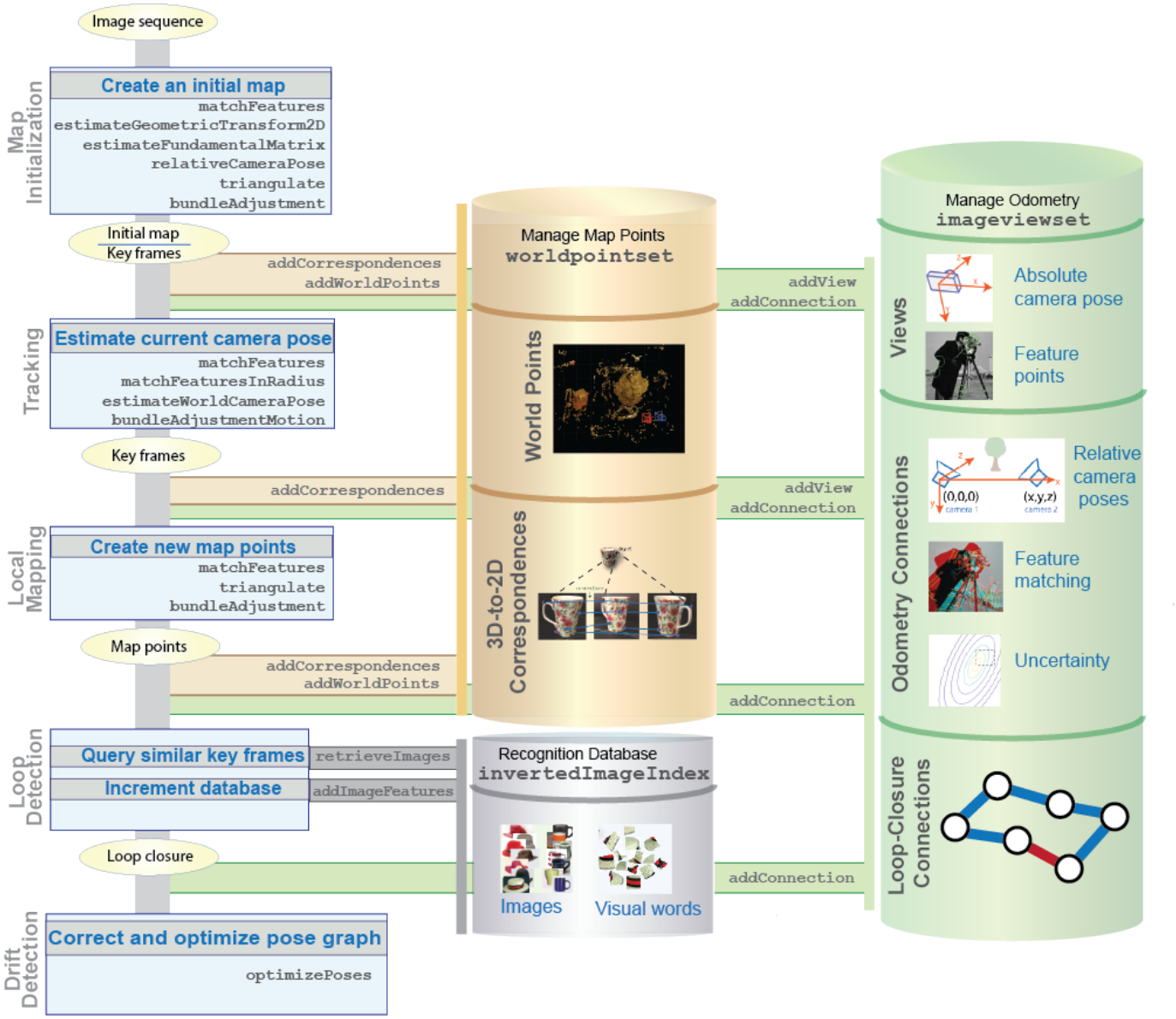
- Key Frames — A subset of video frames that contain cues for localization and tracking. Two consecutive key frames usually indicate a large visual change caused by a camera movement.
- Map Points — A list of 3-D world points that represent the map of the environment reconstructed from the key frames.
- Covisibility Graph — A graph of key frames 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.
- Recognition Database — A database that stores the visual word-to-image mapping based on the input bag of features. Determine whether a place has been visited in the past by searching the database for an image that is visually similar to the query image.

Typical Feature-based Visual SLAM Workflow

To construct a feature-based visual SLAM pipeline on a sequence of images, follow these steps:

1. Initialize Map — Initialize the map of 3-D points from two image frames. Compute the 3-D points and relative camera pose by using triangulation based on 2-D feature correspondences.
2. Track Features — For each new frame, estimate the camera pose by matching features in the current frame to features in the last key frame.
3. Create Local Map — If you identify the current frame as a key frame, create a new 3-D map of points. Use bundle adjustment to refine the camera pose and 3-D points.
4. Detect Loops — Detect loops for each key frame by comparing the current frame to all previous key frames using the bag-of-features approach.
5. Correct Drift — Optimize the pose graph to correct the drift in the camera poses of all the key frames.

The figure illustrates a typical feature-based visual SLAM workflow. It also shows the points at which data is stored or retrieved from objects that manage the data.



Key Frame and Map Data Management

Use the view set, point set, and transformation objects to manage key frames and map data.

- Use the `imageviewset` object to manage data associated with the odometry and mapping process. The object contains data as a set of views and pairwise connections between views. The object can also be used to build and update a pose graph.

  - Each view consists of the absolute camera pose and the feature points extracted from the image. Each view, with its unique identifier (view ID), within the view set forms a node of the pose graph.
  - Each connection stores information that links one view to another view. The connection includes the indices of matched features between the views, the relative transformation between the connected views, and the uncertainty in computing the measurement. Each connection forms an edge in the pose graph.
  - Use a `rigid3d` object input with `imageviewset` to store the absolute camera poses and relative camera poses of odometry edges. Use an `affine3d` object input with `imageviewset` to store the relative camera poses of loop-closure edges.
- Use the `worldpointset` object to store correspondences between 3-D map points and 2-D image points across camera views.
  - The `WorldPoints` property of `worldpointset` stores the 3-D locations of map points.
  - The `Correspondence` property of `worldpointset` stores the view IDs of the key frames that observe the map points.

Map Initialization

To initialize mapping, you must match features between two images, estimate the relative camera pose, and triangulate initial 3-D world points. This workflow commonly uses the Speeded-Up Robust Features (SURF) and Oriented FAST and Rotated BRIEF (ORB) features point features. The map initialization workflow consists of a detecting, extracting, and matching features, and then finding a relative camera pose estimate, finding the 3-D locations of matched features, and refining the initial map. Finally, store the resulting key frames and mapped points in an image view set and a world point set, respectively.

Workflow	Function	Description
1. Detect	`detectSURFFeatures`	Detect SURF features and return a `SURFPoints` object.
	`detectORBFeatures`	Detect ORB features and return an `ORBPoints` object.
	`detectSIFTFeatures`	Detect SIFT features and return a `SIFTPoints` object.
2. Extract	`extractFeatures`	Extract feature vectors and their corresponding locations in a binary or intensity image.
3. Match	`matchFeatures`	Obtain the indices of the matching features between two feature sets.
4. Estimate relative camera pose from matched feature points	`estimateGeometricTransform2D`	Compute a homography from matching point pairs.
	`estimateFundamentalMatrix`	Estimate the fundamental matrix from matching point pairs.

Workflow	Function	Description
	`relativeCameraPose`	Compute the relative camera poses, represented as a `rigid3d` object, based on a homography or a fundamental matrix. The location can only be computed up to scale, so the distance between two cameras is set to 1.
5. Find 3-D locations of the matched feature points	`triangulate`	Find the 3-D locations of matching pairs of undistorted image points.
6. Refine initial map	`bundleAdjustment`	Refine 3-D map points and camera poses that minimize reprojection errors.
7. Manage data for initial map and key frames	`addView`	Add the two views formed by the feature points and their absolute poses to the `imageviewset` object.
	`addConnection`	Add the odometry edge defined by the connection between successive key views, formed by the relative pose transformation between the cameras, to the `imageviewset` object.
	`addWorldPoints`	Add the initial map points to the `worldpointset` object.
	`addCorrespondences`	Add the 3-D to 2-D projection correspondences between the key frames and the map points to the `worldpointset` object.

Tracking

The tracking workflow uses every frame to determine when to insert a new key frame. Use these steps and functions for the tracking workflow.

Workflow	Function	Description
Match extracted features	`matchFeatures`	Match extracted features from the current frame with features in the last key frame that have known 3-D locations.
Estimate camera pose	`estimateWorldCameraPose`	Estimate the current camera pose.
Project map points	`worldToImage`	Project the map points observed by the last key frame into the current frame.

Workflow	Function	Description
Search for feature correspondences	`matchFeaturesInRadius`	Search for feature correspondences within spatial constraints.
Refine camera pose	`bundleAdjustmentMotion`	Refine the camera pose with 3-D to 2-D correspondence by performing a motion-only bundle adjustment.

Identify local map points	`findWorldPointsInView`
	
	`findWorldPointsInTracks`
		Identify points in the view and points that correspond to point tracks.

Search for more feature correspondences	`matchFeaturesInRadius`	Search for more feature correspondences in the current frame, which contains projected local map points.
Refine camera pose	`bundleAdjustmentMotion`	Refine the camera pose with 3-D to 2-D correspondence by performing a motion-only bundle adjustment.

Store new key frame	`addView`
	
	`addConnection`
		If you determine that the current frame is a new key frame, add it and its connections to covisible key frames to the `imageviewset`.

Feature matching is critical in the tracking workflow. Use the `matchFeaturesInRadius` function to return more putative matches when an estimation of the positions of matched feature points is available. The two match feature functions used in the workflow are:

- `matchFeatures` — Returns the indices of the matching features in the two input feature sets.
- `matchFeaturesInRadius` — Returns the indices of the matching features, which satisfy spatial constraints, in the two input feature sets.

To get a greater number of matched feature pairs, increase the values for the `MatchThreshhold` and `MaxRatio` name-value arguments of the `matchFeatures` and `matchFeaturesInRadius` functions. The outliers pairs can be discarded after performing bundle adjustment in the local mapping step.

Local Mapping

Perform local mapping for every key frame. Follow these steps to create new map points.

Workflow	Function	Description
Connect key frames	`connectedViews`	Find the covisible key frames of the current key frame.

Workflow	Function	Description
Search for matches in connected key frames	`matchFeatures`	For each unmatched feature point in the current key frame, use the `matchFeatures` function to search for a match with other unmatched points in the covisible key frames.
Compute location for new matches	`triangulate`	Compute the 3-D locations of the matched feature points.
Store new map points	`addWorldPoints`	Add the new map points to the `worldpointset` object.
Store 3-D to 2-D correspondences	`addCorrespondences`	Add new 3-D to 2-D correspondences to the `worldpointset` object.
Update odometry connection	`updateConnection`	Update the connection between the current key frame and its covisible frames with more feature matches.
Refine pose	`bundleAdjustment`	Refine the pose of the current key frame, the poses of covisible key frames, and all the map points observed in these key frames. For improved performance, only include strongly connected, covisible key frames in the refinement process. Use the `minNumMatches` argument of the `connectedViews` function to select strongly-connected covisible key frames.
Remove outliers	`removeWorldPoints`	Remove outlier map points with large reprojection errors from the `worldpointset` object. The associated 3-D to 2-D correspondences are removed automatically.

This table compares the camera poses, map points, and number of cameras for each of the bundle adjustment functions used in 3-D reconstruction.

Function	Camera Poses	Map Points	Number of Cameras
`bundleAdjustment`	Optimized	Optimized	Multiple
`bundleAdjustmentMotion`	Optimized	Fixed	One

Function	Camera Poses	Map Points	Number of Cameras
`bundleAdjustmentStructure`	Fixed	Optimized	Multiple

Loop Detection

Due to an accumulation of errors, using visual odometry alone can lead to drift. These errors can result in severe inaccuracies over long distances. Using graph-based SLAM helps to correct the drift. To do this, detect loop closures by finding a previously visited location. A common approach is to use this bag-of-features workflow:

Workflow	Function	Description
Construct bag of visual words	`bagOfFeatures`	Construct a bag of visual words for place recognition.
Create recognition database	`indexImages`	Create a recognition database, `invertedImageIndex`, to map visual words to images.
Identify loop closure candidates	`retrieveImages`	Search for images that are similar to the current key frame. Identify consecutive images as loop closure candidates if they are similar to the current frame. Otherwise, add the current key frame to the recognition database.
Compute relative camera pose for loop closure candidates	`estimateGeometricTransform3D`	Compute the relative camera pose between the candidate key frame and the current key frame, for each loop closure candidate
Close loop	`addConnection`	Close the loop by adding a loop closure edge with the relative camera pose to the `imageviewset` object.

Drift Correction

The `imageviewset` 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 closures are added. The `optimizePoses` function returns an `imageviewset` 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 Navigation Toolbox to optimize the modified pose graph, and then use the `updateView` function to update the camera poses in the view set.

Visualization

To develop the visual SLAM system, you can use the following visualization functions.

Function	Description
`imshow`	Display an image
`showMatchedFeatures`	Display matched feature points in two images
`plot`	Plot image view set views and connections
`plotCamera`	Plot a camera in 3-D coordinates
`pcshow`	Plot 3-D point cloud
`pcplayer`	Visualize streaming 3-D point cloud data

References

[1] Hartley, Richard, and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. Cambridge: Cambridge University Press, 2003.

[2] Fraundorfer, Friedrich, and Davide Scaramuzza. “Visual Odometry: Part II: Matching, Robustness, Optimization, and Applications.” *IEEE Robotics & Automation Magazine* 19, no. 2 (June 2012): 78–90. https://doi.org/10.1109/MRA.2012.2182810.

[3] Mur-Artal, Raul, J. M. M. Montiel, and Juan D. Tardos. “ORB-SLAM: A Versatile and Accurate Monocular SLAM System.” *IEEE Transactions on Robotics* 31, no. 5 (October 2015): 1147–63. https://doi.org/10.1109/TRO.2015.2463671.

[4] Kümmerle, Rainer, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. "G^2o: A General Framework for Graph Optimization." In *2011 IEEE International Conference on Robotics and Automation (ICRA 2011)*, Shanghai, 9–13 May 2011, 3607–13. New York: Institute of Electrical and Electronics Engineers. https://doi.org//10.1109/ICRA.2011.5979949.

See Also

Functions

`detectSURFFeatures` | `detectSIFTFeatures` | `detectORBFeatures` | `extractFeatures` | `matchFeatures` | `matchFeaturesInRadius` | `estimateGeometricTransform2D` | `estimateGeometricTransform3D` | `estimateFundamentalMatrix` | `relativeCameraPose` | `estimateWorldCameraPose` | `worldToImage` | `triangulate` | `bundleAdjustment` | `bundleAdjustmentMotion` | `bundleAdjustmentStructure` | `createPoseGraph` | `optimizePoses`

Objects

`imageviewset` | `worldpointset` | `bagOfFeatures` | `invertedImageIndex` | `rigid3d` | `affine3d`

Related Examples

- What is SLAM?
- “Structure from Motion Overview” on page 16-49

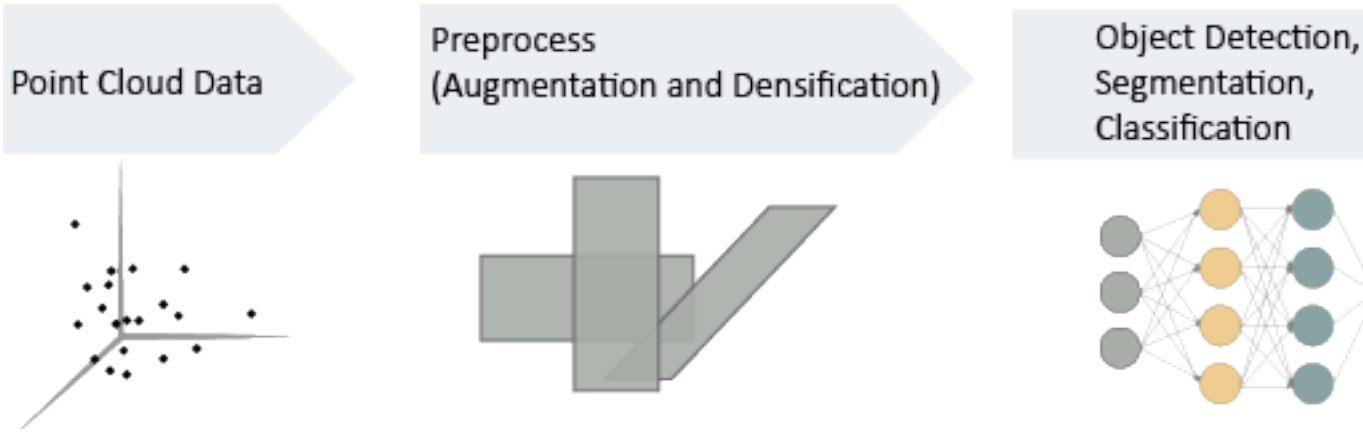
- “Visual Localization in a Parking Lot” on page 1-38
- “Stereo Visual SLAM for UAV Navigation in 3D Simulation” on page 1-44
- “Monocular Visual Simultaneous Localization and Mapping” on page 1-72
- “Stereo Visual Simultaneous Localization and Mapping” on page 1-102
- “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” (Automated Driving Toolbox)

Point Cloud Processing

- “Getting Started with Point Clouds Using Deep Learning” on page 12-2
- “Implement Point Cloud SLAM in MATLAB” on page 12-4
- “The PLY Format” on page 12-12

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 5-26 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 5-21 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 5-30 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 3-374 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 peform object classification.

See Also

`pcbin` | `pcread` | `bboxwarp` | `bboxcrop` | `bboxresize`

Related Examples

- “Lidar 3-D Object Detection Using PointPillars Deep Learning” (Lidar Toolbox)
- “Point Cloud Classification Using PointNet Deep Learning” (Deep Learning Toolbox)
- “Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network” (Deep Learning Toolbox)

More About

- Segmentation, Detection, and Labeling (Lidar Toolbox)

Implement Point Cloud SLAM in MATLAB

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).

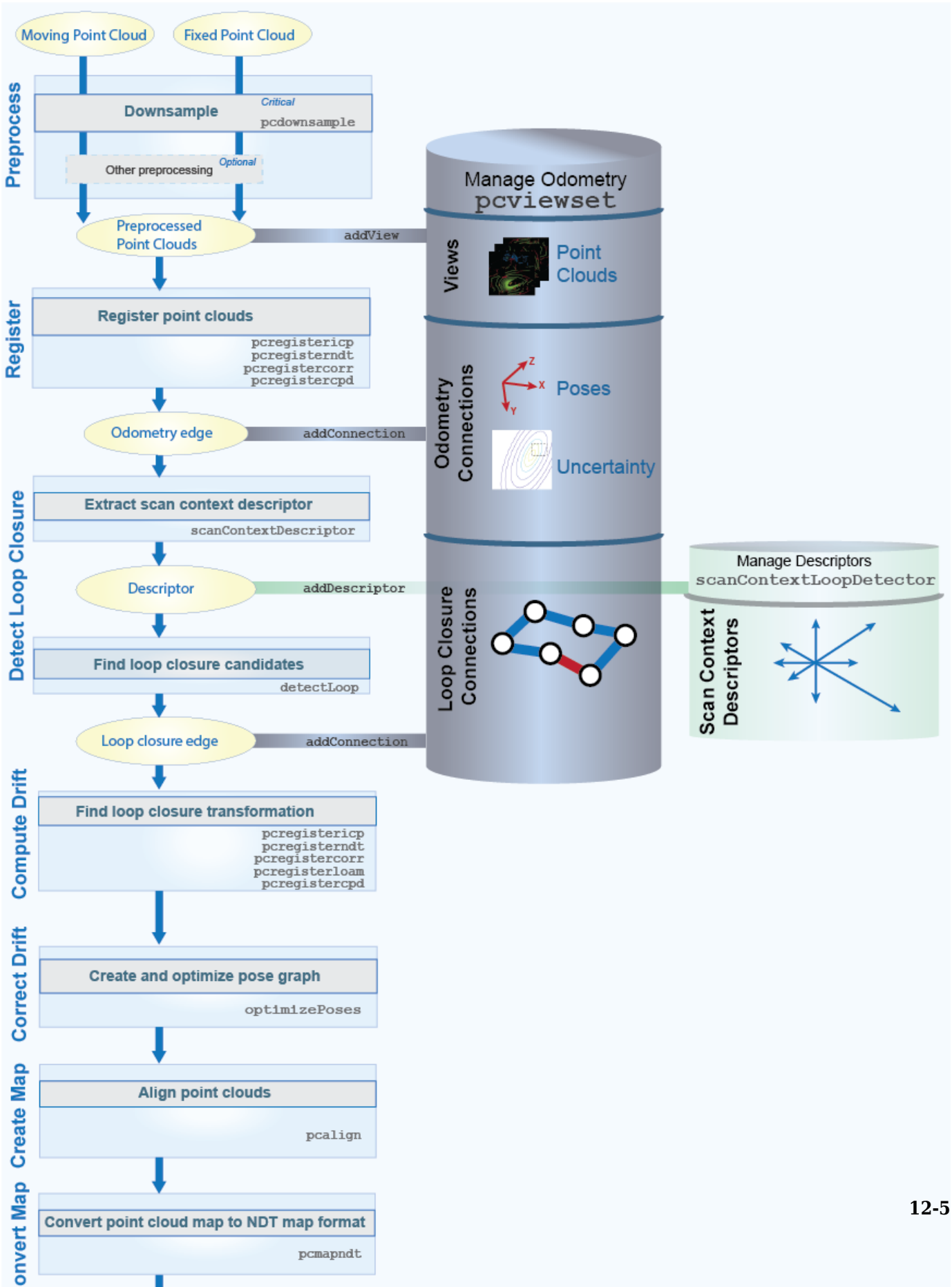
Point cloud registration is the process of aligning two or more 3-D point clouds of the same scene into a common coordinate system. Mapping is the process of building a map of the environment around a robot or a sensor. You can use registration and mapping to reconstruct a 3-D scene or build a map of a roadway for localization. While registration commonly precedes mapping, there are other applications for registration, that may not require mapping, such as deformable motion tracking. 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.

Simultaneous localization and mapping (SLAM) refers to the process of calculating the position and orientation of a vehicle, with respect to its surroundings, while simultaneously mapping the environment. The process uses only point cloud inputs from a sensor. Applications for point cloud SLAM include robotics and autonomous driving. For a general description on why SLAM matters and how it works for different applications, see What is SLAM?

Mapping and Localization Workflow

Follow these steps to perform point cloud registration and mapping on a sequence of point clouds. Then you can localize the vehicle in the prebuilt map.

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 — Perform loop closure detection to minimize drift. Loop closure detection is the process of identifying the return of the sensor to a previously visited location, which forms a loop in the trajectory of the sensor.
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 you have found sufficient loops. Pose graph optimization results in a set of optimized absolute poses.
5. Assemble Map — Assemble a point cloud map by aligning the registered point clouds using their optimized absolute poses. You can use such a prebuilt point cloud map for Localization, which is the process of locating the vehicle within the map.
6. Localize — Find the pose of the vehicle based on the assembled map.


Moving Point Cloud
Fixed Point Cloud
Preprocess
Downsample
Critical
pcdownsample
Other preprocessing
Optional
Preprocessed Point Clouds
addView
Manage Odometry
pcviewset
Views
Point Clouds
Register
Register point clouds
pcregistericp
pcregisterndt
pcregistercorr
pcregistercpd
Odometry Connections
Poses
Uncertainty
Odometry edge
addConnection
Detect Loop Closure
Extract scan context descriptor
scanContextDescriptor
Descriptor
addDescriptor
Manage Descriptors
scanContextLoopDetector
Find loop closure candidates
detectLoop
Loop Closure Connections
Scan Context Descriptors
Loop closure edge
addConnection
Compute Drift
Find loop closure transformation
pcregistericp
pcregisterndt
pcregistercorr
pcregisterloam
pcregistercpd
Correct Drift
Create and optimize pose graph
optimizePoses
Create Map
Align point clouds
pcalign
Convert Map
Convert point cloud map to NDT map format
pcmapndt

Manage Data for Mapping and Localization

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.
- `scanContextLoopDetector` object — The loop closure detection object stores scan context descriptors with a corresponding view ID. The ID associates a detected loop closure to a view in `pcviewset`.
- `pcmapndt` object — The NDT map object stores a compressed, memory-efficient map representation for localization. The object converts the point cloud map into a set of voxels (3-D boxes), each voxel represented by a 3-D normal distribution.

Preprocess Point Clouds

Preprocessing includes removing unwanted features and noise from the point clouds, and segmenting or downsampling them. Preprocessing can include these functions:

1. `pcdownsample` — Downsample the point cloud.
2. `pcsegdist` or `segmentLidarData` — Segment the point cloud data into clusters, then use the `select` function to select the desired points.
3. `pcfitplane`, `segmentGroundFromLidarData`, or `segmentGroundSMRF` — Segment the ground plane, then use the `select` function to select the desired points.
4. `pcdenoise` — Remove unwanted noise from the point cloud.

Register Point Clouds

You can use the `pcregisterndt`, `pcregistericp`, `pcregistercorr`, `pcregisterloam`, or `pcregistercpd` function to register a moving point cloud to a fixed point cloud. The registration algorithms used by these functions are based on the normal-distributions transform (NDT) algorithm, the iterative closest point (ICP) algorithm, a phase correlation algorithm, the lidar odometry and mapping (LOAM) algorithm, and the coherent point drift (CPD) algorithm, respectively. For more information on these algorithms, see References on page 12-10.

When registering a point cloud, choose the type of transformation that represents how objects in the scene change between the fixed and moving point clouds.

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 undergoing translations and rotations.
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
`pcregisterndt`	Rigid	• Local registration method that relies on an initial transform estimate • Robust to outliers • Better with point clouds of differing resolutions and densities	Provide an initial estimate to enable the algorithm to converge faster.
`pcregistericp`	Rigid	• Local registration method that relies on an initial transform estimate	
`pcregistercorr`	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	Decrease the size of the occupancy grids to decrease the computational requirements of the function.

Registration Method (function)	Transformation Type	Description	Performance Characteristics
`pcregisterloam`	Rigid	• Feature-based registration • Supports organized point clouds collected with spinning lidar sensors	Provides increased control between the processing speed versus registration accuracy Use the `detectLOAMFeatures` function to detect feature points prior to using `pcregisterloam`.
`pcregistercpd`	Rigid, affine, and nonrigid	• Global method that does not rely on an initial transformation estimate	Slowest registration method. Not recommended for map building.

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 to the view set. 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.

Tips for Registration

- Local registration methods, such as those that use NDT or ICP (`pcregisterndt` or `pcregistericp`, respectively), require initial estimates for better performance. To obtain an initial estimate, use another sensor such as an inertial measurement unit (IMU) or other forms of odometry.
- For increased accuracy in registration results, increase the value for the `'MaxIterations'` argument or decrease the value for the `'Tolerance'` argument. Changing these values in this way consequently slows registration speed.
- Consider downsampling point clouds using `pcdownsample`, before using `pcregisterndt`, `pcregistericp`, or `pcregistercpd`, to improve the efficiency and accuracy of registration.
- Denoising using `pcdenoise` before registration can improve registration accuracy, but it can slow down the execution time of the map building workflow.

Detect Loops

Using odometry alone leads to drift due to 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. Use loop closures to correct for accumulated drift. Follow these steps to detect loop closures:

1. Use the `scanContextDescriptor` function to extract scan context descriptors, which capture the distinctiveness of a point cloud.
2. Add the scan context descriptors to the `scanContextLoopDetector` using `addDescriptor`.
3. Use the `detectLoop` function to find potential loop closures.
4. 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.

For an alternative approach to loop closure detection based on segment matching, refer to the `findPose` 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 closures are detected. 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 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. This point cloud map can now be used for online localization using the NDT localization algorithm.

Localize Vehicle in Map

Convert the prebuilt point cloud map to the NDT map format using the `pcmapndt` object. The `pcmapndt` object stores the map in a compressed voxel representation that can be saved to disk and used for online localization. Use the `findPose` function to localize in the map.

Alternate Workflows

Alternative workflows for map building and localization are available in Computer Vision Toolbox, Navigation Toolbox, and Lidar Toolbox.

- Visual SLAM using Computer Vision Toolbox features — Calculate the position and orientation of a camera with respect to its surroundings, while simultaneously mapping the environment. For more details, see “Implement Visual SLAM in MATLAB” on page 11-8.
- Build an occupancy map using Navigation Toolbox features — Build an occupancy map from point clouds. For details, see “Perform SLAM Using 3-D Lidar Point Clouds” (Navigation Toolbox).
- Segment matching using Lidar Toolbox features — Build a map representation of segments and features using the `pcmapsegmatch` object. Use the `findPose` function for loop closure detection

and localization. This approach is robust to dynamic obstacles and is recommended for large scale environments. For an example of this approach, see the “Build Map and Localize Using Segment Matching” (Lidar Toolbox) example. The table highlights the similarities and differences between the `pcmapndt` and `pcmapsegmatch` map representations.

Workflow	**`pcmapndt`**	**`pcmapsegmatch`**
Algorithm	Normal distributions transform (NDT)	SegMatch — segment matching approach
Mapping	Build the map first — Incrementally build the map using `pcviewset`. Then, use `pcalign` to assemble the map and convert the prebuilt map to an NDT map representation.	Build the map incrementally using `pcmapsegmatch` — Add views to `pcviewset` (using `addView`) and to `pcmapsegmatch` (using `addView`) for each point cloud scan. Detect loop closures using `findPose` and correct for accumulated drift with `optimizePoses`.
Localization Similarity	Select a submap for localization, and then find the pose for localization using one set of the following options: • The `selectSubmap` and `findPose` functions of the `pcmapndt` object. • The `selectSubmap` and `findPose` functions of the `pcmapsegmatch` object.	
Localization Difference	Relies on a pose estimate.	Does not rely on a pose estimate.
Visualization	Visualize the map or selected submap using the `show` function of the `pcmapndt` object or the `show` function of the `pcmapsegmatch` object.	

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. https://doi.org/10.1109/TPAMI.2010.46

[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. https://doi.org/10.1016/0262-8856(92)90066-C.

[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. https://doi.org/10.1109/34.121791.

[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. https://doi.org/10.1109/IROS.2003.1249285.

[5] Magnusson, Martin. “The Three-Dimensional Normal-Distributions Transform: An Efficient Representation for Registration, Surface Analysis, and Loop Detection.” PhD thesis, Örebro universitet, 2009. http://urn.kb.se/resolve?urn=urn:nbn:se:oru:diva-8458 urn:nbn:se:oru:diva-8458

[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. https://doi.org/10.5220/0005719006260633.

[7] Zhang, Ji, and Sanjiv Singh. “LOAM: Lidar Odometry and Mapping in Real-Time.” In *Robotics: Science and Systems X*. Robotics: Science and Systems Foundation, 2014. https://doi.org/10.15607/RSS.2014.X.007.

See Also

Functions

`pcregistercorr` | `pcregisterndt` | `pcregistericp` | `pcregistercpd` | `pcalign` | `scanContextDistance` | `scanContextDescriptor` | `findPose` | `show` | `segmentGroundSMRF`

Objects

`pointCloud` | `pcviewset` | `rigid3d` | `pcmapndt` | `scanContextLoopDetector`

Related Examples

- What is SLAM?
- “What are Organized and Unorganized Point Clouds?” (Lidar Toolbox)
- “Build a Map from Lidar Data Using SLAM” on page 5-55
- “Implement Visual SLAM in MATLAB” on page 11-8
- “Design Lidar SLAM Algorithm Using Unreal Engine Simulation Environment” (Automated Driving Toolbox)
- “3-D Point Cloud Registration and Stitching” on page 5-71
- “Build a Map from Lidar Data” on page 5-35
- “Build Map and Localize Using Segment Matching” (Lidar Toolbox)

The PLY Format

In this section...
“File Header” on page 12-12 “Data” on page 12-13 “Common Elements and Properties” on page 12-14

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 coefficent

See Also

`pcread` | `pcwrite`

Using the Installer for Computer Vision System Toolbox Product

- “Install Computer Vision Toolbox Add-on Support Files” on page 13-2
- “Install OCR Language Data Files” on page 13-3
- “Install and Use Computer Vision Toolbox Interface for OpenCV in MATLAB” on page 13-6
- “Build MEX-Files for OpenCV Interface” on page 13-8
- “Use Prebuilt MATLAB Interface to OpenCV” on page 13-10
- “Perform Edge-Preserving Image Smoothing Using OpenCV in MATLAB” on page 13-15
- “Subtract Image Background by Using OpenCV in MATLAB” on page 13-19
- “Perform Face Detection by Using OpenCV in MATLAB” on page 13-22
- “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 13-24
- “Draw Different Shapes by Using OpenCV Code in Simulink” on page 13-31
- “Convert RGB Image to Grayscale Image by Using OpenCV Importer” on page 13-38
- “Smile Detection by Using OpenCV Code in Simulink” on page 13-45
- “Shadow Detection by Using OpenCV Code in Simulink” on page 13-55
- “Vehicle and Pedestrian Detector by Using OpenCV Importer” on page 13-60
- “Video Cartoonizer by Using OpenCV Code in Simulink” on page 13-64
- “Convert Between Simulink Image Type and Matrices” on page 13-69

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:

- 

- 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 13-3 “Pretrained Language Data and the ocr function” on page 13-3

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:

- 

- 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` | `visionSupportPackages` | **OCR Trainer**

Related Examples

- “Recognize Text Using Optical Character Recognition (OCR)” on page 4-50

Install and Use Computer Vision Toolbox Interface for OpenCV in MATLAB

In this section...
“Installation” on page 13-6 “Support Package Contents” on page 13-6

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:

- 

- 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
`clibopencv` folder	Prebuilt utility functions based on the MATLAB `clib` package for OpenCV interface.
`mexOpenCV.m` file	Function to build MEX-files.
`example` folder	Template Matching, Foreground Detector, and Oriented FAST and Rotated BRIEF (ORB) examples using MEX-files. Each subfolder in the example folder contains a `README.txt` file with step-by-step instructions.
`registry` folder	Registration 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 or later (Professional or Community editions)
Linux® 64 bit	gcc-4.9.3 (g++)
Mac 64 bit	Xcode 6.2.0 (Clang++)

Build MEX-Files for OpenCV Interface

In this section...
“Create MEX-File from OpenCV C++ file” on page 13-8
“Create Your Own OpenCV MEX-files” on page 13-8
“Run OpenCV Examples” on page 13-8

Use the OpenCV Interface files to integrate your OpenCV C++ code into MATLAB and build MEX-files that call OpenCV functions.

The `mexOpenCV` interface utility functions convert data between OpenCV and MATLAB. These functions support CPP-linkage only.

The Computer Vision Toolbox Interface for OpenCV in MATLAB API supports OpenCV version 4.2.0 and 4.5.0.

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
   ```

Create Your Own OpenCV MEX-files

Call the `mxArray` function with your source file.

```
mexOpenCV yourfile.cpp
```

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

```
help mexOpenCV
```

Run OpenCV Examples

Each example subfolder in the 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 matchTemplateOCV.cpp
```

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

```
testMatchTemplate
```

Run Foreground Detector Example

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

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

2. Create the MEX-file from the source file:

```
mexOpenCV backgroundSubtractorOCV.cpp
```

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

```
testBackgroundSubtractor.m
```

Run Oriented FAST and Rotated BRIEF (ORB) Detector Example

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

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

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

```
mexOpenCV detectORBFeaturesOCV.cpp
```

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

```
mexOpenCV extractORBFeaturesOCV.cpp
```

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

```
testORBFeaturesOCV.m
```

See Also

`mxArray` | “C Matrix API”

More About

- “Install Computer Vision Toolbox Add-on Support Files” on page 13-2
- Using OpenCV with MATLAB

Use Prebuilt MATLAB Interface to OpenCV

In this section...
“Call MATLAB Functions” on page 13-11 “Call Functions in OpenCV Library” on page 13-11 “Display Help for MATLAB Functions” on page 13-12 “Display Help for MATLAB Interface to OpenCV Library” on page 13-12 “Limitations” on page 13-13

The Computer Vision Toolbox Interface for OpenCV provides a prebuilt MATLAB interface to the OpenCV library that you can use to directly call OpenCV functions from MATLAB without writing a C /C++ code.

Note The Computer Vision Toolbox Interface for OpenCV in MATLAB supports OpenCV version 4.2.0 and 4.5.0. The interface does not provide GPU support.

The Computer Vision Toolbox Interface for OpenCV also provides MATLAB functions to:

- Create MATLAB objects that represent `Mat`, `UMat`, OpenCV base classes pointed by a smart pointer, and OpenCV arrays.

MATLAB **Function**	**Description**
`createMat`	Returns `Mat` objects that represent the OpenCV `cv::Mat` data structure, and the associated input, output, or input-output array class.
`createUMat`	Returns `UMat` objects that represent the OpenCV `cv::UMat` data structure, and the associated input, output, or input-output array class.
`clibArray`	Returns an object that represents OpenCV native arrays or `std::vector` types.
`getBasePtr`	Returns an object that represents an OpenCV base class pointed by a smart pointer.

- Export OpenCV outputs to MATLAB supported formats.

MATLAB **Function**	**Description**
`getImage`	Reads and exports images stored in OpenCV `Mat` and `UMat` objects to a matrix or 3-D array in the MATLAB workspace.
`keyPointsToStruct`	Exports the keypoints returned by an OpenCV keypoint detector as a MATLAB structure.

`rectToBbox`	Exports the parameters returned by an OpenCV rectangle class to a vector in the MATLAB workspace. The parameters are exported as bounding box coordinates in one-based indexing.
`underlyingValue`	Returns the underlying numeric values for OpenCV enumeration objects.

The OpenCV functions in the prebuilt library do not return errors except during the run-time.

Call MATLAB Functions

To call a MATLAB function in the Computer Vision Toolbox Interface for OpenCV support package, add the package name `vision.opencv.util` to the import list and then call the MATLAB function. For example:

```
import vision.opencv.util.*
[ocvMat,ocvArray] = createMat;
```

You can also add the partial package name `vision.opencv` to the import list and call the MATLAB function by prepending `util` to the function name. For example:

```
import vision.opencv.*
[ocvMat,ocvArray] = util.createMat;
```

Alternatively, you can also call the MATLAB function by prefixing it with the full package name. Use this syntax to import a specific function without importing every function in the package.

```
[ocvMat,ocvArray] = vision.opencv.util.createMat;
```

Call Functions in OpenCV Library

To call a function or class in the prebuilt MATLAB interface to the OpenCV library, add the library name `clib.opencv` to the import list. Then call an OpenCV function by prefixing the function name with a namespace.

```
import clib.opencv.*
retVal = namespace.funcname(arg1,arg2,…,argN)
```

`namespace` is the namespace of the function, `funcname` is the OpenCV function name, and `arg1`,`arg2`,...,`argN` represents the input arguments for the OpenCV function. `retVal` is the output argument.

Alternatively, you can also call an OpenCV function by adding the package name as a prefix as shown here.

```
retVal = clib.opencv.namespace.funcname(arg1,arg2,…,argN)
```

Note The interface supports only the `cv` and `cvflann` namespaces. For example, you can call a function in the `cv` namespace by using this syntax:

```
retVal = clib.opencv.cv.funcname(arg1,arg2,…,argN)
```

Display Help for MATLAB Functions

To view information about the MATLAB functions in the support package, use `doc` or `help`.

Type one of these commands in MATLAB command window.

```
doc vision.opencv.util.
```

or

```
help vision.opencv.util.
```

Then press **Tab**. This command loads the package, and MATLAB displays the list of functions in the package.

To view information about the package in the Help browser, type this command in the MATLAB command window, and then press **Enter**.

```
doc vision.opencv.util
```

Display Help for MATLAB Interface to OpenCV Library

Use these MATLAB functions to view information about the classes and the functions in the Prebuilt MATLAB interface to OpenCV library.

- `doc` and `help` — View classes and functions in the OpenCV library.
- `methods` — View constructor, method, and package function names for a class.
- `methods` with `'-full'` option — View constructor, method, and package function signatures.
- `methodsview` — View a table representation of method signatures. The `methodsview` window enables you to find information without scrolling through the Command Window.

Display the classes and package functions by entering this command and then pressing **Tab**.

```
doc clib.opencv.
```

This command loads the package, and MATLAB displays a list of the package members. To view the package, press **Backspace** to remove the period, then press **Enter**. MATLAB displays all the classes and functions in the OpenCV library.

To display class methods, call the `methods` function for that class. For example, to see the methods of the `Affine3d` class, enter this command.

```
methods clib.opencv.cv.Affine3d
```

```
Methods for class clib.opencv.cv.Affine3d:

Affine3d     eq          gt          le          ne          rotation    translate
concatenate  ge          inv         lt          rotate      rvec        translation

Static methods:

Identity

Methods of clib.opencv.cv.Affine3d inherited from handle.
```

To display method signatures for a class, call the `methodsview` function for that class. For example:

```
methodsview clib.opencv.cv.Affine3d
```

This command opens a window that displays the methods and information about their arguments and returned values.

Limitations

The prebuilt MATLAB interface to the OpenCV library does not support functionalities that contain these following data types or language features.

- Any type with a size greater than 64 bits, such as `long double`
- Arrays of character types (`wchar_t`, `char16_t`, `char32_t`)
- References to a pointer, such as `int*&`
- Pointers or arrays of type `std::string`
- Pointers or references to enumerations
- Reference data members
- `void*` data members
- `std` smart pointers other than `std::shared_ptr`
- Static data members
- ** pointers, except:
  - MATLAB supports `char**`
  - MATLAB supports ** pointers to custom classes used as function or method parameter types.
- Multilevel pointers, such as `type***`
- C function pointers and `std::function` as function return types or data members. You also cannot pass a MATLAB function as input to C function pointers or an `std::function` parameter.
- Class templates with incomplete or no instantiations
- User-defined data type `union`
- Arrays of type `std::vector`
- Types defined in the `std` namespace, except these supported types:
  - `std::string`
  - `std::wstring`
  - `std::u16string`
  - `std::u32string`
  - `std::vector`
  - `std::shared_ptr`
  - `std::function`

See Also

More About

- “MATLAB Object For C++ Arrays”
- “Install Computer Vision Toolbox Add-on Support Files” on page 13-2

- “Use Prebuilt MATLAB Interface to C++ Library”
- Using OpenCV with MATLAB

Perform Edge-Preserving Image Smoothing Using OpenCV in MATLAB

This example shows how to perform edge-preserving image smoothing by using prebuilt MATLAB® interface to the OpenCV function `cv::edgePreservingFilter`. In this example, you also use the `createMat` utility function to define the input and output arrays, and the `getImage` utility function to read the output image returned by the OpenCV function.

Add the MATLAB interface to OpenCV package names to the import list.

```
import clib.opencv.*;
import vision.opencv.util.*;
```

Read an image into the MATLAB workspace.

```
img = imread("peppers.png");
```

Create MATLAB interface objects for the OpenCV `MatND` and `InputArray` classes to store the input image.

```
[inputMat,inputArray] = createMat(img);
```

Create MATLAB interface objects for the OpenCV `MatND` and `OutputArray` classes to write the output image returned by the OpenCV function.

```
[outputMat,outputArray] = createMat;
```

Smooth the input image by using the OpenCV function `cv::edgePreservingFilter`. To call this function from MATLAB, you must use the notation `cv.edgePreservingFilter`.

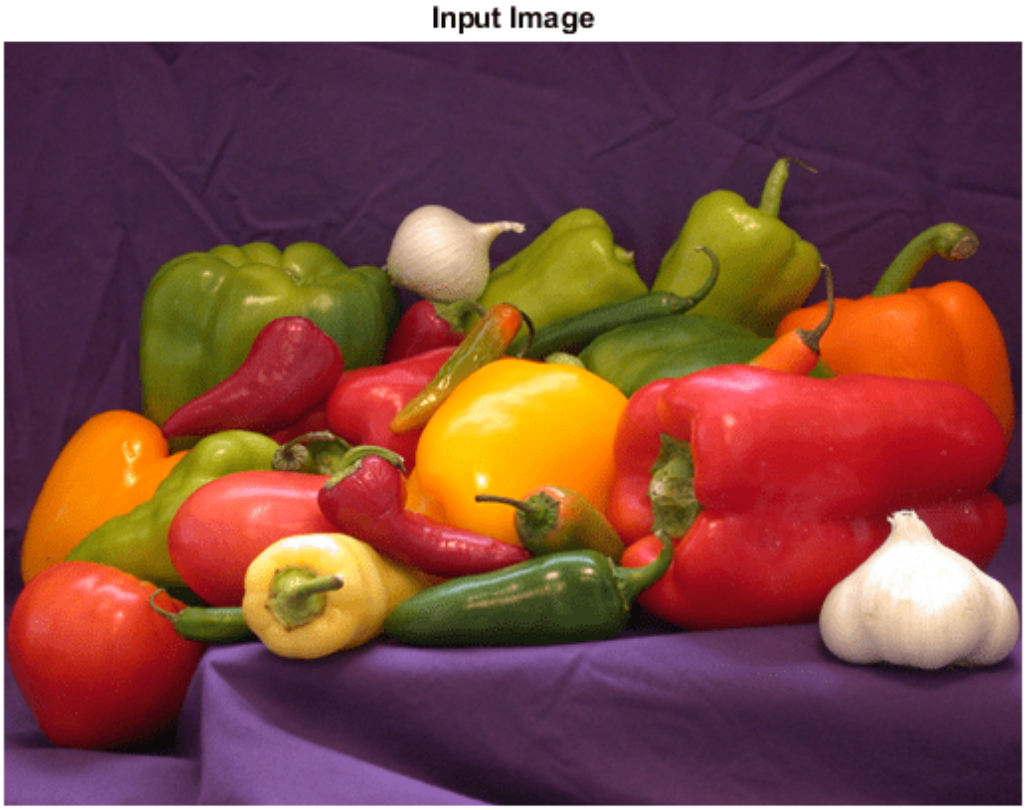
```
cv.edgePreservingFilter(inputArray,outputArray);
```

Read the filtered output image returned by the OpenCV function.

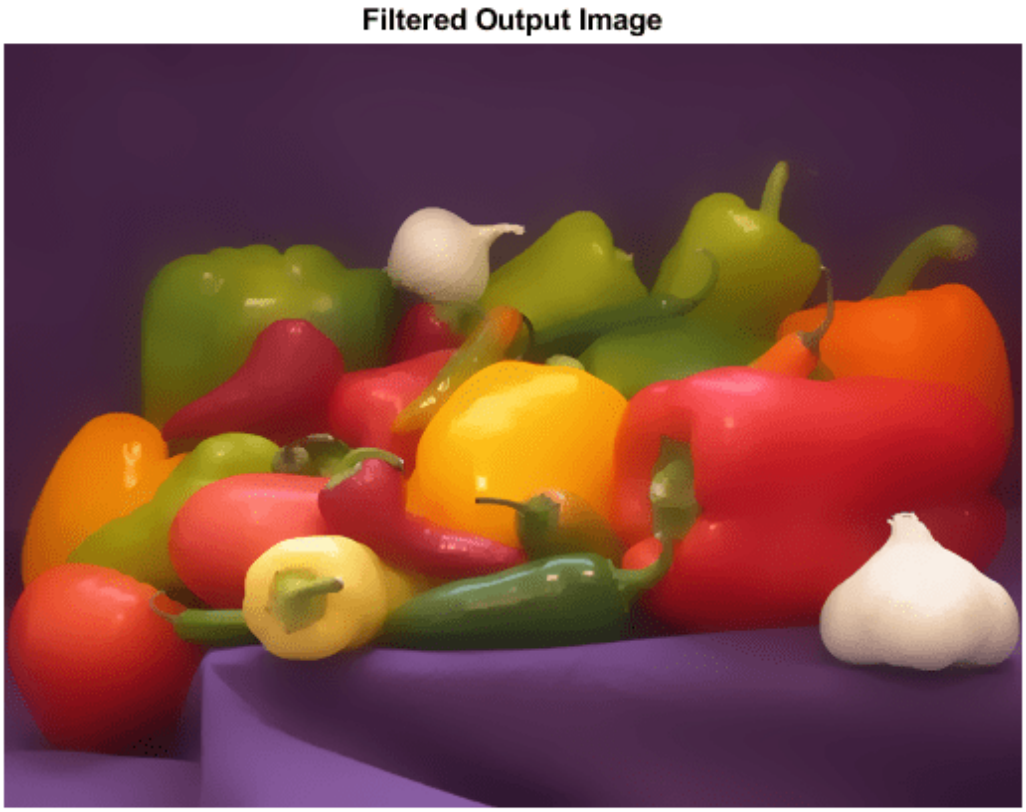
```
filteredImg = getImage(outputArray);
```

Display the original input and the filtered output images.

```
figure
imshow(img)
title("Input Image")
```




```
figure
imshow(filteredImg)
title("Filtered Output Image")
```



Modify Filter Parameters

Specify the parameter values for the edge-preserving filter. Set these values:

- `flags` to 2, to perform normalized convolution filtering.
- Standard deviations `sigma_s` to `20` and `sigma_r` to `0.3`.

```
flags = 2;
sigma_s = 20;
sigma_r = 0.2;
```

Perform filtering by using the defined filter parameters.

```
cv.edgePreservingFilter(inputArray,outputArray,flags,sigma_s,sigma_r);
```

Read the filtered output image returned by the OpenCV function.

```
filteredImg = getImage(outputArray);
```

Display the filtered output image.

```
figure
imshow(filteredImg)
title("Filtered Output Image")
```

See Also

Functions
`createMat` | `getImage`

More About

- “Use Prebuilt MATLAB Interface to OpenCV” on page 13-10

Subtract Image Background by Using OpenCV in MATLAB

This example shows how to subtract the background in an image sequence or a video by using the prebuilt MATLAB interface to the OpenCV function `cv::BackgroundSubtractorKNN`. In this example, you also use the `createMat` utility function to define the input and output arrays, and the `getImage` utility function to read the output image returned by the OpenCV function. The input video must be have a static background and dynamic foreground objects.

Read a video into the MATLAB workspace by using the `VideoReader` MATLAB function.

```
videoSample = VideoReader("atrium.mp4");
videoSample.CurrentTime = 2.5;
```

Add the MATLAB interface to OpenCV package names to the import list.

```
import clib.opencv.*;
import vision.opencv.*;
```

Specify the parameter values to compute the background by using the OpenCV function for k-Nearest Neighbor (KNN) background subtractor `cv::BackgroundSubtractorKNN`. Set these values:

- Number of last frames to consider for computing the KNN background model `history` to `300`.
- Threshold for differentiating the foreground and background pixels `threshold` to `400`.
- `shadow` to `true`, to detect the shadow regions.

```
history = 300;
threshold = 400;
shadow = true;
```

Create a MATLAB interface object by using the `getBasePtr` utility function to represent the `BackgroundSubtractorKNN` class.

```
cvPtr = cv.createBackgroundSubtractorKNN(history,threshold,shadow);
kNNBase = util.getBasePtr(cvPtr);
```

You can also set the parameter values for the background subtractor by accessing the public methods of the `BackgroundSubtractorKNN` class. Set the number of `k` nearest neighbors required for classifying a pixel as belonging to the background model to 2.

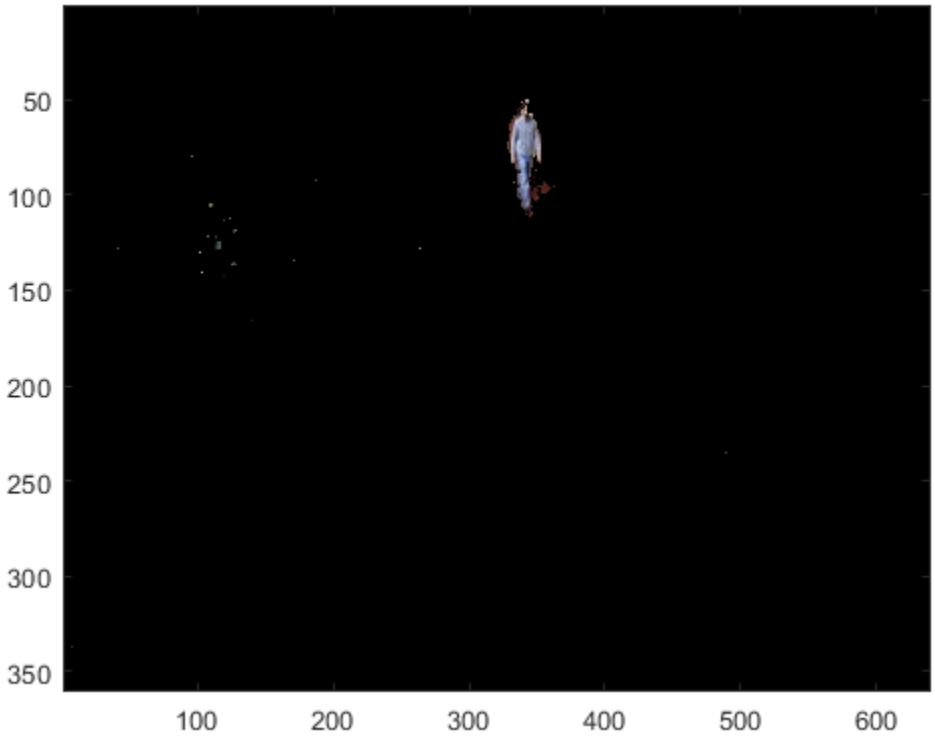
```
kNNBase.setkNNSamples(2);
```

Follow these steps to extract the foreground region by using the `apply` method of the OpenCV class `BackgroundSubtractorKNN`.

- Create an `InputArray` and `OutputArray` class by using the `createMat` MATLAB utility function to store the input video frame and the output foreground mask respectively.
- The `apply` method takes the video frames as inputs and then, computes the foreground pixels by using the k-NN algorithm. The `apply` method stores the mask containing the foreground pixel regions to the `OutputArray` class.
- Export the output foreground mask returned by the `apply` method to MATLAB workspace by using the `getImage` MATLAB utility function.
- Extract the desired foreground region by using the foreground mask and the input video frames.

```
foregroundmask = zeros(videoSample.Height,videoSample.Width,videoSample.NumFrames);
while hasFrame(videoSample)
```

```
    frame = readFrame(videoSample);
    [inMat,imgInput] = util.createMat(frame);
    [outMat,outImg] = util.createMat();
    kNNBase.apply(imgInput,outImg);
    foregroundmask = util.getImage(outImg);

    foregroundmask = rescale(foregroundmask);
    foregroundmask = cast(foregroundmask,"like",frame);

    foreground(:,:,1) = frame(:,:,1).*foregroundmask;
    foreground(:,:,2) = frame(:,:,2).*foregroundmask;
    foreground(:,:,3) = frame(:,:,3).*foregroundmask;

    image(foreground,Parent=gca);
    pause(0.01);
end
```



See Also

Objects
`VideoReader`

Functions
`createMat` | `rescale` | `getImage` | `createUMat` | `readFrame` | `getBasePtr` | `cast`

More About

- “Use Prebuilt MATLAB Interface to OpenCV” on page 13-10

Perform Face Detection by Using OpenCV in MATLAB

This example shows how to detect faces in an image or video by using prebuilt MATLAB® interface to the OpenCV function `cv::CascadeClassifier`. This example uses a Harr face detection model that is trained for scale-invariant, frontal face detection. In this example, you also use the `createMat` utility function to define the input and output arrays, the `getImage` utility function to read the output image returned by the OpenCV function, and the `rectToBbox` utility function to convert the face detection output returned by the OpenCV function to bounding box coordinates in MATLAB®.

Read a video into the MATLAB workspace by using the `VideoReader` MATLAB function.

```
videoSample = VideoReader("tilted_face.avi");
```

Add the MATLAB interface to OpenCV package names to the import list.

```
import clib.opencv.*;
import vision.opencv.util.*;
```

Specify the file name of a pre-trained trained Haar face detection model.

```
trainedModel = "haarcascade_frontalface_alt.xml";
```

Load the pre-trained model by using the `load` method of the OpenCV function `cv.CascadeClassifier`.

```
cascadeClassify = cv.CascadeClassifier();
cascadeClassify.load(trainedModel);
```

Specify the scale factor to use for multi-scale detection.

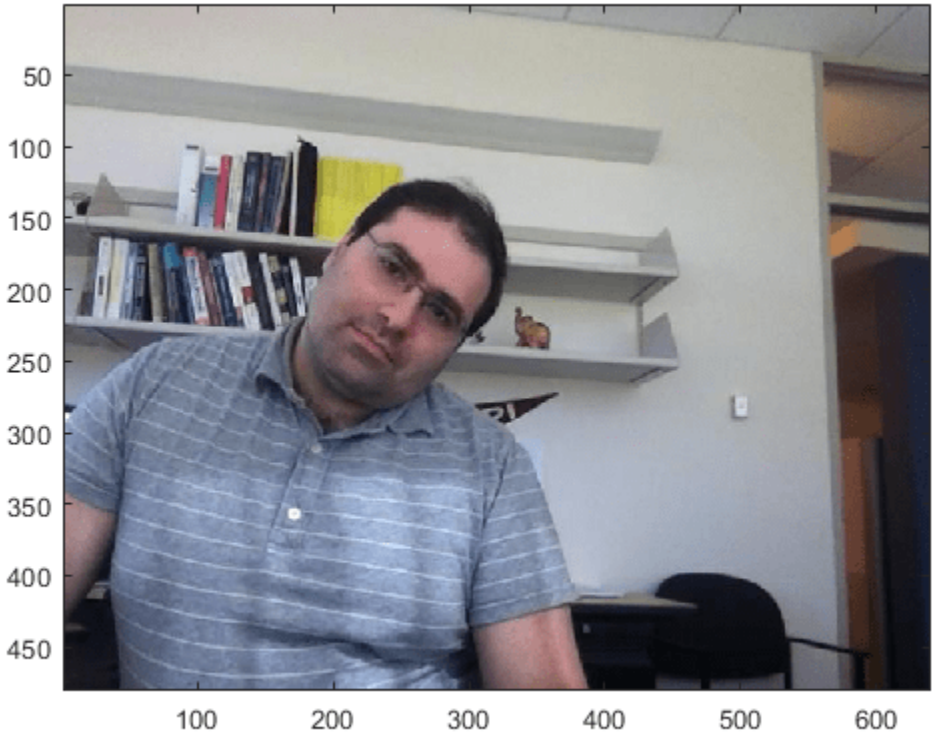
```
scaleFactor = 1.2;
```

Follow these steps to detect faces in each frame by using the `detectMultiScale` method of the OpenCV class `cv.CascadeClassifier`.

- Create a `Mat` object and store the input frame to the `Mat` object by using the `createMat` function. Specify the `Mat` object as an input to the `detectMultiScale` method.
- Create a MATLAB array to represent the OpenCV class for 2D rectangles `cv::Rect2i`. Specify the array as an input to the `detectMultiScale` method. The method uses the array to return the detection results.
- Export the detection results returned by the `detectMultiScale` method to a row vector by using the `rectToBbox` function. The row vector specifies bounding box coordinates in one-based indexing.
- Draw the bounding boxes on the input frame to represent the detected faces.

```
count = 1;
detections = cell(1,videoSample.NumFrames);
while(hasFrame(videoSample))
    testFrame = readFrame(videoSample);

    [inputMat,inputArray] = createMat(testFrame);
    results = clibArray("clib.opencv.cv.Rect2i", 0);

    cascadeClassify.detectMultiScale(inputArray,results,scaleFactor);
    if results.Dimensions ~= 0
```

```
        detections{count} = rectToBbox(results);
    else
        detections{count} = [];
    end
    testFrame = insertShape(testFrame,Rectangle=detections{count},LineWidth=5);
    image(testFrame,Parent=gca);
    pause(0.01)
    count = count+1;
end
```



See Also

Objects
`VideoReader`

Functions
`createMat` | `getImage` | `createUMat` | `readFrame` | `clibArray` | `rectToBbox` | `insertShape`

More About

- “Use Prebuilt MATLAB Interface to OpenCV” on page 13-10

Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink

In this section...
“Installation” on page 13-24 “Import OpenCV Code into Simulink” on page 13-24 “Limitations” on page 13-29

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, use one of the following methods:

- 

- Select **Get Add-ons** from the **Add-ons** drop-down menu from the MATLAB toolstrip. In the **Add-Ons** Explorer window, find and click the Computer Vision Toolbox Interface for OpenCV in Simulink support package, and then click **Install**.
- Type `visionSupportPackages` in a MATLAB Command Window and follow the prompts.

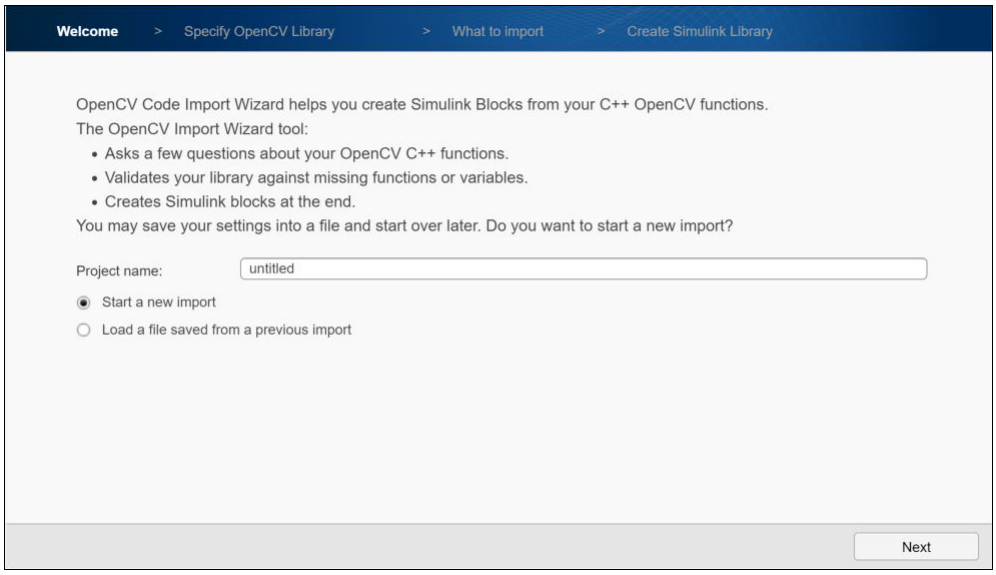
Import OpenCV Code into Simulink

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 Importer app opens to a Welcome page.

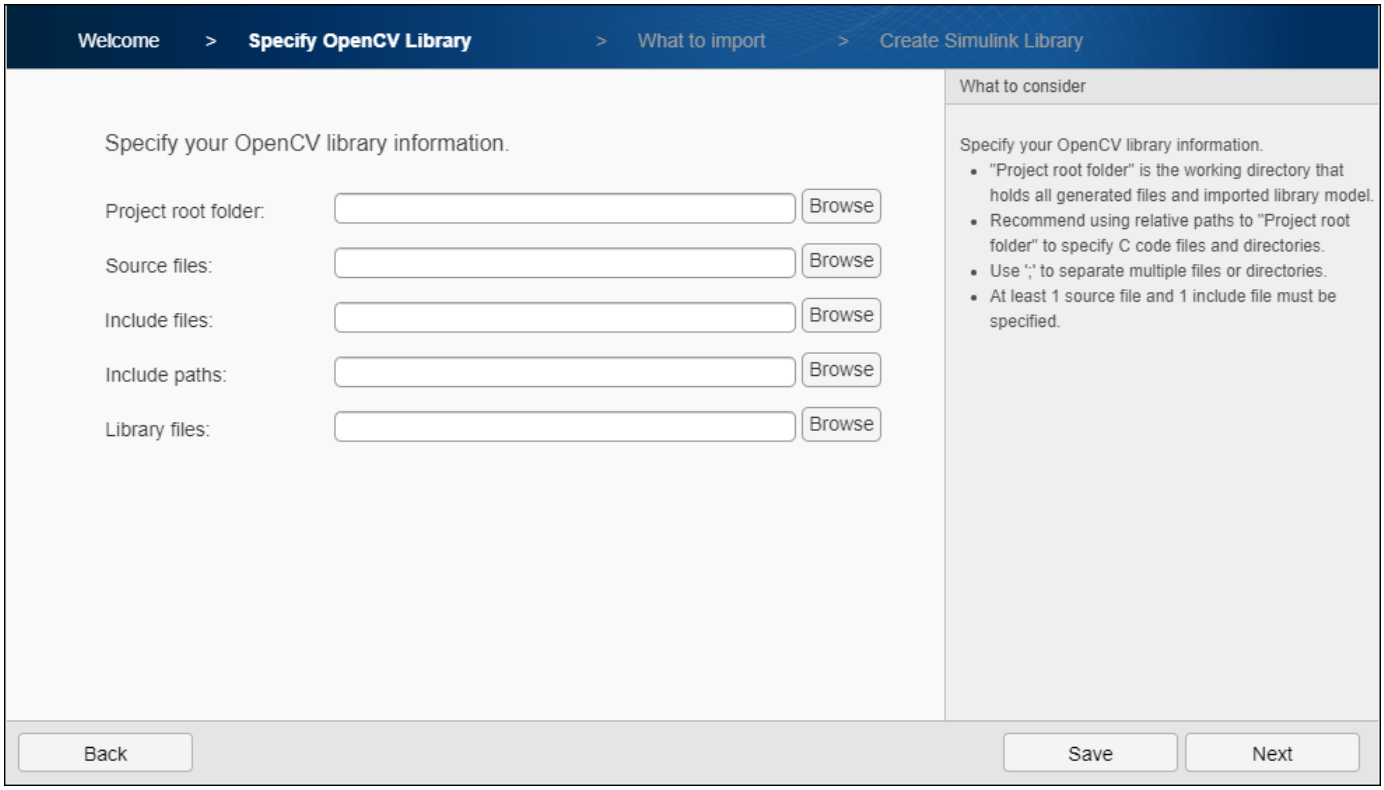
1. In the **Project name** field, specify a name for your import. You can either start a new import or load files saved from a previous import. The projects are saved in `.m` file format. To browse a saved file from previous import, select **Load a file saved from a previous import**. Click **Next**.



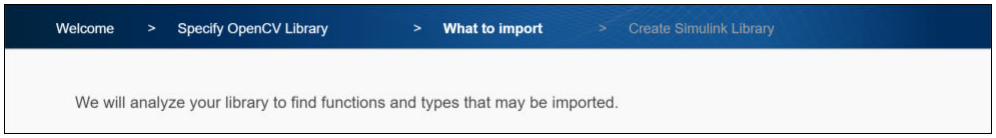
2 In the Specify OpenCV Library page, specify your C++ library information. If you import a previously saved project file, all the fields are autopopulated.

- **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 app uses the file from the specified location. If you do not provide the absolute path, then the app 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 app uses the file from the specified location. If you do not provide the absolute path, then the app uses the path relative to the project root.
- **Include paths:** Define any additional include folders (Folder information). MATLAB OpenCV include files are included.
- **Library files:** Specify the external library files.

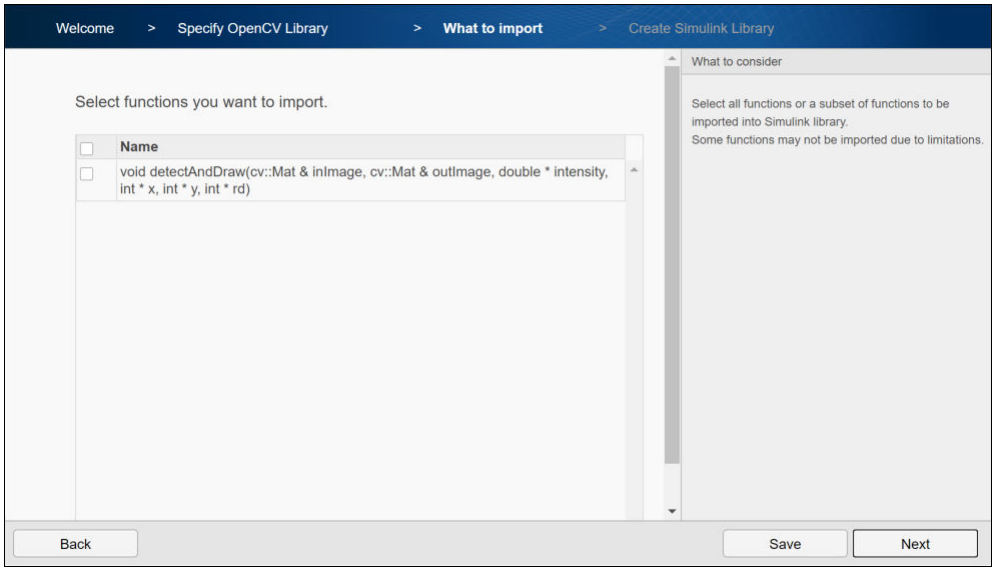
If you have multiple files or folders to specify, use a semicolon-separated list of files or folders. Click **Next**.



3 To find functions and types that are supported for import, analyze your library by clicking **Next**. Once the analysis is complete, click **Next**.



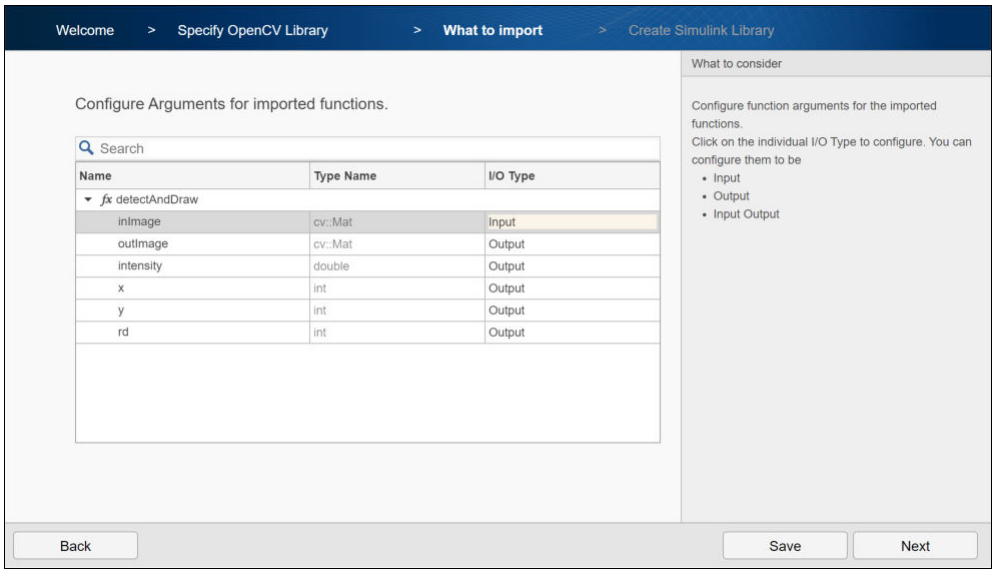
4 In the What to Import page, the functions that Computer Vision Toolbox Interface for OpenCV in Simulink supports are listed. Select the functions that you want to import into Simulink library and click **Next**.



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**.

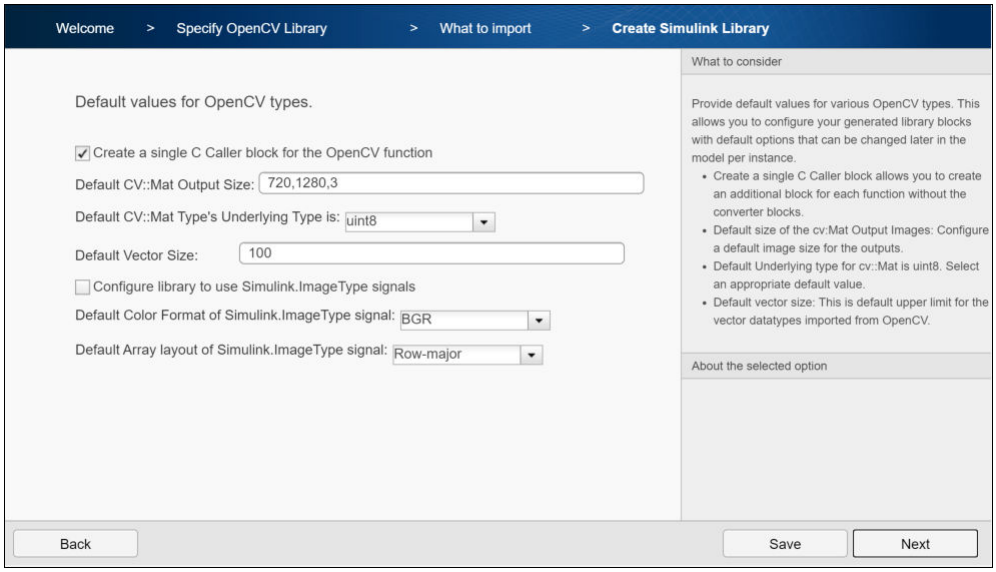


6 In the Create Simulink Library page, you can generate either just a subsystem block or a subsystem block and a C Caller block of the selected function. A C Caller block integrates your OpenCV data into Simulink. The generated subsystem block contains C Caller blocks configured by using data conversion blocks.

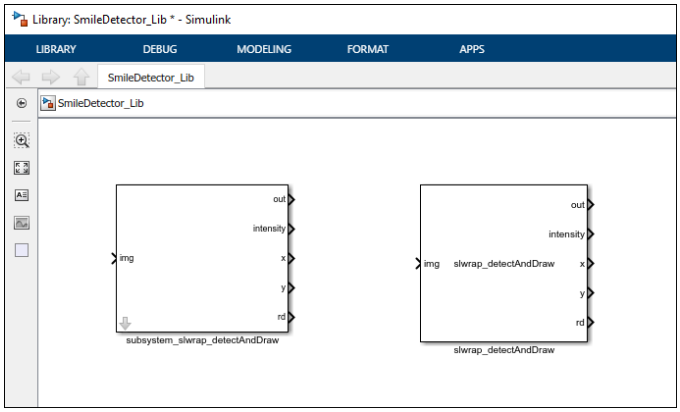
To generate a subsystem block and a C Caller block, select **Create a single C-caller block for the OpenCV function**, and then click **Next**.

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.

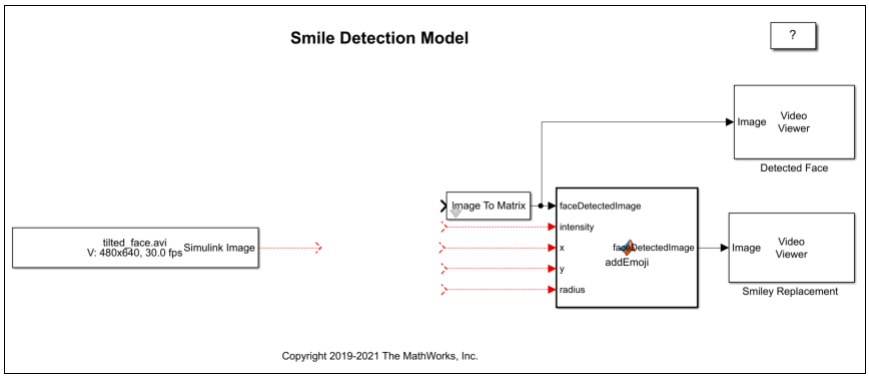
To use a `Simulink.ImageType` data type for your images, select **Configure library to use Simulink.ImageType signals** and configure parameters **Default Color Format of Simulink.ImageType signal** and **Default Array layout of Simulink.ImageType signal** based on your model requirements.



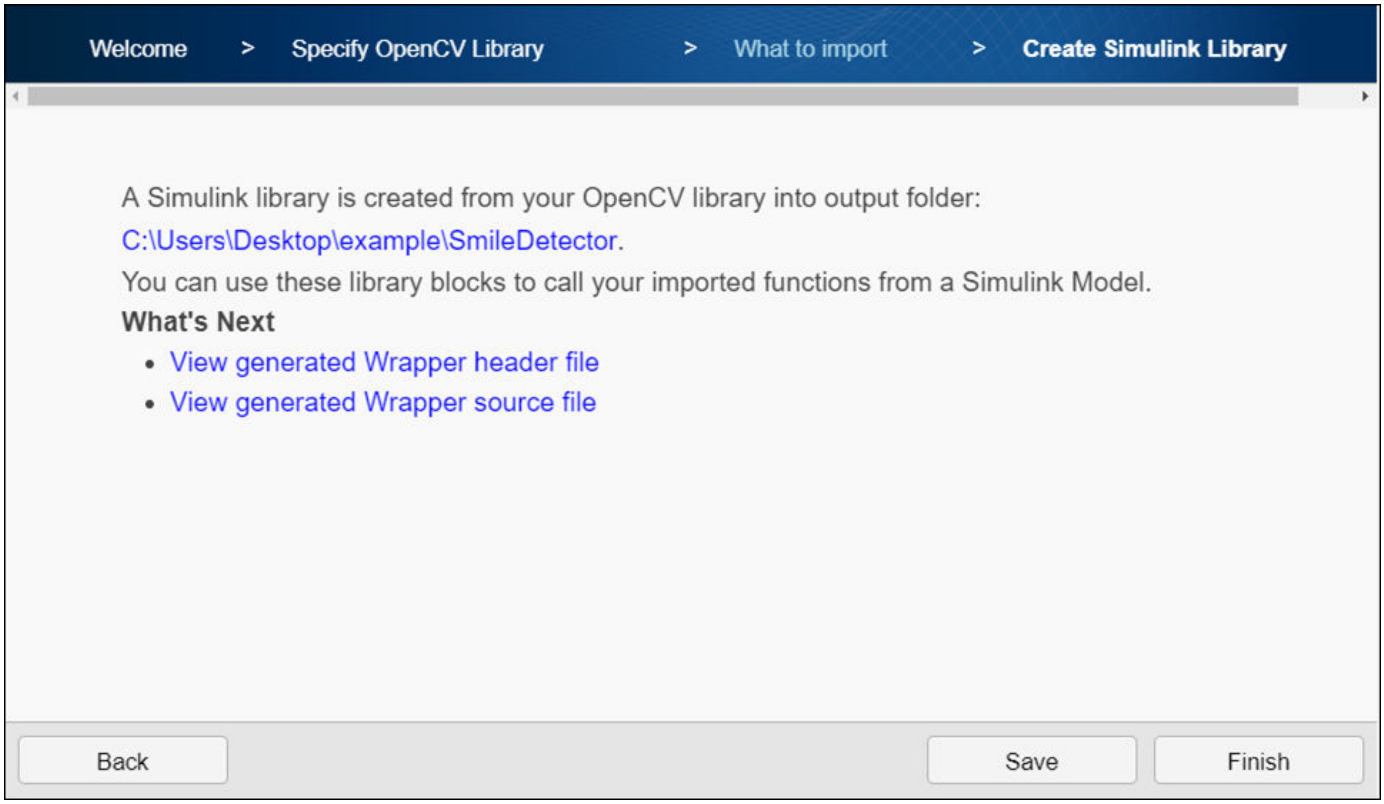
7 A Simulink library `Projectname_Lib.slx` is created from your OpenCV code into a project root folder. The library contains a subsystem block and a C Caller block.



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 app 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 later Professional and Community editions for Windows 64 operating system. For more information on compilers, see “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4.

- Supports C++ code generation that uses row-major array layout.
- `CV_type` signals are not supported across referenced SIL model boundaries.

See Also

ToOpenCV | FromOpenCV

More About

- “Smile Detection by Using OpenCV Code in Simulink” on page 13-45
- “Convert RGB Image to Grayscale Image by Using OpenCV Importer” on page 13-38
- “Draw Different Shapes by Using OpenCV Code in Simulink” on page 13-31

Draw Different Shapes by Using OpenCV Code in Simulink

This example shows how to draw different shapes on images by using **OpenCV Importer**.

First import an OpenCV function into Simulink® by using the “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 13-24. 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.

Set Up Your C++ Compiler

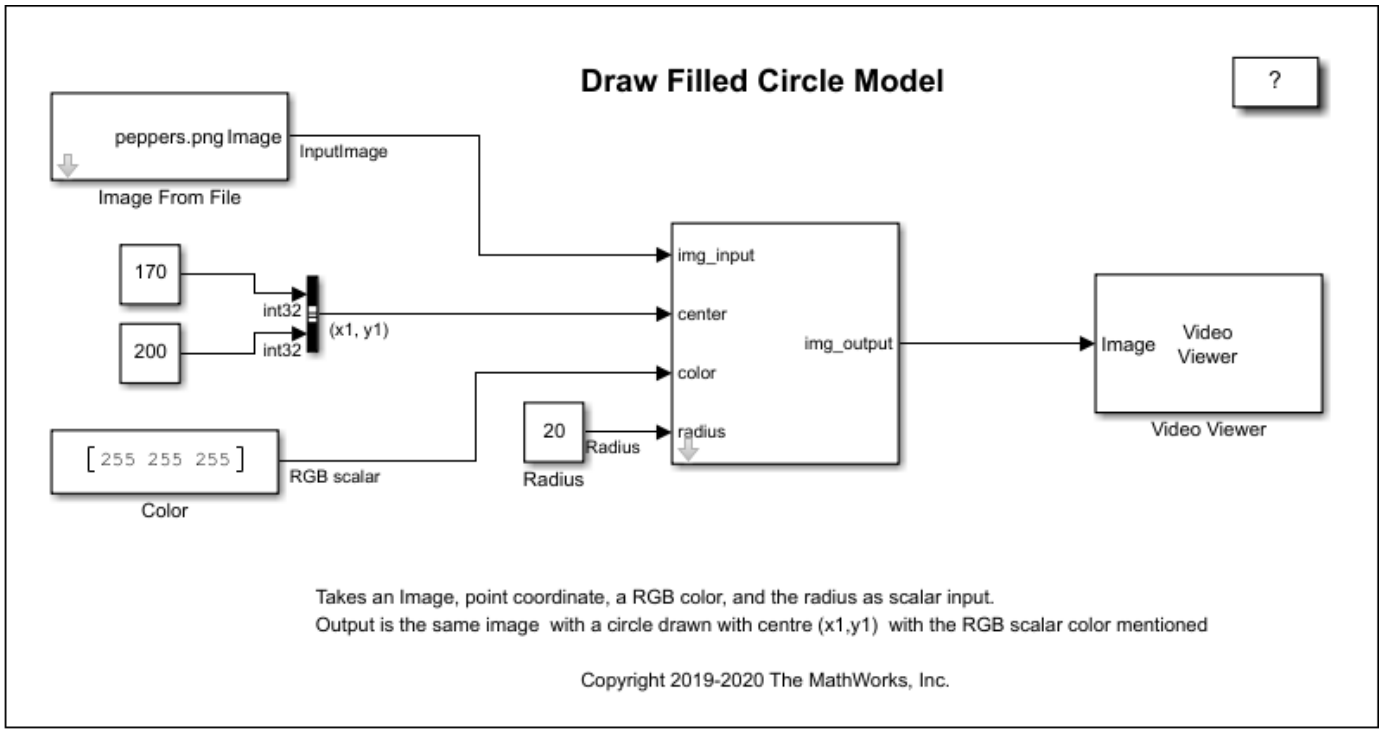
To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4. Configure the identified compiler by using the mex -setup c++ command. For more information, see “Choose a C++ 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**. Select the `drawFilledCircle` function and click **Next**.

4. From What to import, select the `I/O Type` for `img` as `InputOutput`, and other arguements 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.

6. Specify **Default CV::Mat Output Size** as `384,512,3`.

7. 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.

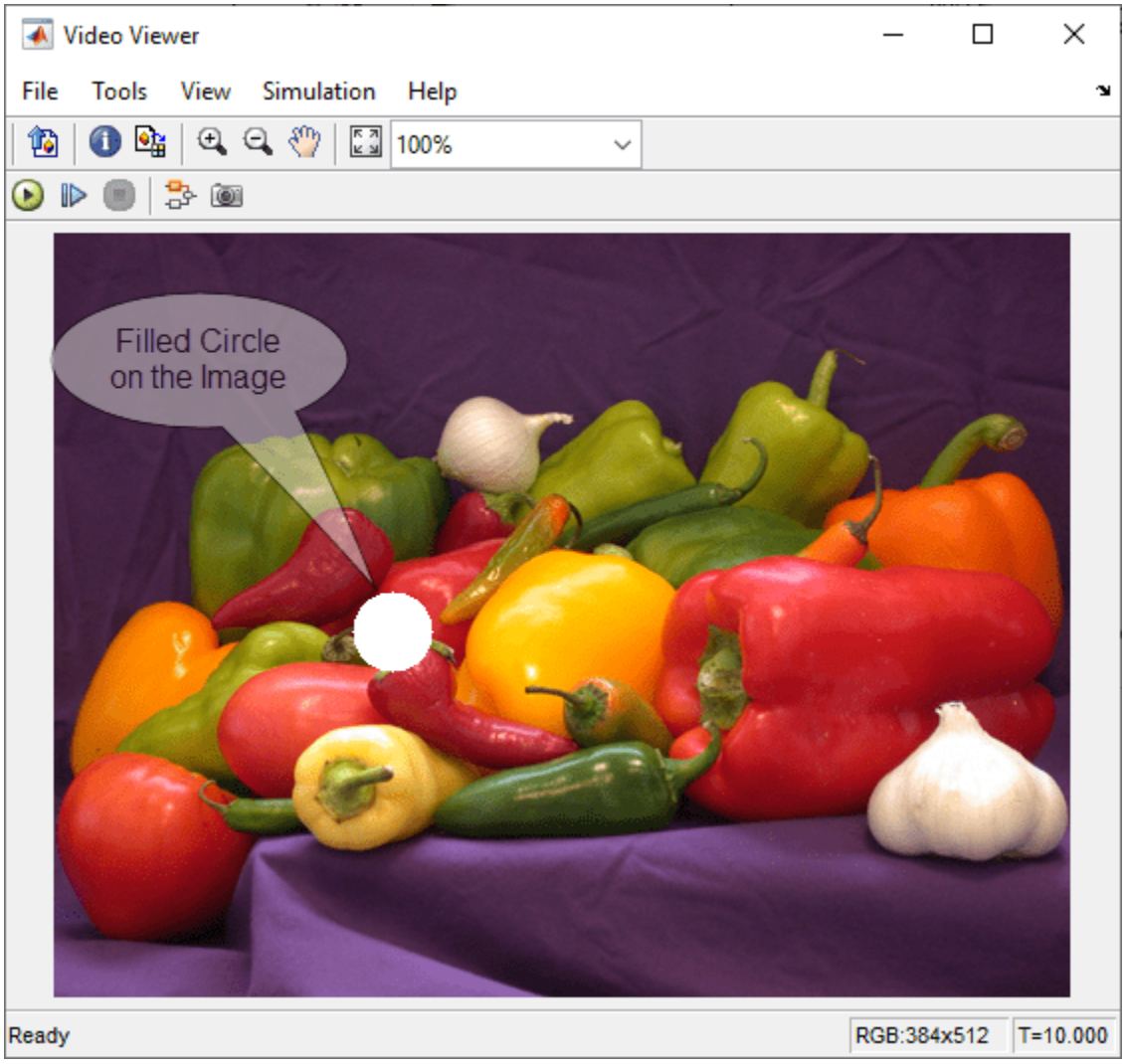
Step 2: Use Generated Subsystem in Simulink Model

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 subsystem to the MATLAB Function blocks.

2. Double-click the subsystem and verify the parameter values.

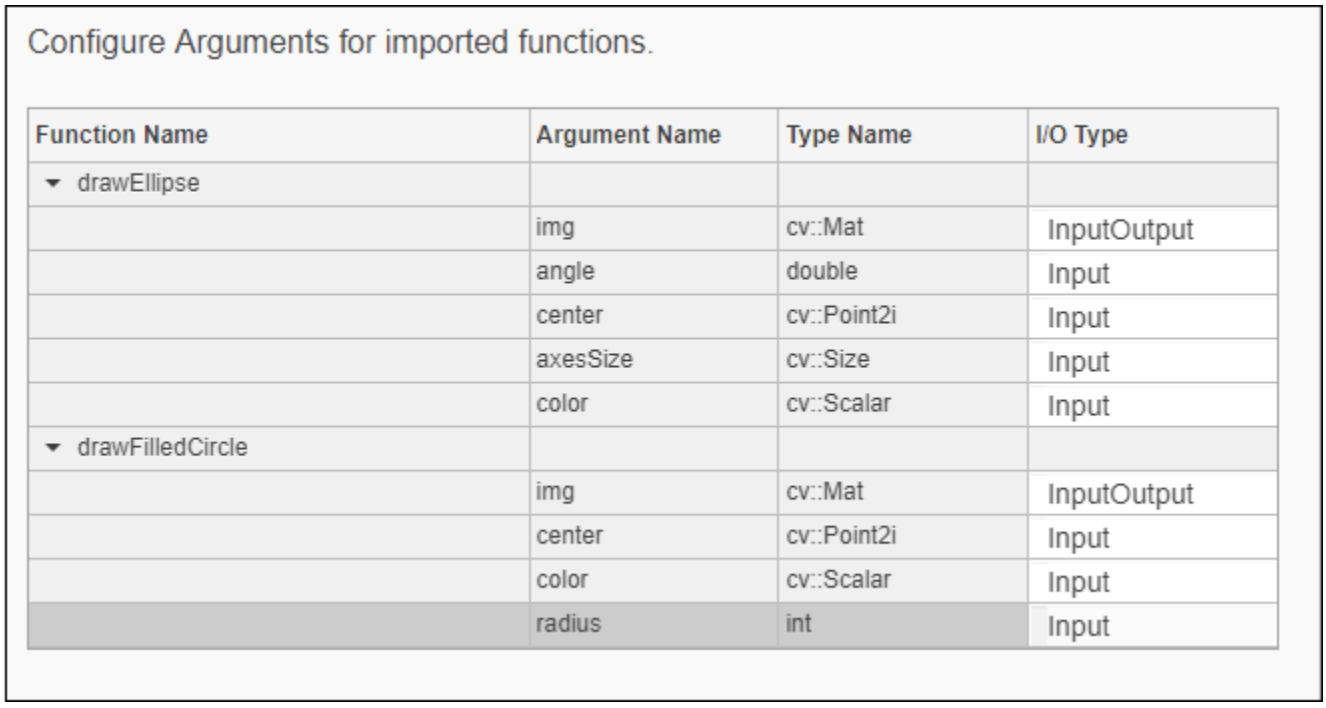
On the Simulink Toolstrip, in the **Simulation** tab, click on simulate the model button. After the simulation is complete, the Video Viewer block displays the filled circle on the input image `peppers.png`.



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.

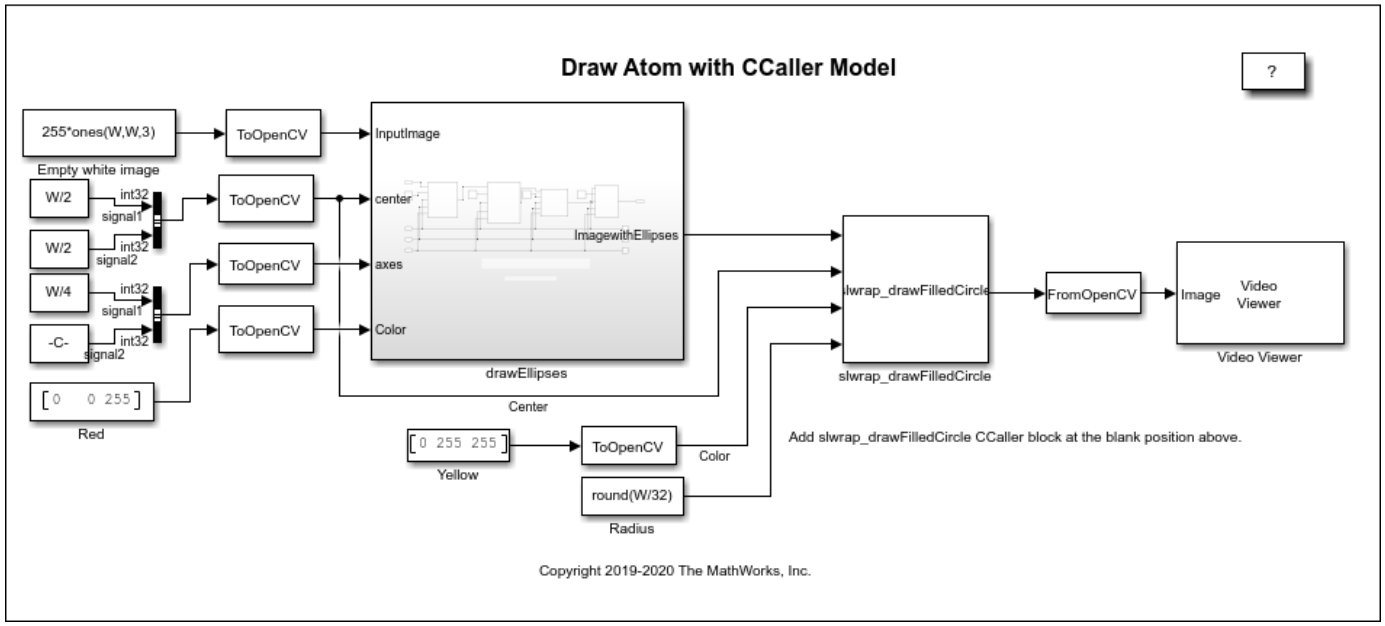
1. Import `drawEllipse` and `drawFilledCircle` OpenCV functions into Simulink by using the **OpenCV Importer** app. During import, select the `I/O Type` for `drawEllipse` and `drawFilledCircle`, as shown in this graphic.



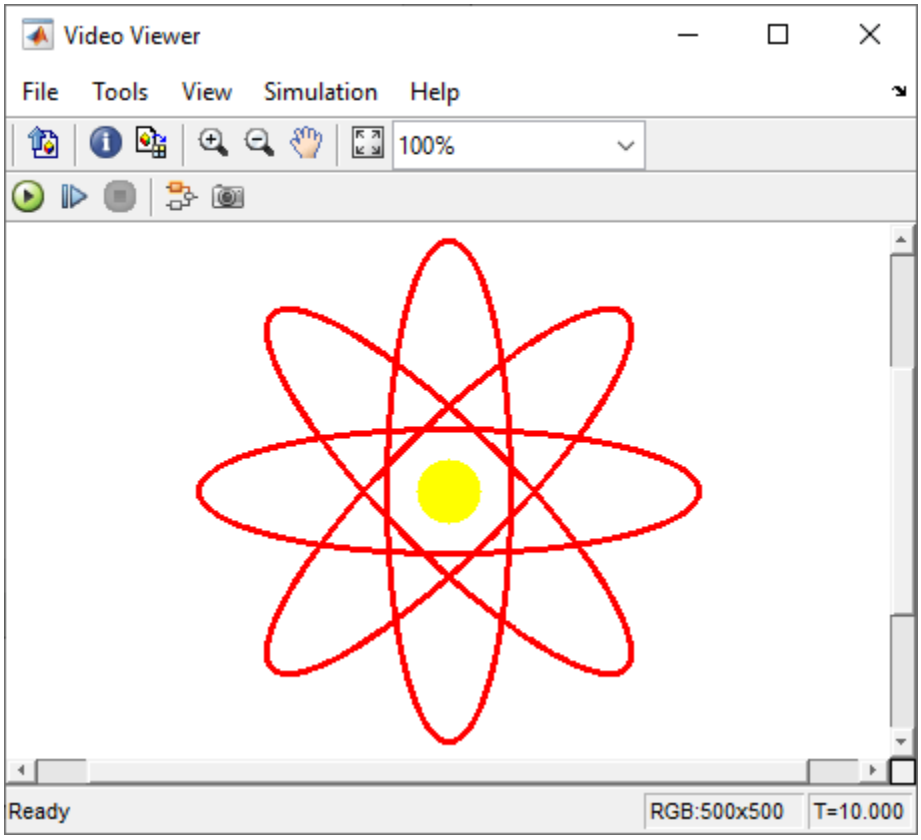
2. Once you import the functions, the `DrawCircle_Lib` library is created. This Simulink library contains subsystems and the C Caller blocks required to draw an atom on an image.

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 on simulate the model button. After the simulation is complete, the Video Viewer block displays the atom on a white input image.



See Also

ToOpenCV | FromOpenCV

More About

- Smile Detection by Using OpenCV Code in Simulink on page 13-45
- Convert RGB Image to Grayscale Image by Using OpenCV Importer on page 13-38

Convert RGB Image to Grayscale Image by Using OpenCV Importer

This example shows how to convert an RGB image to a grayscale image by using the **OpenCV Importer** app. 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 “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 13-24. The app 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.

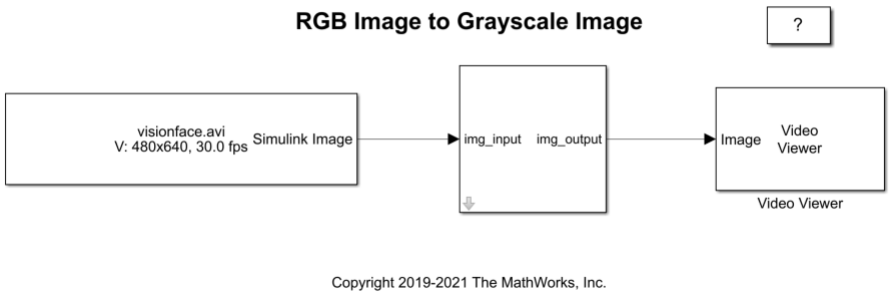
Set Up Your C++ Compiler

To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4. 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.

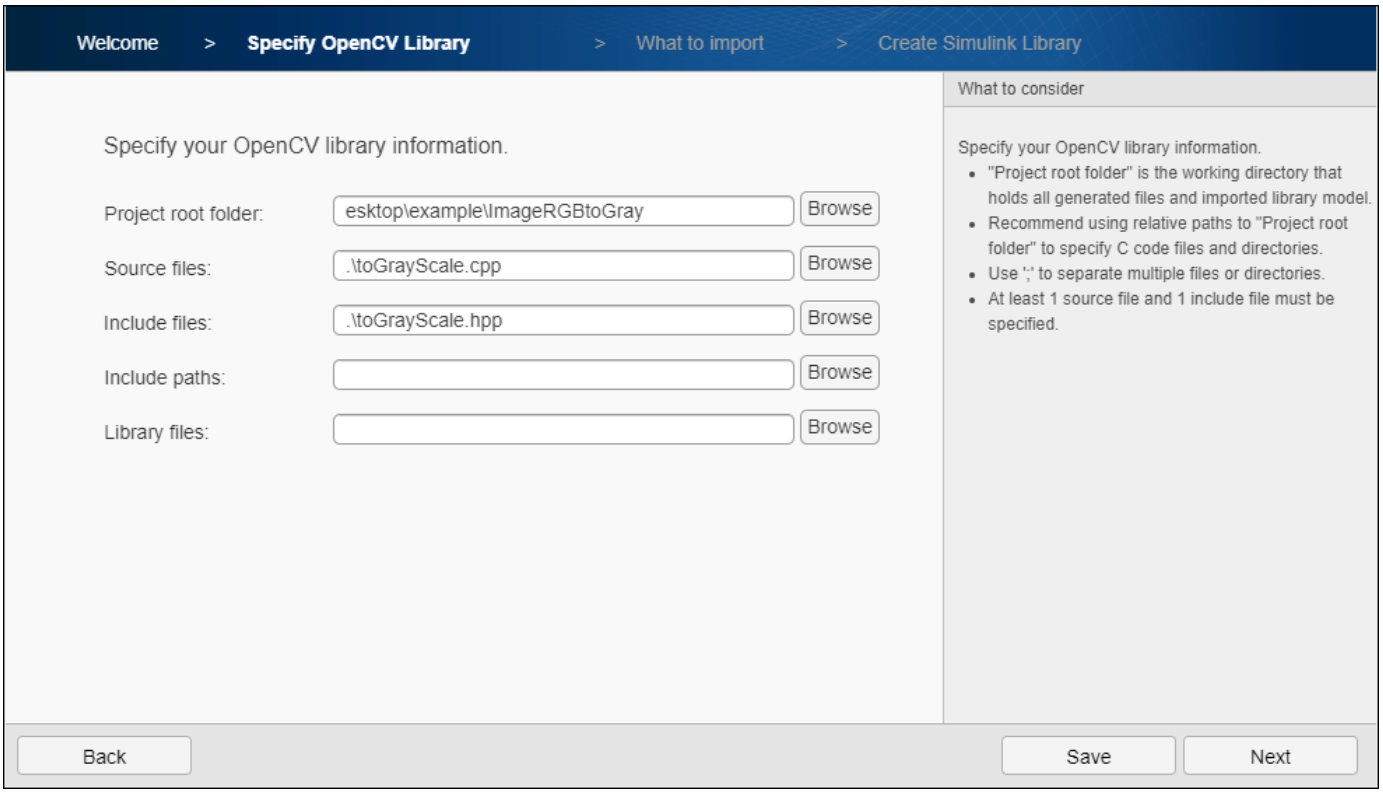
Before proceeding with these steps, ensure that you copy the example folder to a writable folder location and change your current working folder to `...example\ImageRGBtoGray`. All your output files are saved to this folder.

Step 1: Import OpenCV Function to Create a Simulink Library

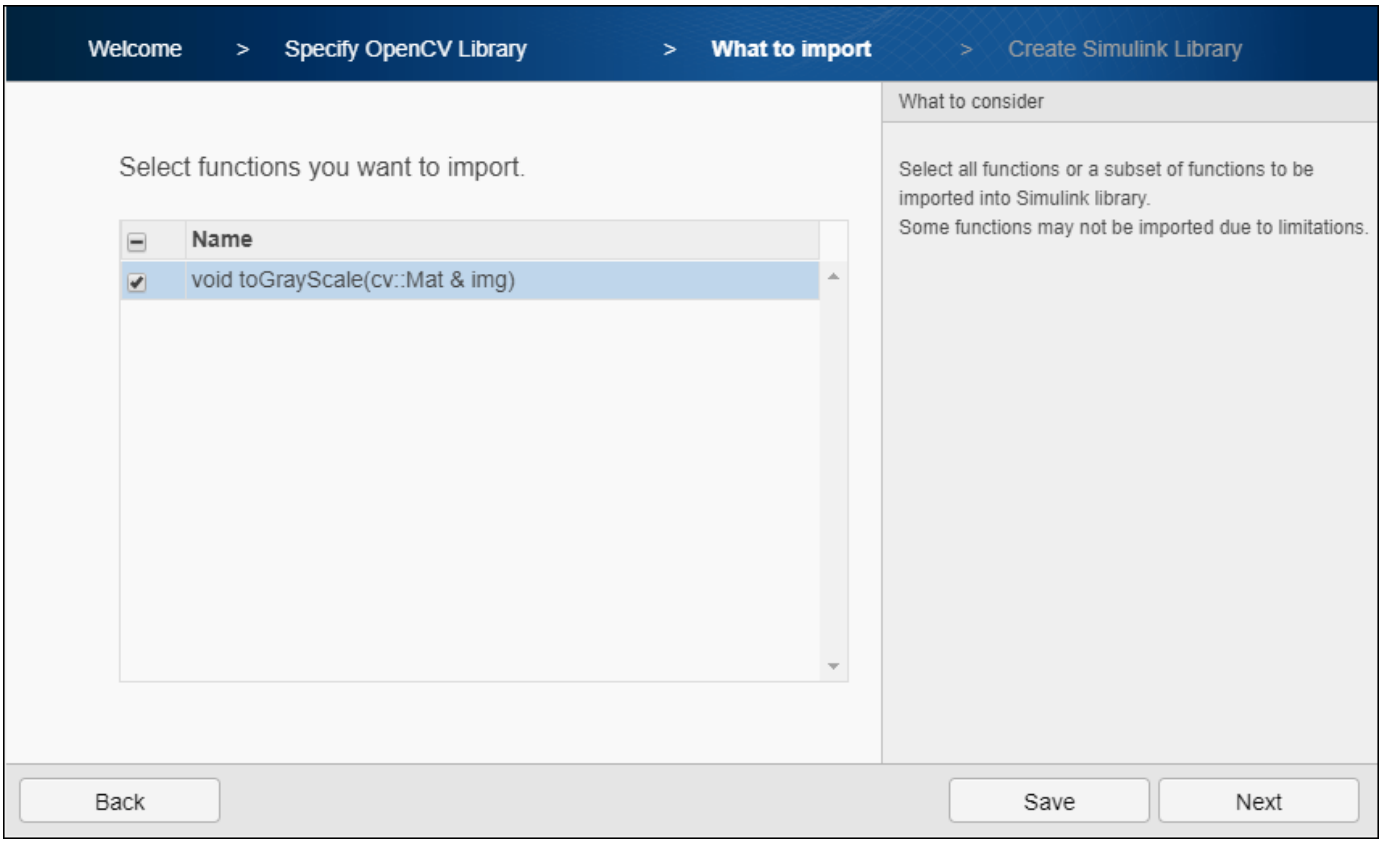
1. To start the **OpenCV Importer** app, click **Apps** on the MATLAB Toolstrip. In the Welcome page, specify the **Project name** as `RGBtoGRAY`. Make sure that the project name does not contain any spaces. Click **Next**.

2. In Specify OpenCV Library, specify these file locations, and then click **Next**.

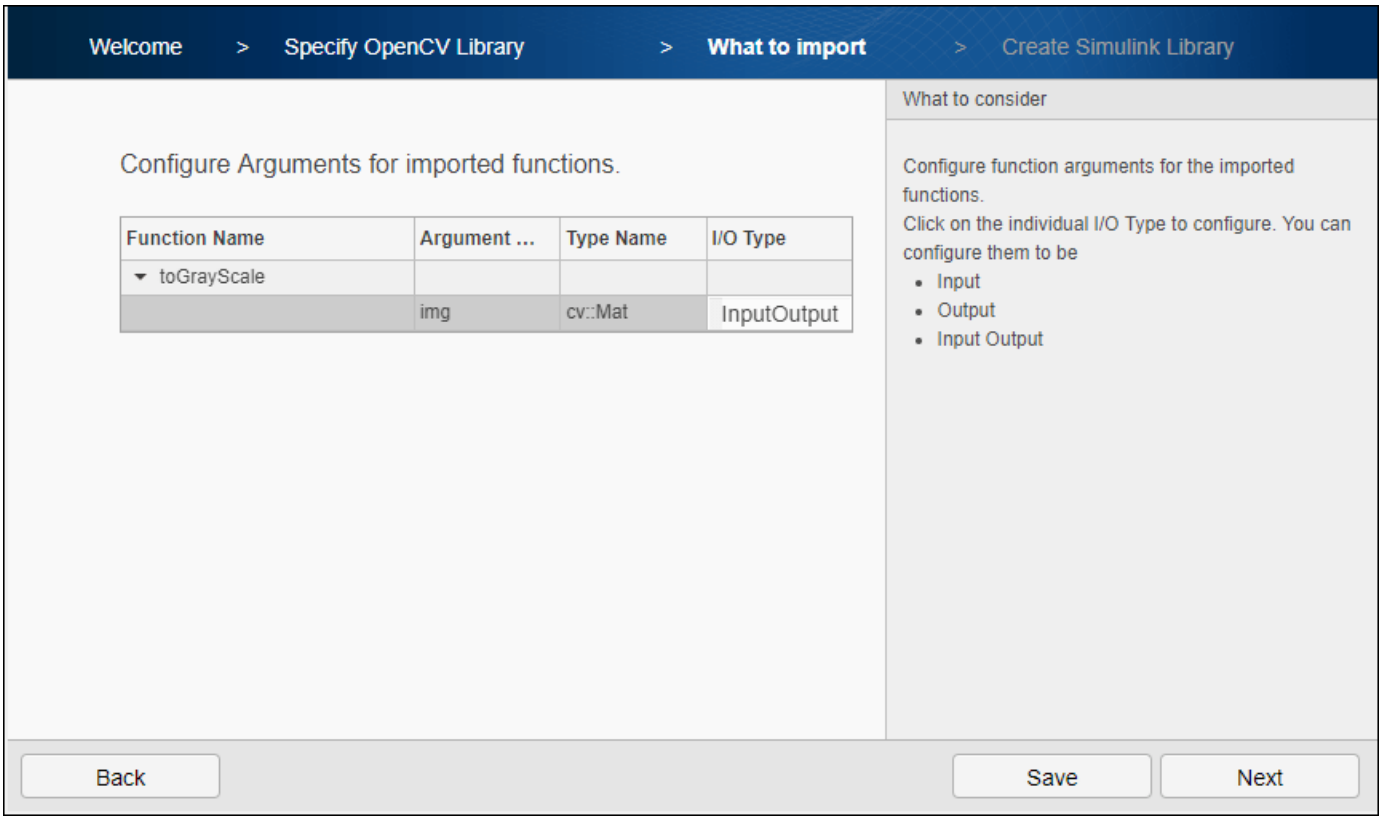
- **Project root folder**: Specify the path of your example folder. This path is the path to the writable project folder where you have saved your example files. All your output files are saved to this folder.
- **Source files**: Specify the path of the `.cpp` file located inside your project folder 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, select the **I/O Type** for `img` as `InputOutput`, and then click **Next**.

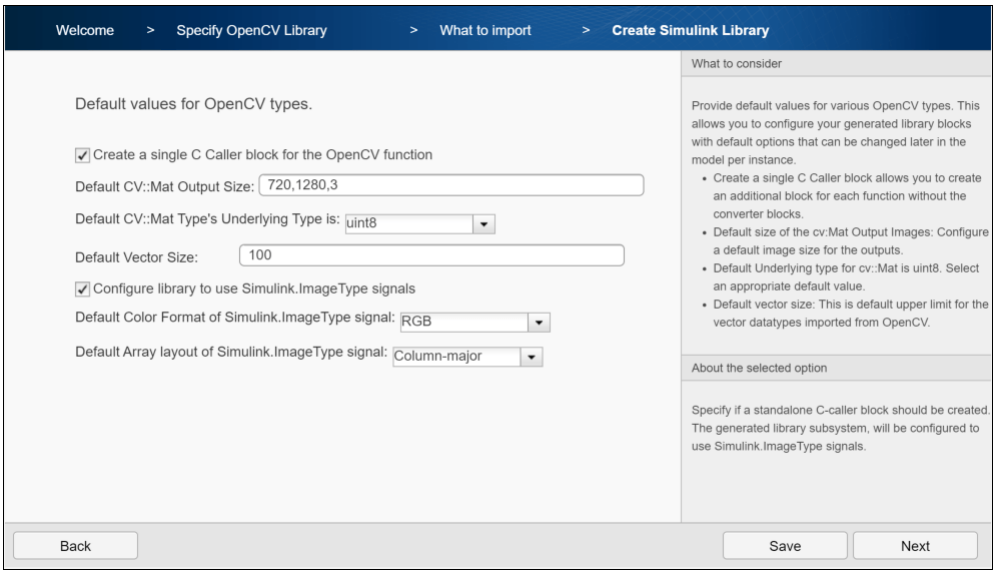


5. In Create Simulink Library, configure the default values of OpenCV types. By default, **Create a single C-caller block for the OpenCV function** is selected to create a C Caller block along with the subsystem in the generated Simulink library.

6. Select **Configure library to use Simulink.ImageType signals** to configure the generated library subsystem to use `Simulink.ImageType` signals.

7. Set **Default Color Format of Simlink.ImageType signal** to `RGB`, which is the default color format of the image.

8. Set **Default Array layout of Simulink.ImageType signal** to `Column-major`, which is the default array layout of the image.



9. To create a Simulink library, click **Next**.

A Simulink library `RGBtoGRAY_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_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 from the library to the model. Insert the subsystem between From Multimedia File block and Video Viewer block.

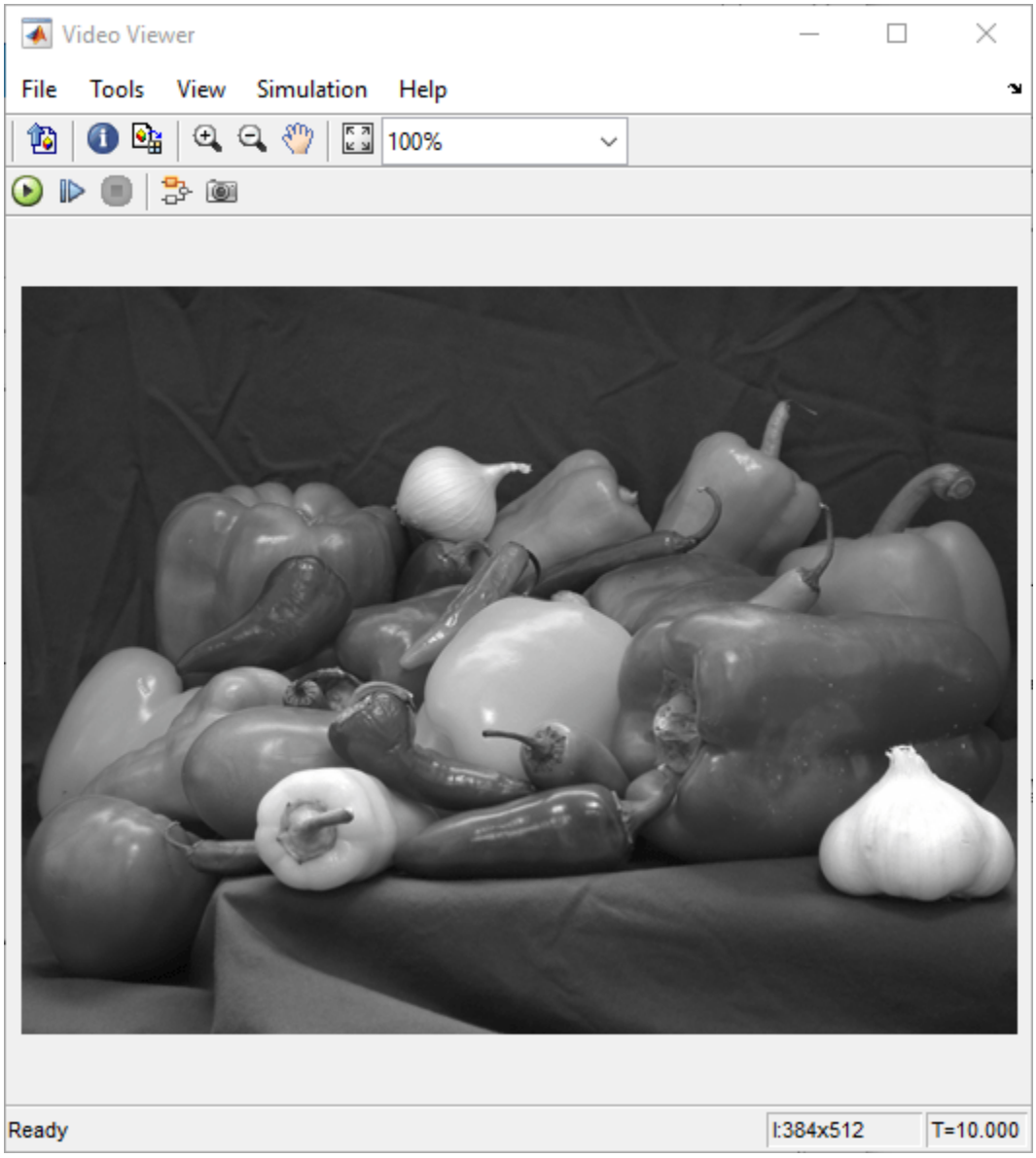
2. Double-click the subsystem and configure these parameter values:

- **Rows**: `480`
- **Columns**: `640`
- **Channels**: `1`
- **Underlying Type**: `uint8`

3. Click **Apply**, and then click **OK**.

Step 3: Simulate the RGB to Gray Convertor

On the Simulink Toolstrip, in the **Simulation** tab, click on **Run** 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

ToOpenCV | FromOpenCV | `Simulink.ImageType`

More About

- Smile Detection by Using OpenCV Code in Simulink on page 13-45
- Draw Different Shapes by Using OpenCV Code in Simulink on page 13-31

Smile Detection by Using OpenCV Code in Simulink

This example shows how to build a smile detector by using the **OpenCV Importer** app. The detector estimates the intensity of the smile on a face image or a video. 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 following the "Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink" on page 13-24. The app 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 facial image or a video 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.

Set Up Your C++ Compiler

To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in "Portable C Code Generation for Functions That Use OpenCV Library" on page 22-4. Configure the identified compiler by using the `mex -setup c++` command. For more information, see "Choose a C++ Compiler".

Model Description

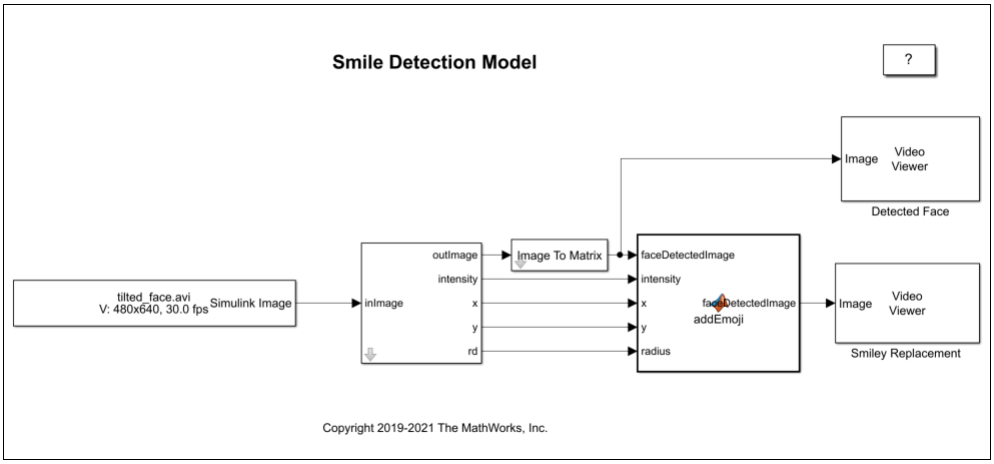
In this example, a smile detector is implemented by using the Simulink model `smileDetect.slx`.

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** app. The subsystem accepts a face image or a video and provides these output values.

- `outImage`: 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 model is configured to use `Simulink.ImageType` datatype. The `outImage` from the subsystem is of `Simulink.ImageType` datatype. The Image To Matrix block converts `outImage` from `Simulink.ImageType` to a numerical matrix because a MATLAB Function block operates on numerical matrixes only.

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 Detected Face and Smiley Replacement Video Viewer blocks.



Copy Example Folder to a Writable Location

To access the path to the example folder, at the MATLAB command line, enter:

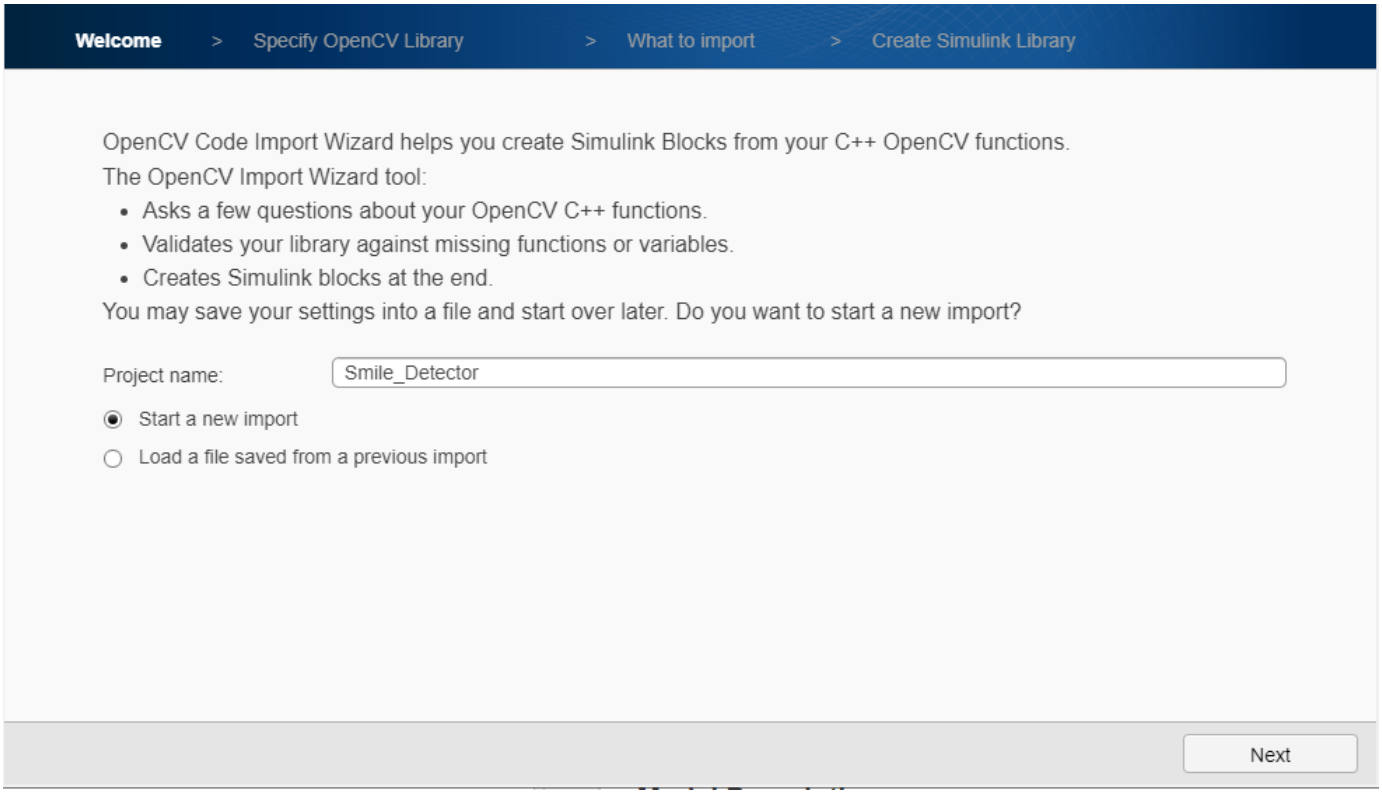
```
OpenCVSimulinkExamples;
```

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.

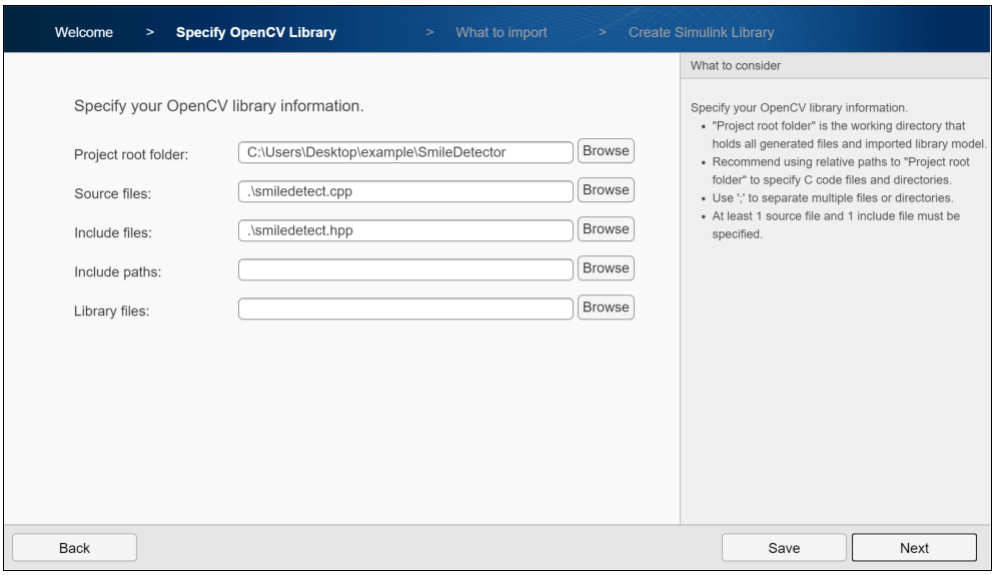
Step 1: Import OpenCV Function to Create a Simulink Library

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**.



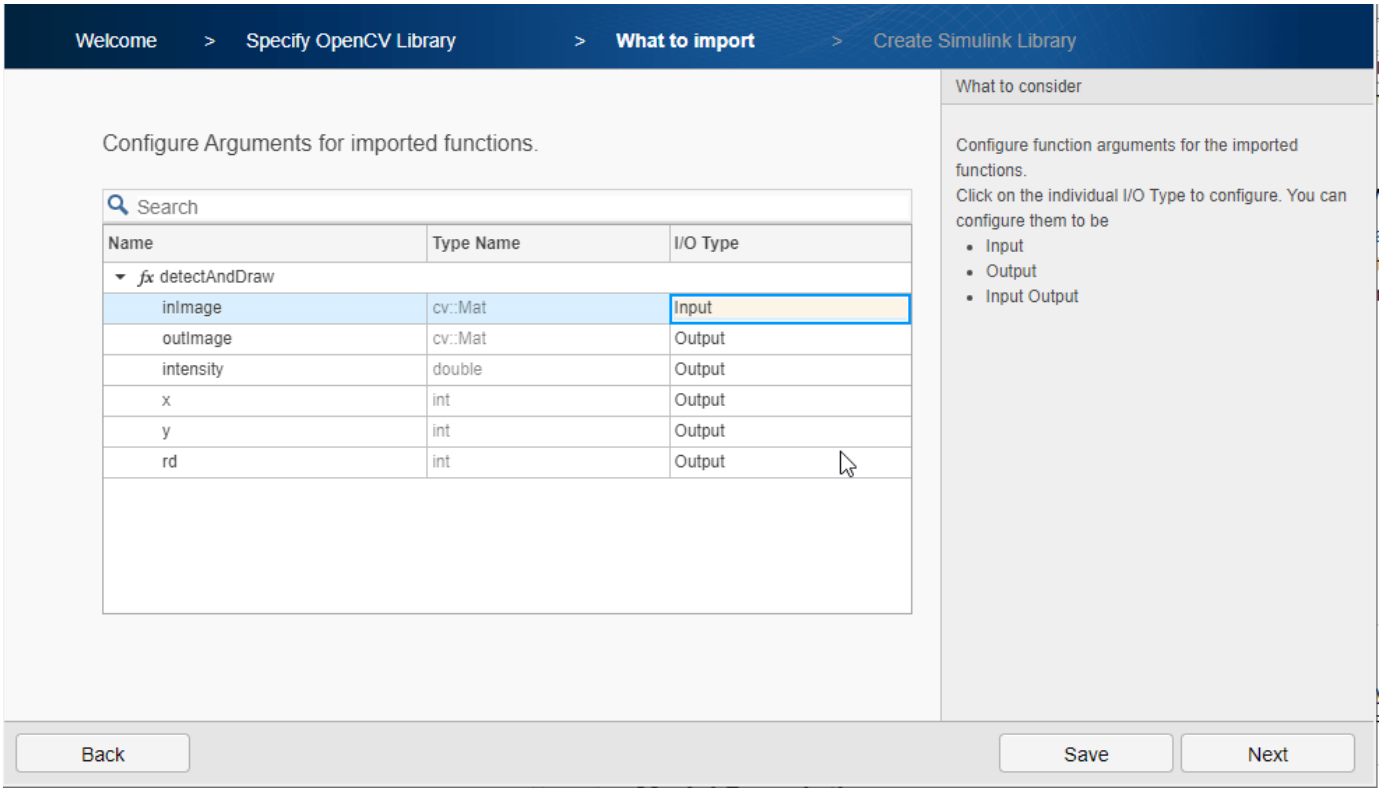
2. In Specify OpenCV Library, specify these file locations, and then click **Next**.

- **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 `inImage` as `Input`, and then click **Next**.

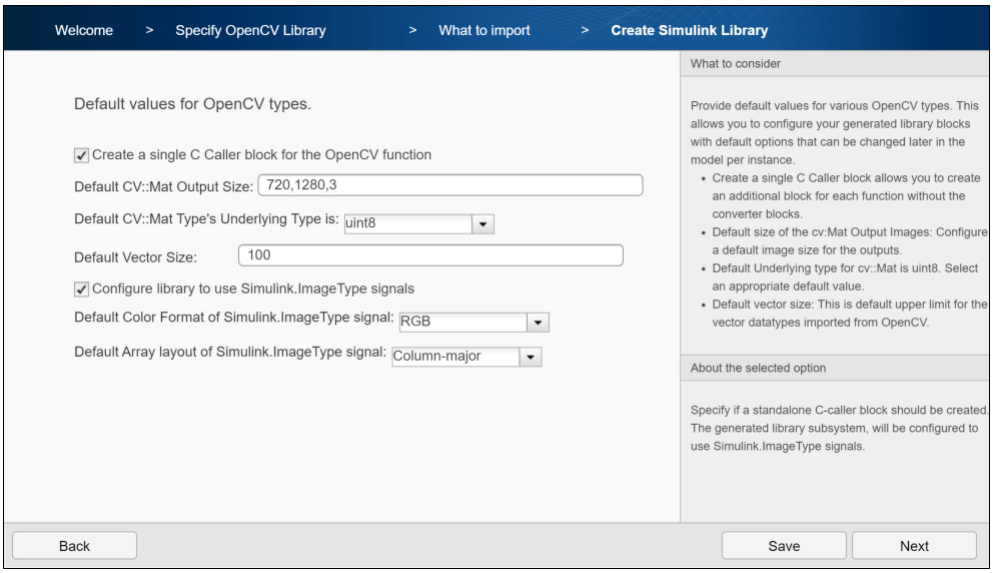


5. In Create Simulink Library, configure the default values of OpenCV types. By default, **Create a single C-caller block for the OpenCV function** is selected to create a C Caller block along with the subsystem in the generated Simulink library.

6. Select **Configure library to use Simulink.ImageType signals** to configure the generated library subsystem to use `Simulink.ImageType` signals.

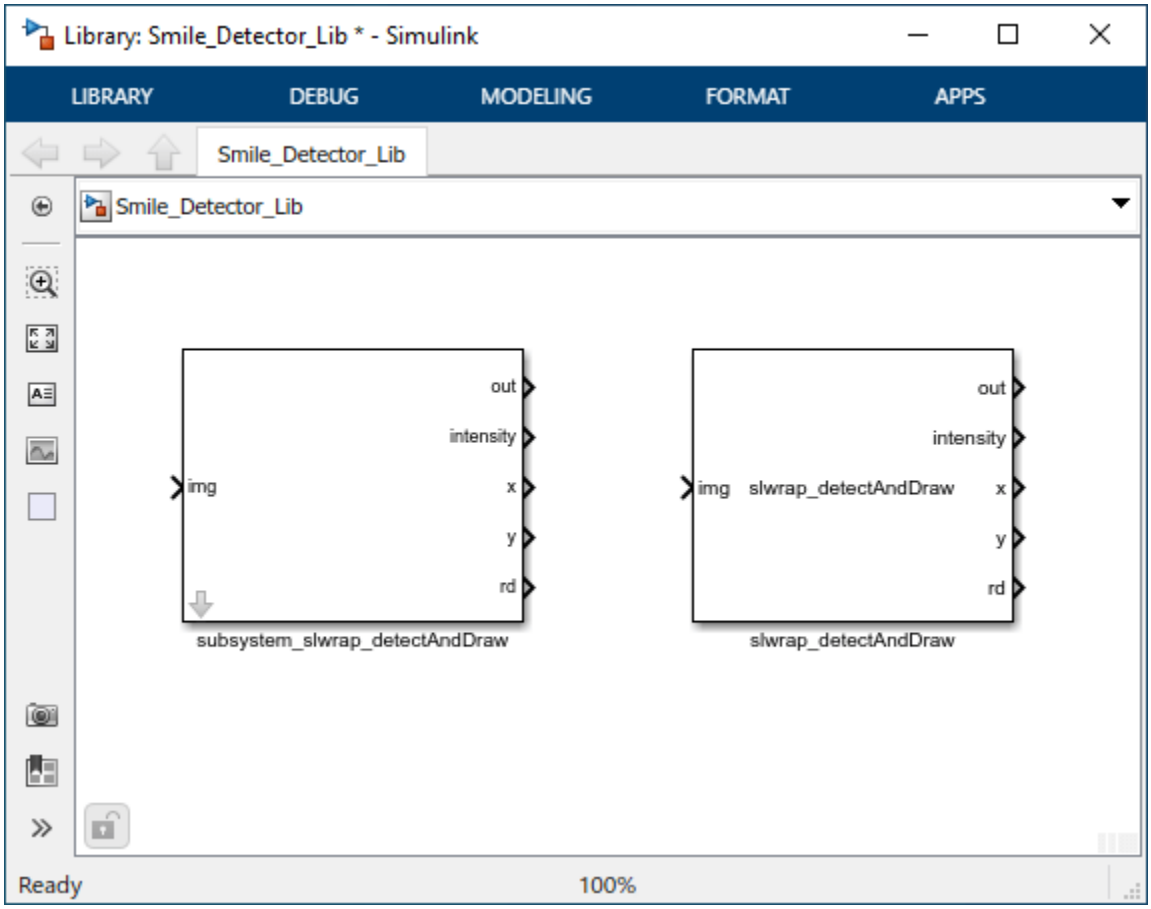
7. Set **Default Color Format of Simlink.ImageType signal** to `RGB`, which is the default color format of the image.

8. Set **Default Array layout of Simulink.ImageType signal** to `Column-major` which, is the default array layout of the image.



9. To create a Simulink library, 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. Drag the generated subsystem from the library to the model. Connect the subsystem to the MATLAB Function block.

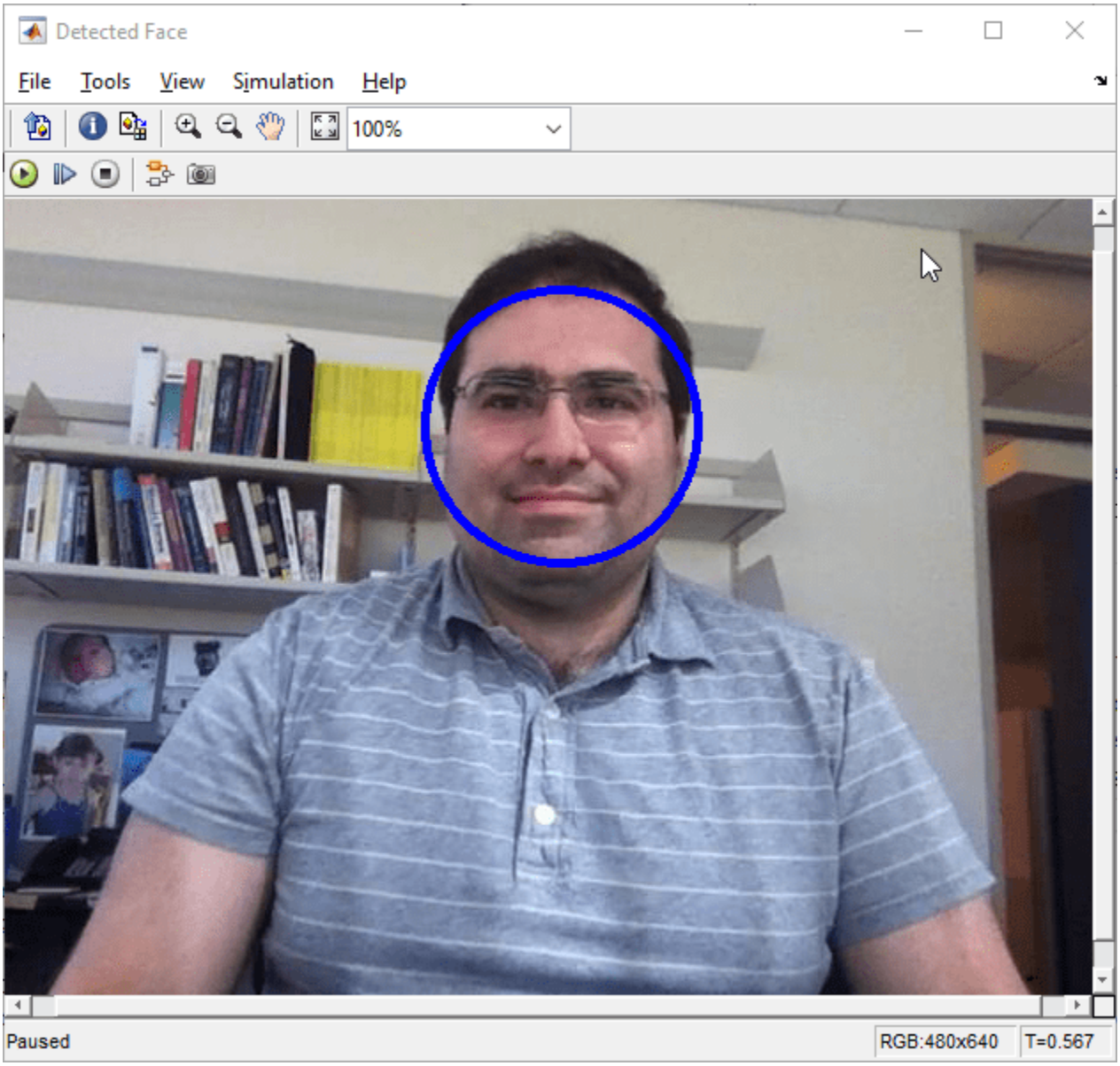
2. Double-click the subsystem and configure these parameter values:

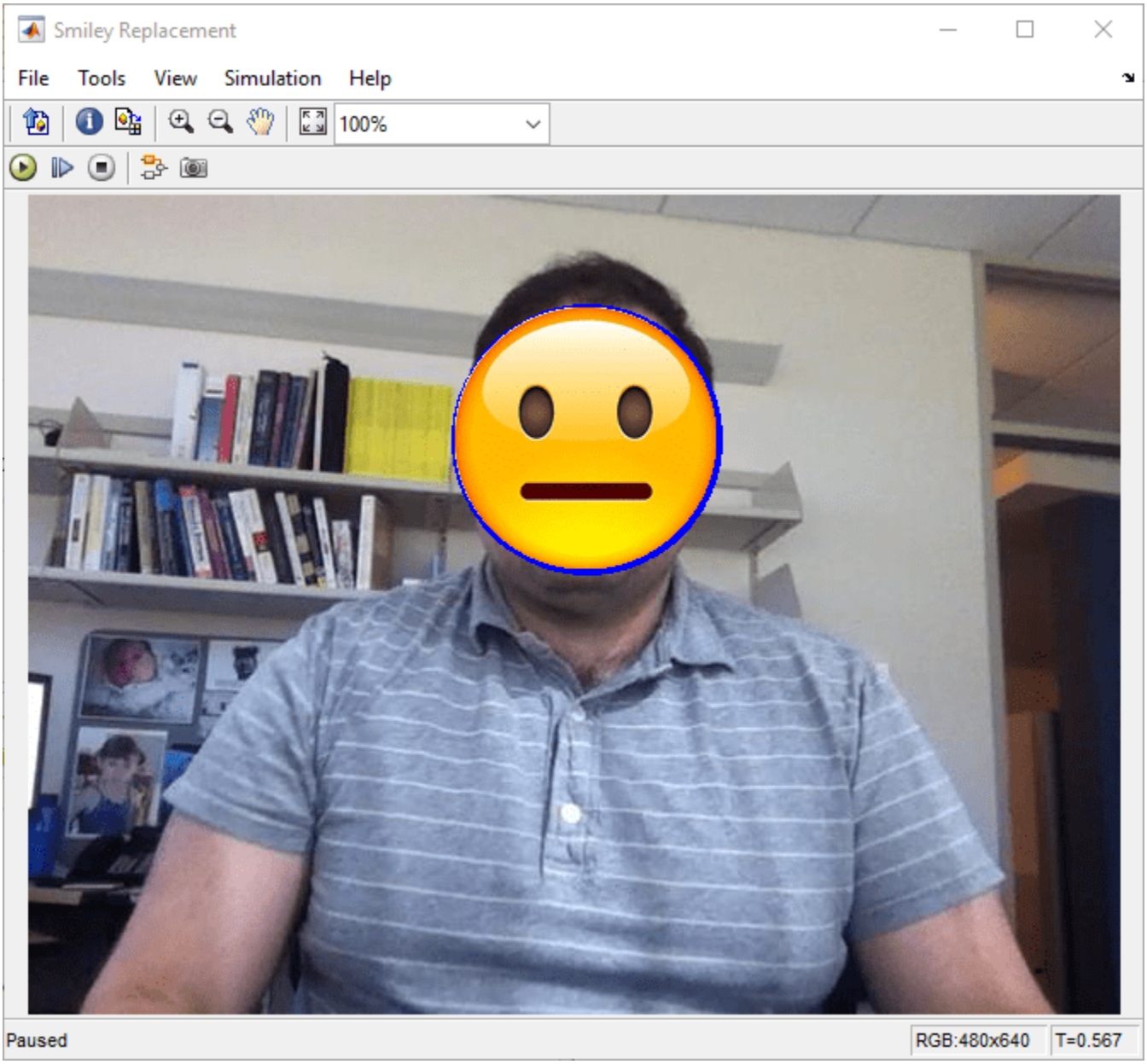
- **Rows**: `480`
- **Columns**: `640`
- **Channels**: `3`
- **Underlying Type**: `uint8`

3. Click **Apply**, and then click **OK**.

Step 3: Simulate the Smile Detector

On the Simulink Toolstrip, in the **Simulation** tab, click on **Run** to simulate the model. After the simulation is complete, the Video Viewer blocks display the face detected. The model overlays an emoji on the face. The emoji represents the intensity of the smile.


Detected Face
File
Tools
View
Simulation
Help
100%
Paused
RGB:480x640
T=0.567

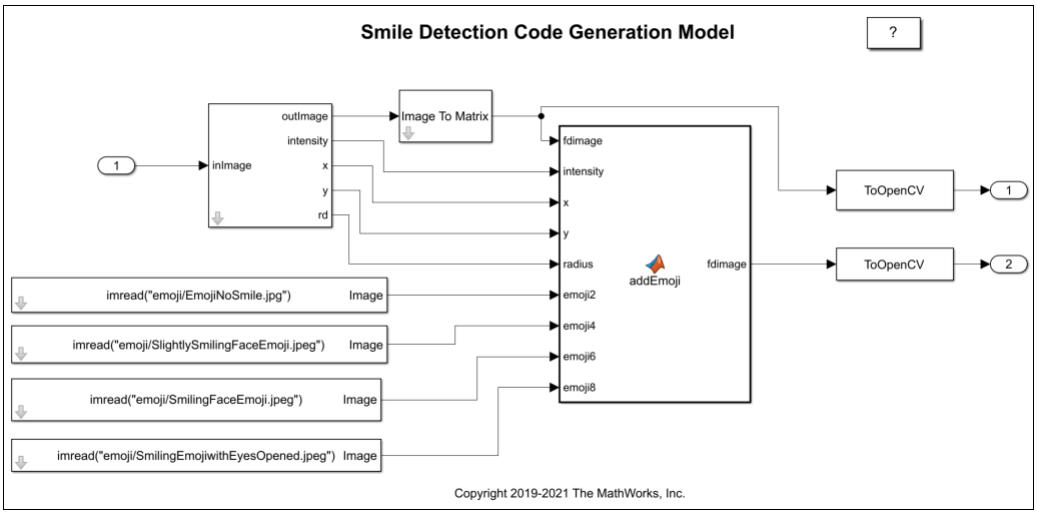


Step 4: Generate C++ Code from the Smile Detector Model

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 current MATLAB folder.



2. On the **Apps** tab on the Simulink toolstrip, select **Embedded Coder**. On the **C++ Code** tab, select the **Settings** list, then click **C/C++ Code generation settings** to open the Configuration Parameters dialog box. Verify these settings:

- Under the **Code Generation** pane > in the **Target selection** section > **Language** is set to `C++`.
- Under the **Code Generation** pane > in the **Target selection** section > **Language standard** is set to `C++11 (ISO)`.
- Under the **Code Generation** pane > **Interface** > in the **Data exchange interface** section > **Array layout** is set to `Row-major`.

3. If you want to generate production C++ code, where images are represented using the OpenCV class `cv::Mat` instead of the C++ class `images::datatypes::Image` implemented by The MathWorks®, under **Data Type Replacement** pane > select **Implement images using OpenCV Mat class**.

4. Connect the generated subsystem `subsystem_slwrap_detectAndDraw` to the MATLAB Function block.

5. To generate C++ code, under the **C++ Code** tab, click the `Build` button. After the model finishes building, the generated code opens in the **Code** view.

6. You can inspect the generated code. When a model contains signals of `Simulink.ImageType` data type, the code generator produces additional shared utility files. These files declare and define utilities to construct, destruct, and return information about meta attributes of the images:

- `image_type.h`
- `image_type.cpp`

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

ToOpenCV | FromOpenCV | `Simulink.ImageType`

More About

- Convert RGB Image to Grayscale Image by Using OpenCV Importer on page 13-38
- Draw Different Shapes by Using OpenCV Code in Simulink on page 13-31

Shadow Detection by Using OpenCV Code in Simulink

This example shows how to detect shadows in a recorded video by using the **OpenCV Importer** app.

First import an OpenCV function into Simulink® by following the “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 13-24. 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 recorded video and a threshold value for shadow detection. The shadow detection works per the threshold value. The threshold value can be changed using the slider component available in the model.

You learn how to:

- Import an OpenCV function into a Simulink library.
- Use blocks from a generated library in a Simulink model.

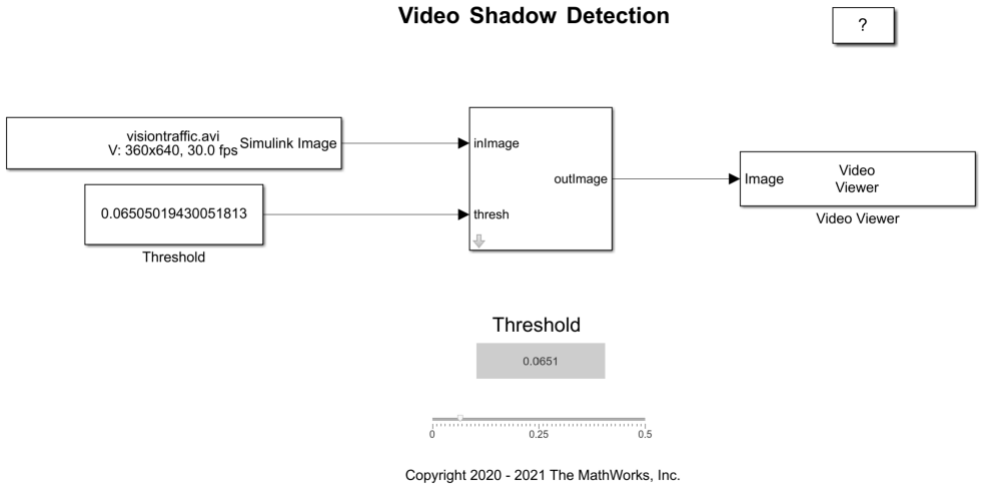
Set Up Your C++ Compiler

To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4. 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 shadow detector is implemented by using the Simulink model `ShadowDetection.slx`.

In this model, the `subsystem_slwrap_run_shadow_detection` subsystem resides in the `Shadow_Detection_Lib` library. You create the `subsystem_slwrap_run_shadow_detection` subsystem by using the **OpenCV Importer** app. The subsystem accepts a video from the From Multimedia File block and a threshold value to detect the shadow in the video. The output is displayed using the Video Viewer block. In the `subsystem_slwrap_run_shadow_detection` subsystem, `inImage` is the input image, `thresh` is the threshold of the algorithm and `outImage` is the output image. The threshold slider is used to change the threshold value during the simulation.



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\ShadowDetection`. All your output files are saved to this folder.

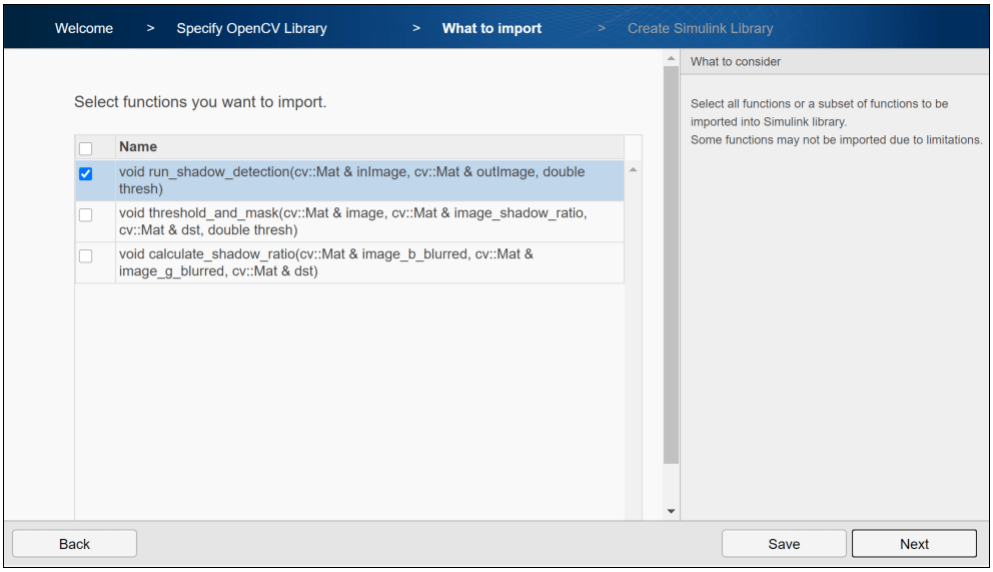
Step 1: Import OpenCV Function to Create a Simulink Library

1. To start the **OpenCV Importer** app, click **Apps** on the MATLAB Toolstrip. In the Welcome page, specify the **Project name** as `Shadow_Detection`. Make sure that the project name does not contain any spaces. Click **Next**.

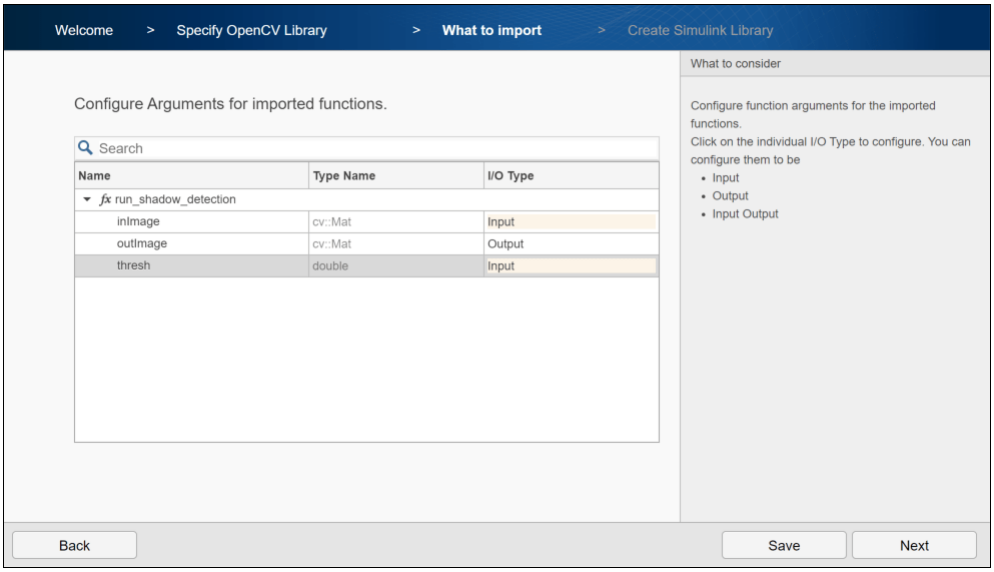
2. In Specify OpenCV Library, specify these file locations, and then click **Next**.

- **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 `shadow_detection.cpp`.
- **Include files** : Specify the path of the `.hpp` header file located inside your project folder as `shadow_detection.hpp`.

3. Analyze your library to find the functions and types for import. Once the analysis is complete, click **Next**. Select the `run_shadow_detection` function and click **Next**.



4. From What to import, select the **I/O Type** for `inImage` and `thresh` as `Input`, `outImage` as Output and then click **Next**.

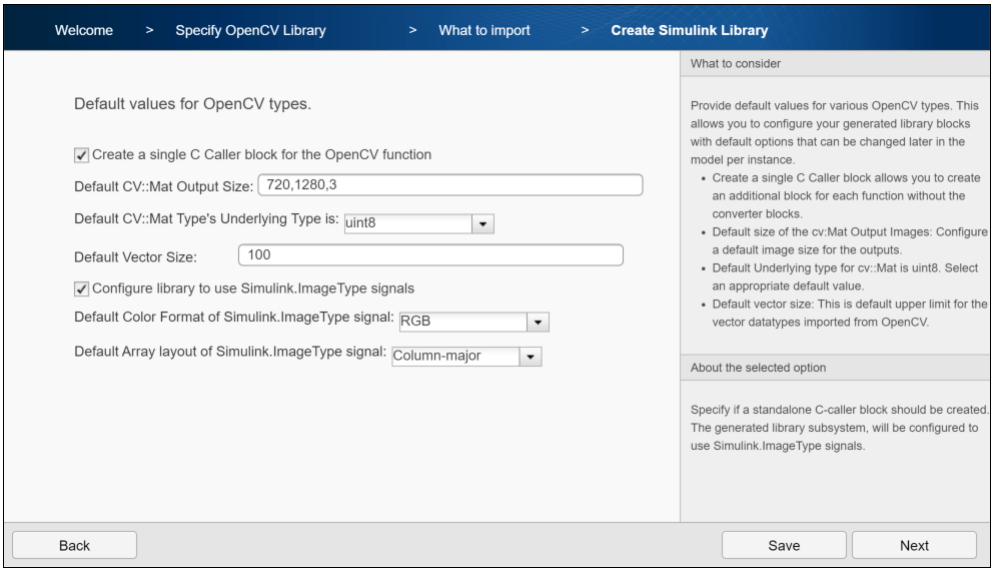


5. In Create Simulink Library, configure the default values of OpenCV types. By default, **Create a single C-caller block for the OpenCV function** is selected to create a C Caller block along with the subsystem in the generated Simulink library.

6. Select **Configure library to use Simulink.ImageType signals** to configure the generated library subsystem to use `Simulink.ImageType` signals.

7. Set **Default Color Format of Simlink.ImageType signal** to `RGB`, which is the default color format of the image.

8. Set **Default Array layout of Simulink.ImageType signal** to `Column-major`, which is the default array layout of the image.



9. To create a Simulink library, click **Next**.

A Simulink library `Shadow_Detection_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_run_shadow_detection` is used.

Step 2: Use Generated Subsystem in Simulink Model

To use the generated subsystem `subsystem_slwrap_run_shadow_detection` with the Simulink model `ShadowDetection.slx`:

1. In your MATLAB current folder, right-click the model `ShadowDetection.slx` and click **Open** from the context menu. Drag the generated subsystem from the library to the model. Insert the subsystem between From Multimedia File block and Constant block and Video Viewer block.

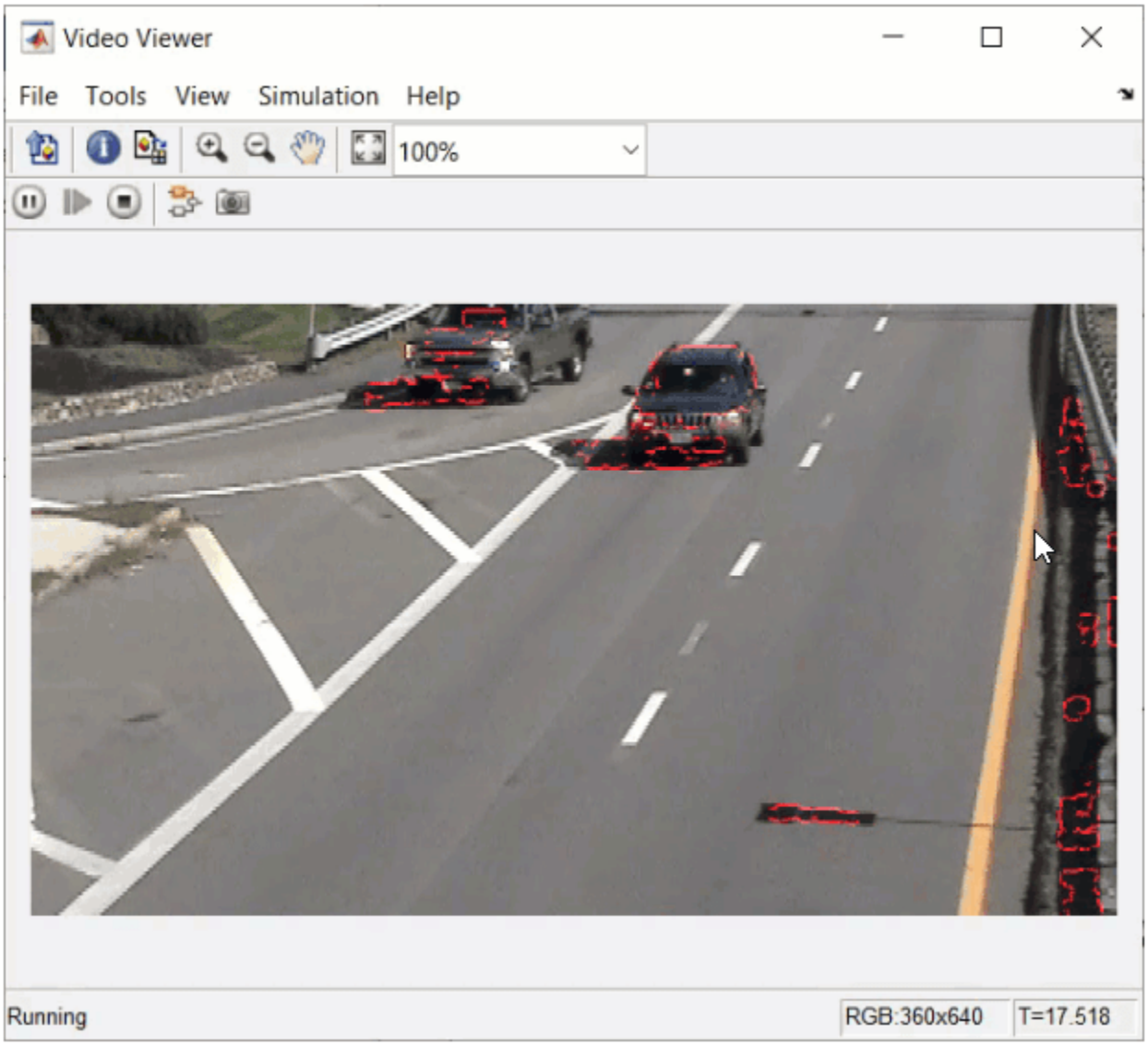
2. Double-click the subsystem and configure these parameter values:

- **Rows**: `360`
- **Columns**: `640`
- **Channels**: `3`
- **Underlying Type**: `uint8`

3. Click **Apply**, and then click **OK**.

Step 3: Simulate the Shadow Detector

On the Simulink Toolstrip, in the Simulation tab, click on **Run** to simulate the model. After the simulation is complete, the Video Viewer block displays a video with shadows marked with a red outline depending on the threshold value. For the displayed example video, threshold value is set to `0.0651`.



See Also

ToOpenCV | FromOpenCV | `Simulink.ImageType`

More About

- Vehicle and Pedestrian Detector by Using OpenCV Code in Simulink on page 13-60
- Video Cartoonizer by Using OpenCV Code in Simulink on page 13-64

Vehicle and Pedestrian Detector by Using OpenCV Importer

This example shows how to build a Vehicle and Pedestrian Detector by using the **OpenCV Importer** app. The model places green and red outline on the vehicle and pedestrians when detected in a recorded video.

First import an OpenCV function into Simulink® by following the “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 13-24. The app 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 recorded video for vehicle and pedestrian detection.

You learn how to:

- Import an OpenCV function into a Simulink library.
- Use blocks from a generated library in a Simulink model.

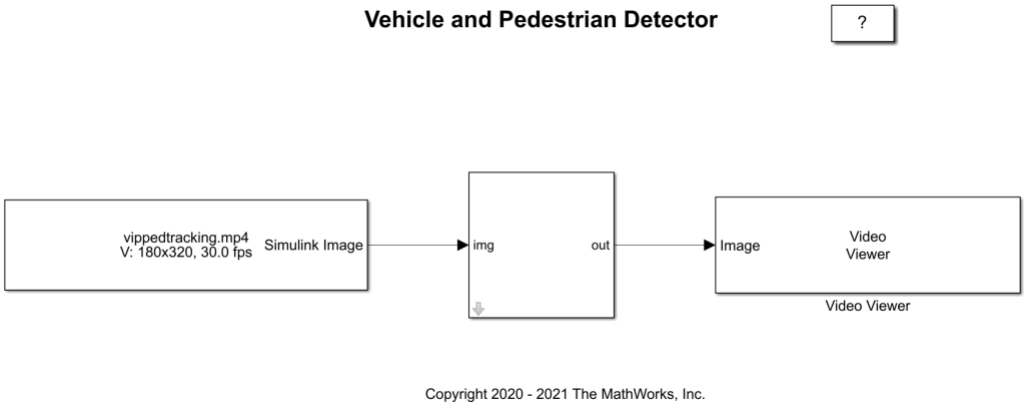
Set Up Your C++ Compiler

To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4. 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 vehicle and pedestrian detector is implemented by using the Simulink model `VehiclePedestrianDetector.slx`.

In this model, the `subsystem_slwrap_drawDetect` subsystem resides in the `Vehicle_Pedestrian_Detector_Lib` library. You create the `subsystem_slwrap_drawDetect` subsystem by using the **OpenCV Importer** app. The subsystem accepts a video from the From Multimedia File 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 \Vehicle_Pedestrian_Detector`. 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 `Vehicle_Pedestrian_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 `vehiclePedestrianDetector.cpp`.
- **Include files**: Specify the path of the `.hpp` header file located inside your project folder as `vehiclePedestrianDetector.hpp`.

3. Analyze your library to find the functions and types for import. Once the analysis is complete, click **Next**. Select the `drawDetect` function and click **Next**.

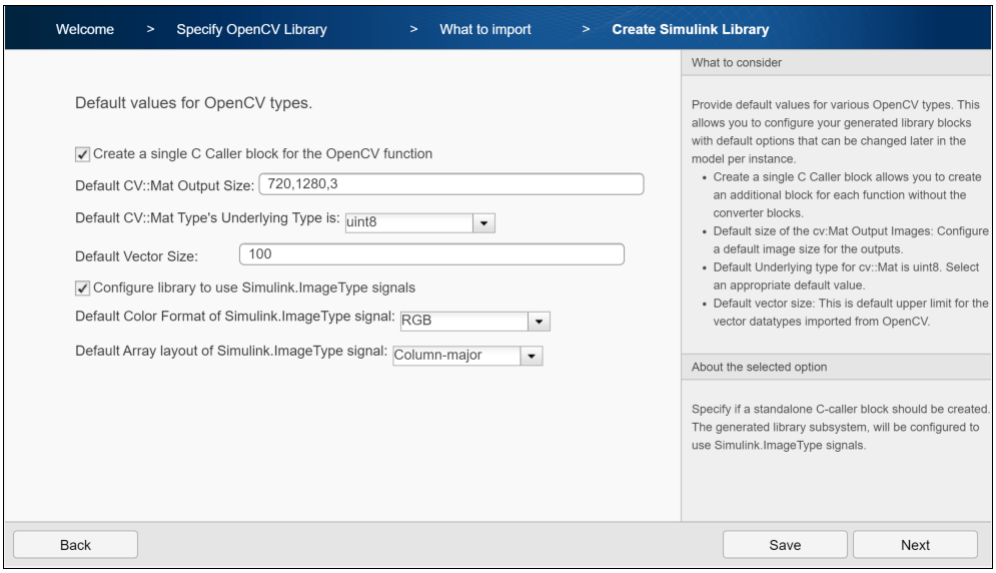
4. From What to import, select the **I/O Type** for `img` as `Input`, `out` as `Output` and then click **Next**.

5. In Create Simulink Library, configure 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 along with the subsystem in the generated Simulink library.

6. Select **Configure library to use Simulink.ImageType signals** to configure the generated library subsystem to use `Simulink.ImageType` signals.

7. Set **Default Color Format of Simlink.ImageType signal** to `RGB`, which is the default color format of the image.

8. Set **Default Array layout of Simulink.ImageType signal** to `Column-major`, which is the default array layout of the image.



9. To create a Simulink library, click **Next**.

A Simulink library `Vehicle_Pedestrian_Detector_Lib` is created from your OpenCV code into the project root folder. You can use any of these blocks for model simulation. In this example, the subsystem `subsystem_slwrap_drawDetect` is used.

Step 2: Use Generated Subsystem in Simulink Model

To use the generated subsystem `subsystem_slwrap_drawDetect` with the Simulink model:

1. In your MATLAB current folder, right-click the `VehiclePedestrianDetector.slx` model and click **Open** from the context menu. Drag the generated subsystem from the library to the model. Insert the subsystem between From Multimedia File and Video Viewer block.

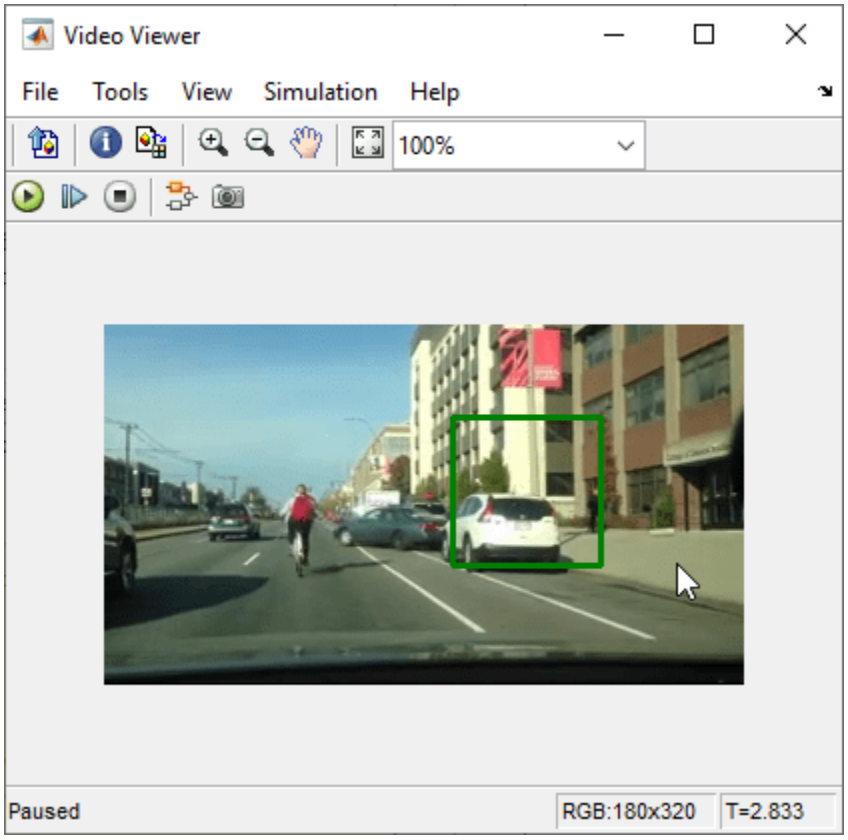
2. Double-click the subsystem and configure these parameter values:

- **Rows**: `180`
- **Columns**: `320`
- **Channels**: `3`
- **Underlying Type**: `uint8`

3. Click **Apply**, and then click **OK**.

Step 3: Simulate the Vehicle and Pedestrian Detector

On the Simulink Toolstrip, in the **Simulation** tab, click on **Run** to simulate the model. After the simulation is complete, the Video Viewer block displays the video of vehicle and pedestrians detected in the video input.



See Also

ToOpenCV | FromOpenCV | `Simulink.ImageType`

More About

- Video Cartoonizer by Using OpenCV Code in Simulink on page 13-64
- Shadow Detection by Using OpenCV Code in Simulink on page 13-55

Video Cartoonizer by Using OpenCV Code in Simulink

This example shows how to use video cartoonizer to enhance colors of the recorded video by using the **OpenCV Importer** app. The cartoonizer varies the colors in the video using parameters such as `maskRadius`, `threshold`, and `ramp`.

First import an OpenCV function into Simulink® by following the “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 13-24. The app 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 recorded video. The cartoonizer works per the `maskRadius`, `threshold`, and `ramp` value. You can change the `maskRadius`, `threshold`, and `ramp` values using the slider component available in the model.

You learn how to:

- Import an OpenCV function into a Simulink library.
- Use blocks from a generated library in a Simulink model.

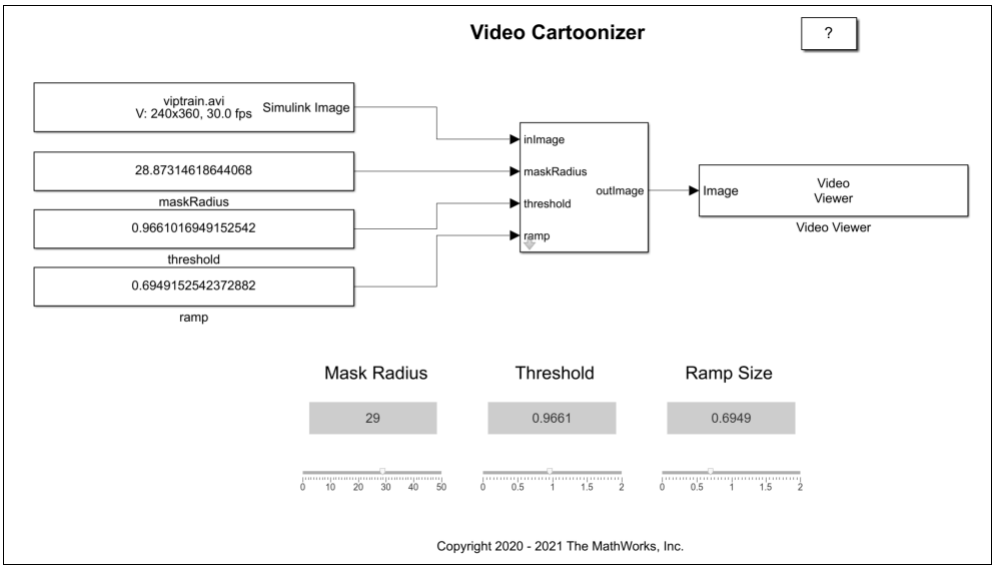
Set Up Your C++ Compiler

To build the OpenCV libraries, identify a compatible C++ compiler for your operating system, as described in “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4. 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 video cartoonizer is implemented by using the Simulink model `Cartoonizer.slx`.

In this model, the `subsystem_slwrap_cartoonize` subsystem resides in the `Cartoonizer_Lib library`. You create the `subsystem_slwrap_cartoonize` subsystem by using the **OpenCV Importer** app. The subsystem accepts a video from the From Multimedia File block, `maskRadius`, `threshold`, and a `ramp` value to cartoonize the video. The output is displayed using the Video Viewer block. In the `subsystem_slwrap_cartoonize` subsystem, `inImage` is the input image, `maskRadius` is the size of the image filter for intensity comparison, `threshold` is the threshold intensity difference between pixels which results in darkening the video, `ramp` is intensity gradient in the output image and `outImage` is the output image. The sliders of `maskRadius`, `threshold`, and `ramp` is used to change the value during the simulation.



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\Cartoonizer`. All your output files are saved to this folder.

Step 1: Import OpenCV Function to Create a Simulink Library

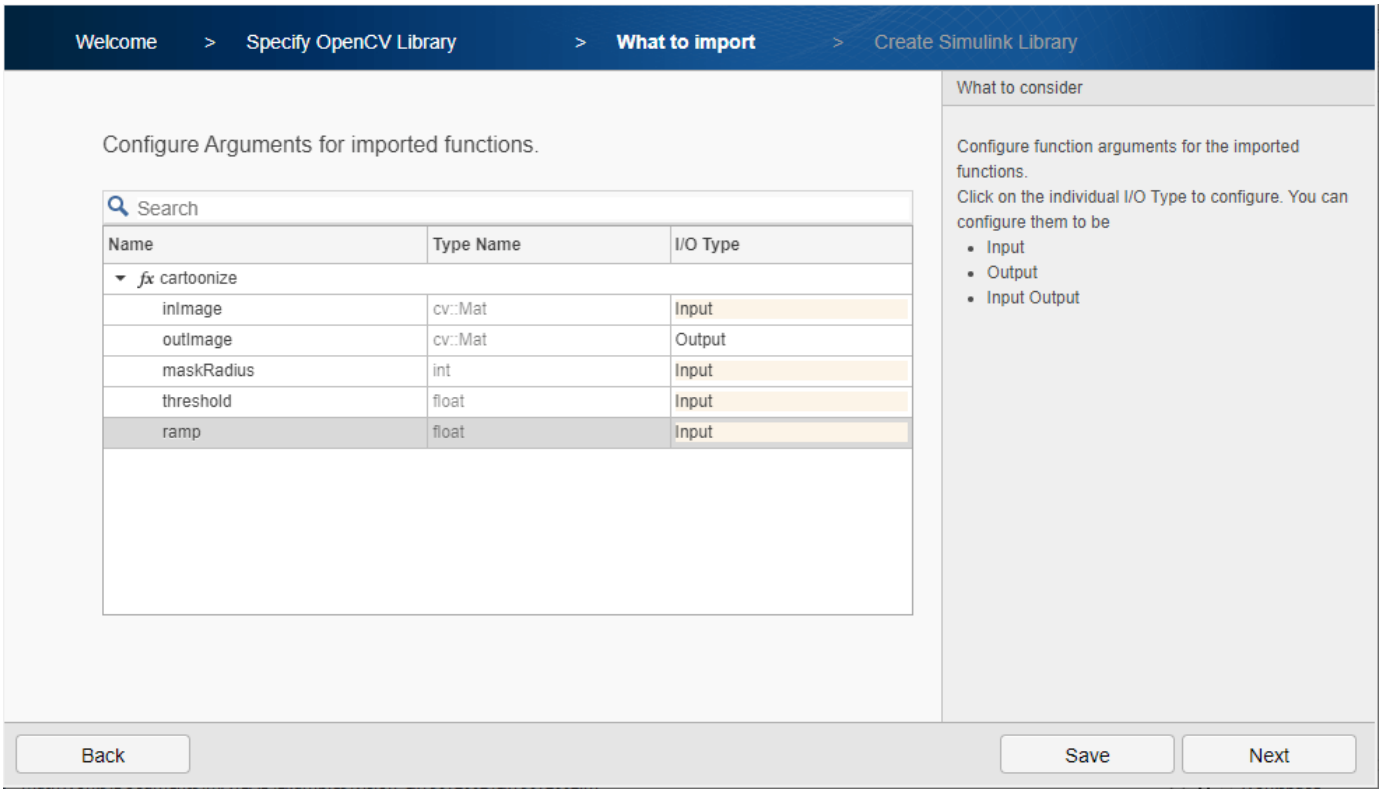
1. To start the **OpenCV Importer** app, click **Apps** on the MATLAB Toolstrip. In the Welcome page, specify the **Project name** as `Cartoonizer`. Make sure that the project name does not contain any spaces. Click **Next**.

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 `cartoonizer.cpp`.
- **Include files**: Specify the path of the `.hpp` header file located inside your project folder as `cartoonizer.hpp`.

3. Analyze your library to find the functions and types for import. Once the analysis is complete, click **Next**. Select the `cartoonize` function and click **Next**.

4. From What to import, select the **I/O Type** for `inImage`, `maskRadius`, `threshold`, and `ramp` as `Input`, `outImg` as Output and then click **Next**.

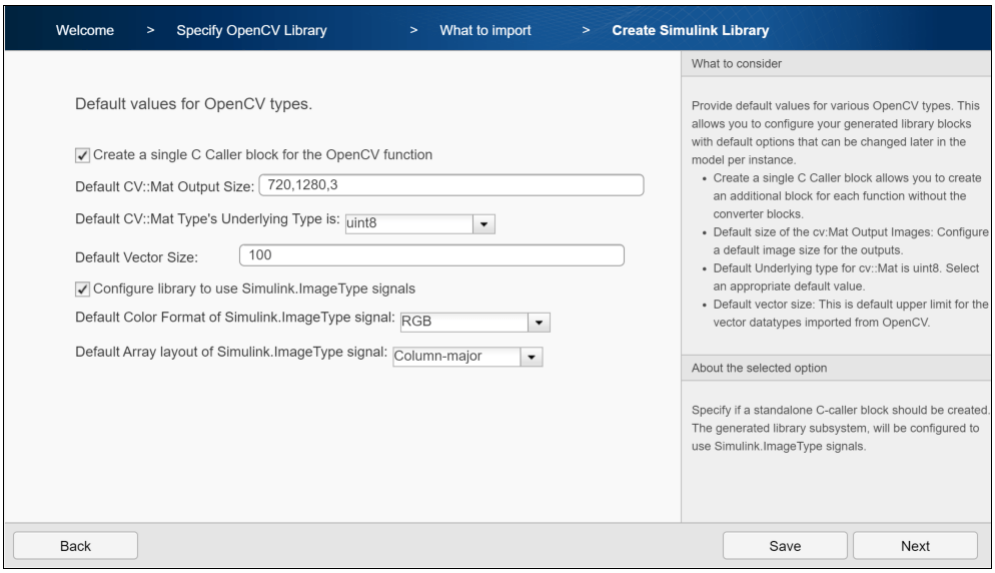


5. In Create Simulink Library, configure the default values of OpenCV types. By default, **Create a single C-caller block for the OpenCV function** is selected to create a C Caller block along with the subsystem in the generated Simulink library.

6. Select **Configure library to use Simulink.ImageType signals** to configure the generated library subsystem to use `Simulink.ImageType` signals.

7. Set **Default Color Format of Simlink.ImageType signal** to `RGB`, which is the default color format of the image.

8. Set **Default Array layout of Simulink.ImageType signal** to `Column-major`, which is the default array layout of the image.



9. To create a Simulink library, click **Next**.

A Simulink library `cartoonize_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_cartoonize` is used.

Step 2: Use Generated Subsystem in Simulink Model

To use the generated subsystem `subsystem_slwrap_cartoonize` with the Simulink model `Cartoonizer.slx`:

1. In your MATLAB current folder, right-click the model `Cartoonizer.slx` and click **Open** from the context menu. Drag the generated subsystem from the library to the model. Insert the subsystem between the input blocks and the Video Viewer block.

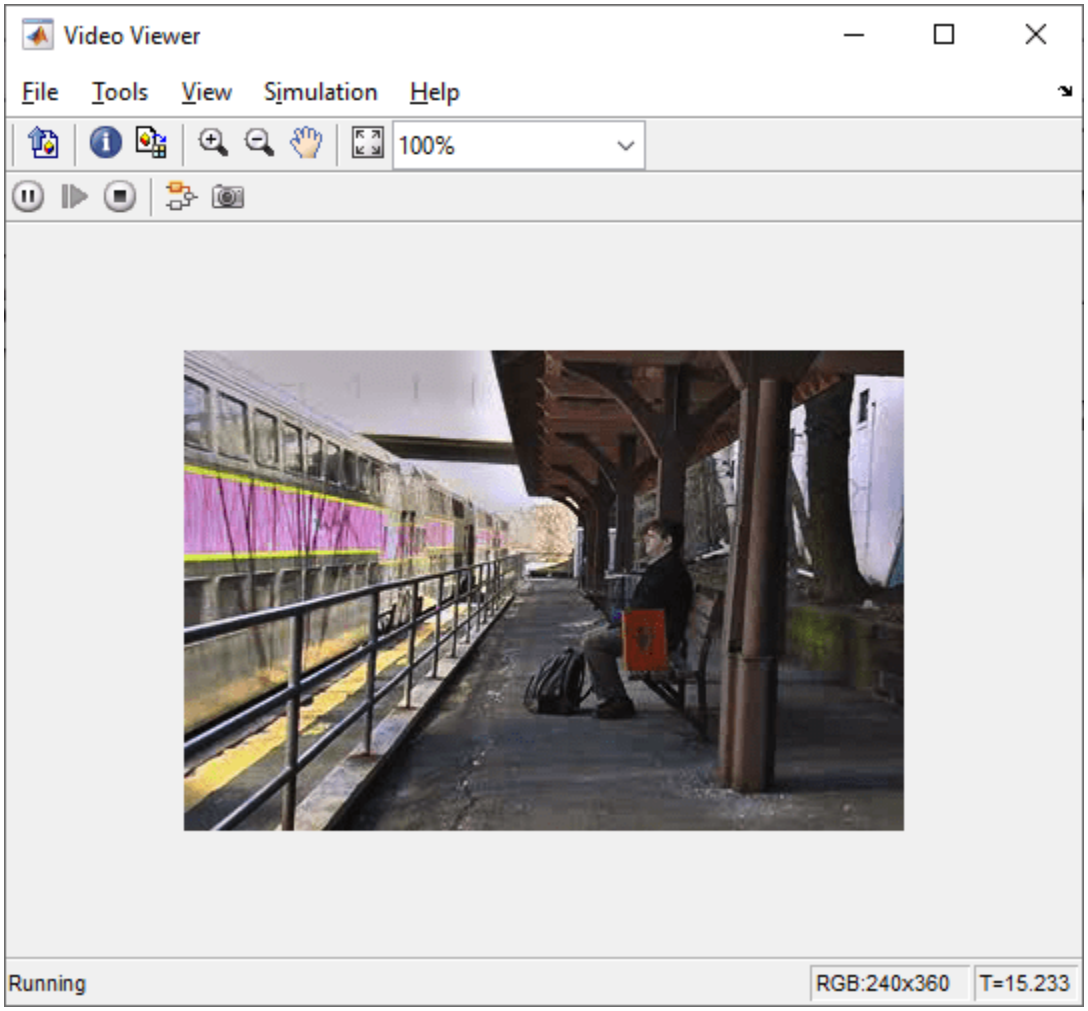
2. Double-click the subsystem and configure these parameter values:

- **Rows**: `240`
- **Columns**: `360`
- **Channels**: `3`
- **Underlying Type**: `uint8`

3. Click **Apply**, and then click **OK**.

Step 3: Simulate the Cartoonizer

On the Simulink Toolstrip, in the **Simulation** tab, click on **Run** to simulate the model. After the simulation is complete, the Video Viewer block displays the video with color enhancement based on the value of `maskRadius`, `threshold`, and `ramp`.



See Also

ToOpenCV | FromOpenCV | `Simulink.ImageType`

More About

- Vehicle and Pedestrian Detector by Using OpenCV Code in Simulink on page 13-60
- Shadow Detection by Using OpenCV Code in Simulink on page 13-55

Convert Between Simulink Image Type and Matrices

You can import image data into a Simulink model as matrix data or as an image with the `Simulink.ImageType` data type. Image To Matrix and Matrix To Image blocks convert between Simulink image data and matrix data. Use these blocks to integrate Simulink image data into an image processing algorithm that contains blocks that do not support Simulink image data.

Copy Example Model to a Writable Location

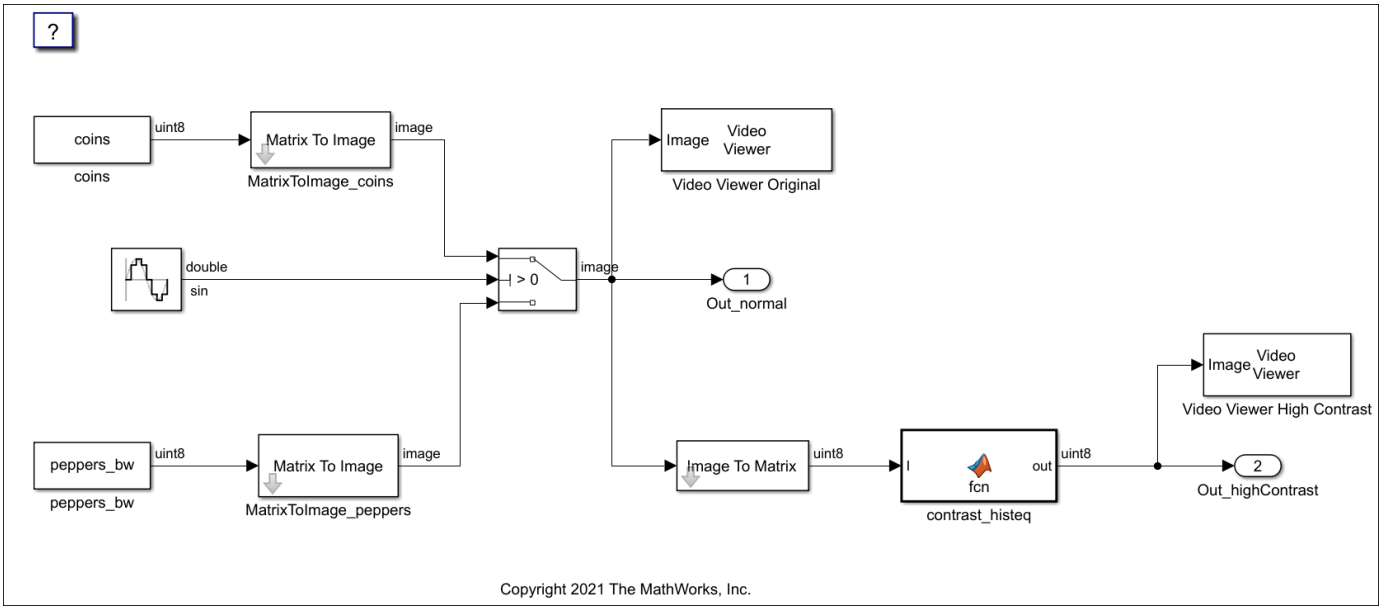
To access the path to the example folder, at the MATLAB command line, enter:

```
OpenCVSimulinkExamples;
```

Copy the example model `ex_imagetypes` to a writable folder location.

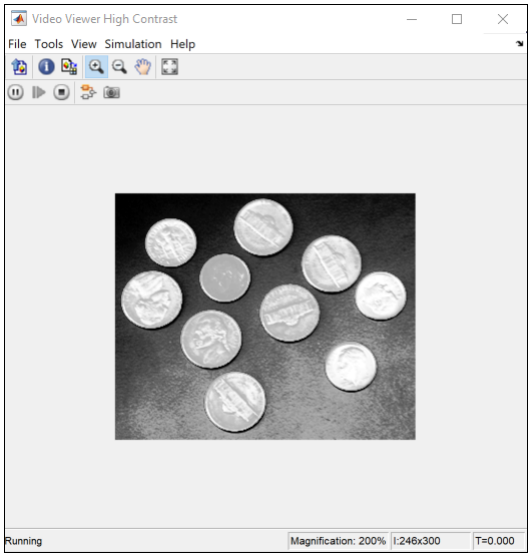
Example Model

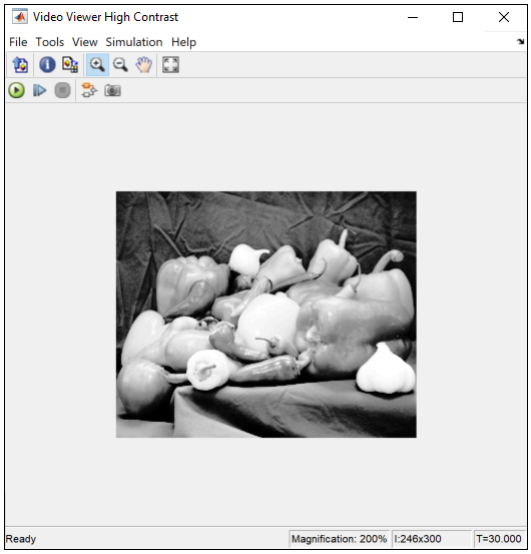
The model `ex_imagetypes` contains two input images, `coins` and `peppers_bw`, specified as matrices. The MATLAB Function block `contrast_histeq` converts the original images into high-contrast images. The MATLAB Function block operates on matrix data only. The Image To Matrix block converts the image data from the Switch block into a matrix to enable the MATLAB Function block to process it.



Simulate Model

On the Simulink toolstrip, in the **Simulation** tab, select **Run** to simulate the model. The Video Viewer block `Video Viewer Original` displays the original images imported into the model, and `Video Viewer High Contrast` displays the high-contrasted images after image processing.





Generate C++ Code

To generate C++ code:

1. On the **Apps** tab on the Simulink toolstrip, select **Simulink Coder**. On the **C++ Code** tab, in the **Settings** list, select **C/C++ Code generation settings** to open the Configuration Parameters dialog box and verify these settings:
   - In the **Simulation Target** pane, **Language** is set to `C++`.
   - In the **Code Generation** pane, under **Target selection**, **Language** is set to `C++`.
   - In the same section, **Language standard** is set to `C++11 (ISO)`.
2. Click the **Build** button and generate code.
3. To view the generated code, on the Simulink toolstrip, click the **Open Report** button.

In the `ex_imagetype.h` file, the code generator declares root-level Outports `Out_normmal` as a member of the C++ class `images::datatypes::Image` implemented by The MathWorks® and `Out_highContrast` as a matrix image:

```
/* External outputs (root outports fed by signals with default storage) */
struct ExtY_ex_imagetypes_T {
  images::datatypes::Image Out_normal; /* '<Root>/Out_normal' */
  uint8_T Out_highContrast[73800];     /* '<Root>/Out_highContrast' */
};
```

The code generator initializes signals of the `Simulink.ImageType` data type in the `ex_imagetypes.c` file:

```
/* Model initialize function */
void ex_imagetypes::initialize()
{
  /* Registration code */
  constructImage(&ex_imagetypes_B.toImage, 1U, 246U, 300U, images::datatypes::
                 ColorFormat::Grayscale, images::datatypes::Layout::
                 ColumnMajorPlanar, images::datatypes::ClassUnderlying::Uint8);
  constructImage(&ex_imagetypes_Y.Out_normal, 1U, 246U, 300U, images::datatypes::
                 ColorFormat::Grayscale, images::datatypes::Layout::
                 ColumnMajorPlanar, images::datatypes::ClassUnderlying::Uint8);
  ...
}
```

This is the code for a Matrix To Image block:

```
/* ToImage: '<S2>/toImage' incorporates:
   *  Constant: '<Root>/coins'
   *  Outport: '<Root>/Out_normal'
   */
  imgData = imageGetDataFcn(&ex_imagetypes_Y.Out_normal);
  inPtr = &ex_imagetypes_ConstP.coins_Value[0];
  std::memcpy(imgData, inPtr, sizeof(uint8_T) * 73800U);
```

This is the code for the Image To Matrix block:

```
/* FromImage: '<S1>/fromImage' incorporates:
   *  Outport: '<Root>/Out_normal'
   */
  tmp_3 = ex_imagetypes_Y.Out_normal;
  imgData = imageGetDataFcn(&tmp_3);
  std::memcpy(&ex_imagetypes_B.fromImage[0], imgData, sizeof(uint8_T) * 73800U);
```

When a model contains signals of the `Simulink.ImageType` data type, the code generator produces additional shared utility files. These files declare and define utilities to construct, destruct, and return information about meta attributes of the images:

- `image_type.h`
- `image_type.cpp`

See Also

Matrix To Image | Image To Matrix | `Simulink.ImageType`

Related Examples

- “Install and Use Computer Vision Toolbox Interface for OpenCV in Simulink” on page 13-24
- “Smile Detection by Using OpenCV Code in Simulink” on page 13-45

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 14-2
- “Import from Video Files” on page 14-4
- “Batch Process Image Files” on page 14-6
- “Convert R'G'B' to Intensity Images” on page 14-7
- “Process Multidimensional Color Video Signals” on page 14-10
- “Video Formats” on page 14-12
- “Image Formats” on page 14-13

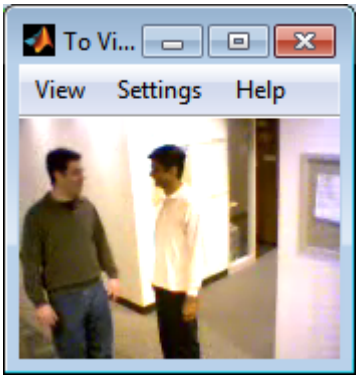
Export to Video Files

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

Configuration Parameters

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)`

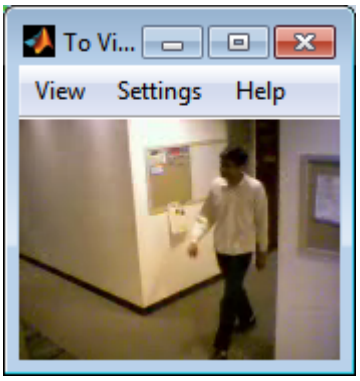
Import from Video Files

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 MATLABcommand line.

```
ex_import_mmf
```

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



You have now imported and displayed a multimedia file in the Simulink model. In the “Export to Video Files” on page 14-2 example you can manipulate your video stream and export it to a multimedia file.

For more information on the blocks used in this example, see the From Multimedia File and To Video Display block reference pages.

Setting Block Parameters for this Example

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

Block	Parameter
From Multimedia File	Use the From Multimedia File block to import the multimedia file into the model: • If you do not have your own multimedia file, use the default `vipmen.avi` file, for the **File name** parameter. • If the multimedia file is on your MATLAB path, enter the filename for the **File name** parameter. • If the file is not on your MATLAB path, use the **Browse** button to locate the multimedia file. • Set the **Image signal** parameter to `Separate color signals`. By default, the **Number of times to play file** parameter is set to `inf`. The model continues to play the file until the simulation stops.

Block	Parameter
To Video Display	Use the To Video Display block to view the multimedia file. • **Image signal**: `Separate color signals` Set this parameter from the **Settings** menu of the display viewer.

Configuration Parameters

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_1m4_*.tif'));
fileNames = {dirOutput.name}'
```

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

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

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

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

For additional information about batch processing, see the “Image Sequences and Batch Processing” 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
```

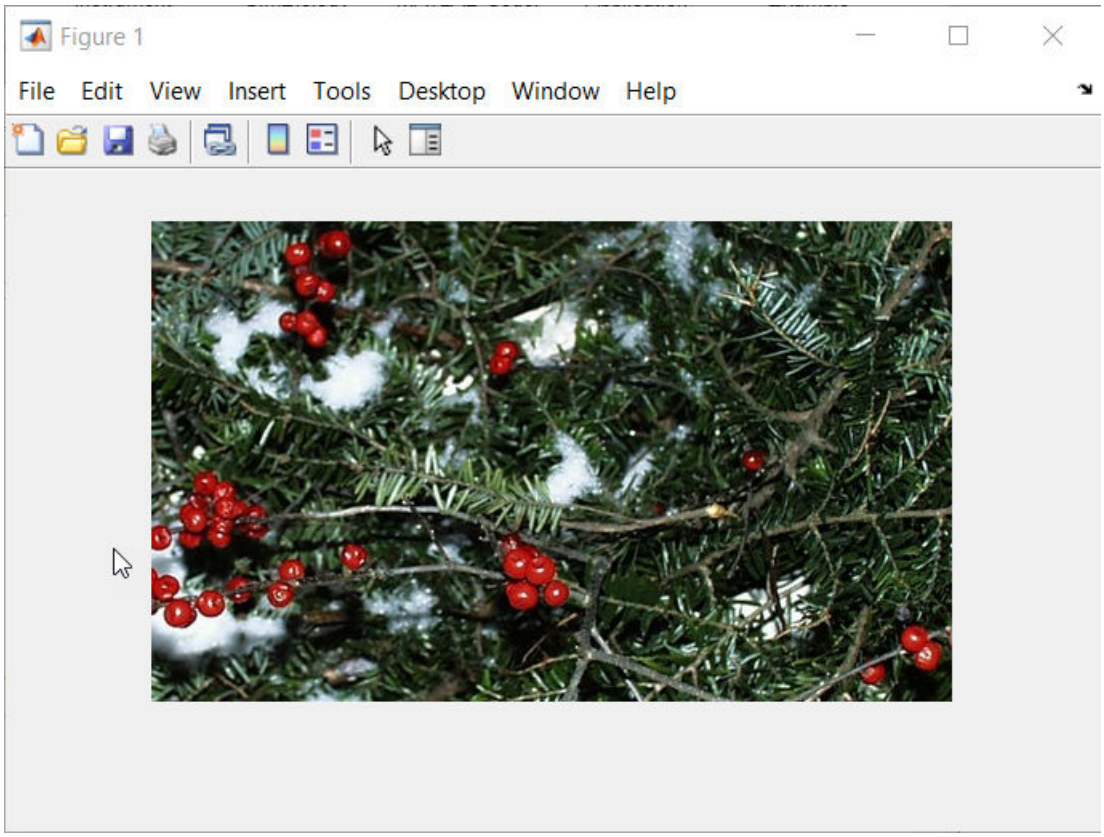
1. Define an R'G'B' image in the MATLAB workspace. To read in an R'G'B' image from a JPG file, at the MATLAB command prompt, type

   ```
   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

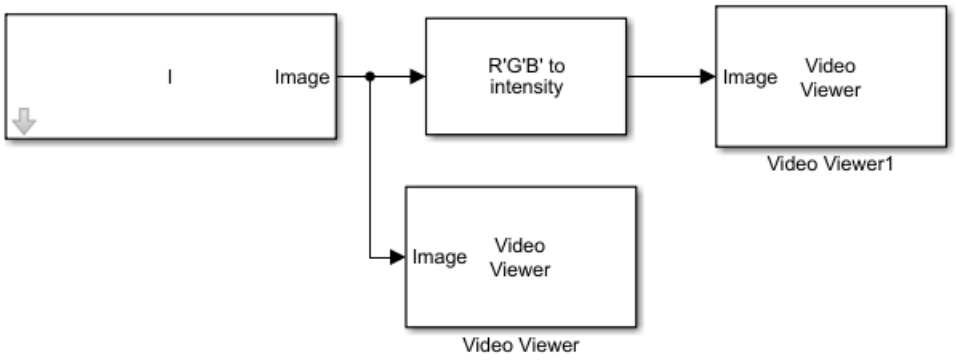
   ```
   imshow(I)
   ```



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

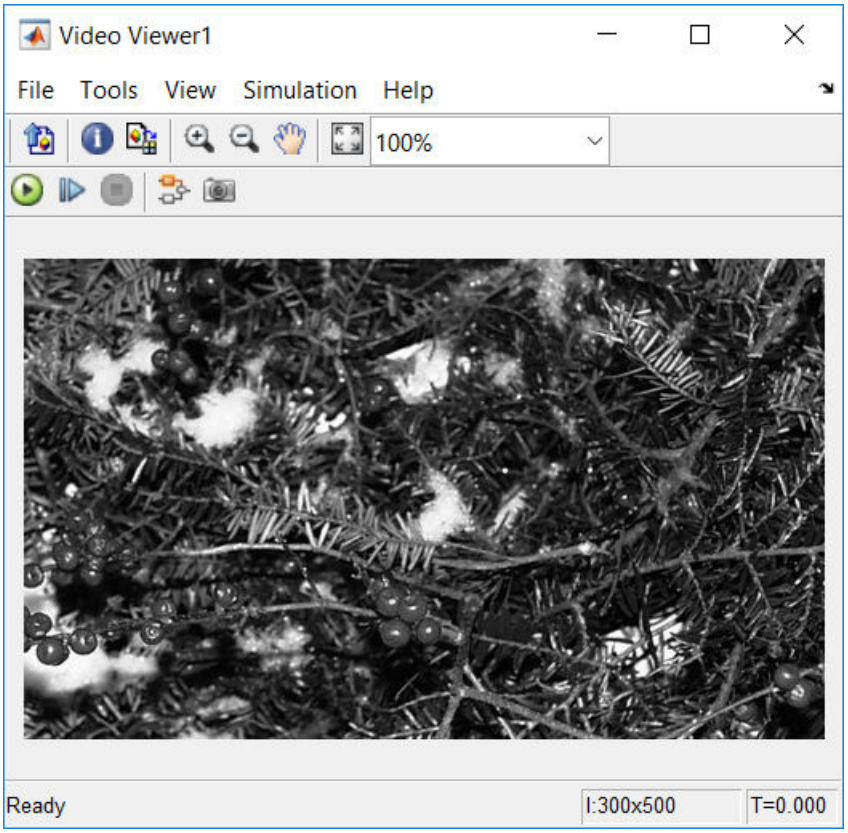
Block	Library	Number of Blocks
Image From Workspace	Computer Vision Toolbox > Sources	1
Color Space Conversion	Computer Vision Toolbox > Conversions	1
Video Viewer	Computer Vision Toolbox > Sinks	2

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.

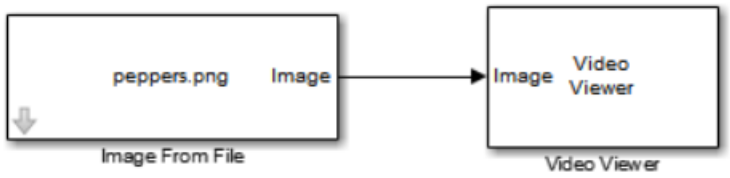
   The image displayed in the Video Viewer window is the intensity version of the `greens.jpg` image.


Video Viewer1
File Tools View Simulation Help
100%
Ready
I:300x500
T=0.000

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
```



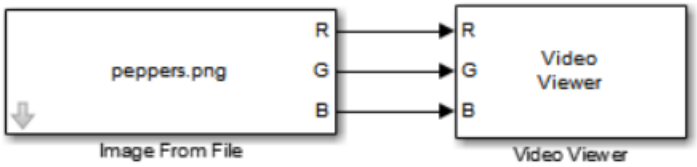
You can choose to process the image as a multidimensional array by setting the **Image signal** parameter to `One multidimensional signal` in the Image From File block dialog box.

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. In the Simulink Toolstrip, on the **Debug** tab, select **Information Overlays > Base 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

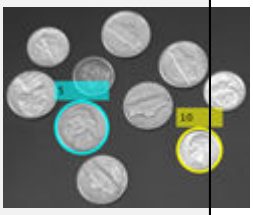
- “Choose Function to Visualize Detected Objects” on page 15-2
- “Display, Stream, and Preview Videos” on page 15-5
- “Draw Shapes and Lines” on page 15-7

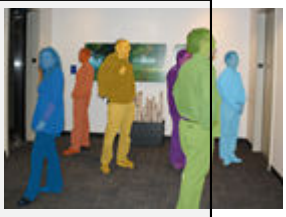
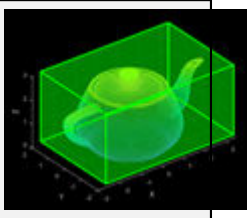
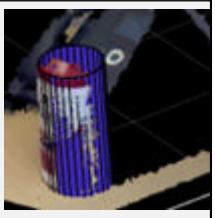
Choose Function to Visualize Detected Objects

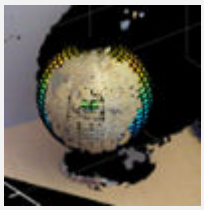
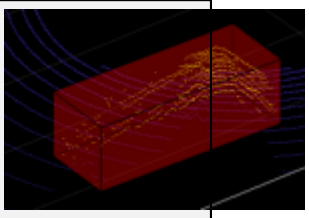
Computer Vision Toolbox offers several functions to visualize detected objects by inserting or overlaying shapes on image, video, and point cloud data.

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
`insertShape`	Yes	Yes	No	Yes	 Insert Shapes on Image
`insertText`	Yes	Yes	No	Yes	 Insert Text on Image
`insertObjectAnnotation`	Yes	Yes	No	Yes	 Annotate an Image

Function	Images	Video	Point Clouds	Code Generation	Example
`insertMarker`	Yes	Yes	No	Yes	 Insert Markers on Image
`insertObjectMask`	Yes	Yes	No	Yes	 Insert Multicolor Masks on Image
`showShape`	Yes	Yes	Yes	No	 Show Cuboid on Detected Object in Point Cloud
`plot` object function of the `cylinderModel`	No	No	Yes	No	 Fit Cylinder Shape on Point Cloud

Function	Images	Video	Point Clouds	Code Generation	Example
`plot` object function of the `sphereModel`	No	No	Yes	No	 "Detect Sphere in Point Cloud"
`plot` object function of the `planeModel`	No	No	Yes	No	 Fit Plane Shape on Point Cloud
`plot` object function of the `cuboidModel` object.	No	No	Yes	No	 "Fit Cuboid Over Point Cloud Data" (Lidar Toolbox)

See Also

Objects
`cuboidModel` | `planeModel` | `sphereModel` | `cylinderModel` | `pcplayer` | `vision.VideoPlayer` | `vision.DeployableVideoPlayer`

Functions
`pcshow` | `imshow` | `showShape` | `insertObjectMask` | `insertMarker` | `insertObjectAnnotation` | `insertText` | `insertShape`

Display, Stream, and Preview Videos

In this section...
“View Streaming Video in MATLAB” on page 15-5
“Preview Video in MATLAB” on page 15-5
“View Video in Simulink” on page 15-5

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 **Video Viewer** app to view videos directly from file or from variables in the MATLAB workspace. The app is a full featured video player with toolbar controls. The app also offers access to tools that enable you to modify the appearance of the video and to inspect the data in a region of the image in finer detail.

You can open several instances of the **Video Viewer** app simultaneously to view multiple video data sources at once. You can also dock the apps 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 **Video Viewer** app enables you to view video signals in Simulink models without adding blocks to your model. You can open several instances of the app simultaneously to view multiple video data sources at once. You can also dock these apps 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 apps.

Set Simulink simulation mode to `Normal` to use **Video Viewer**. The app does not work when you use “Accelerating Simulink Models” on page 22-3.

Example 15.1. Use `implay` to view a Simulink signal:

1. Open a Simulink model.
2. Open the **Video Viewer** app 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 **Video Viewer** toolbar, select **File > Connect to Simulink Signal** .

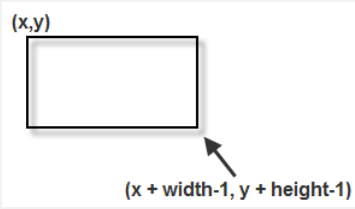
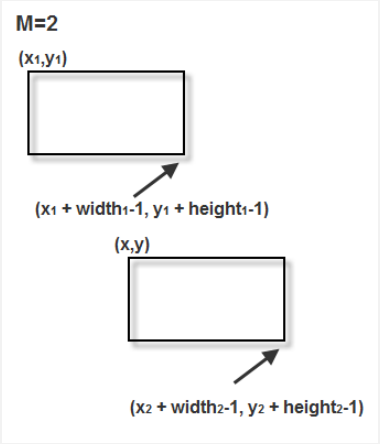
   The video appears in the player window.
6. You can use multiple **Video Viewer** apps to display different Simulink signals.

Note During code generation, the Simulink Coder™ does not generate code for the **Video Viewer** app.

Draw Shapes and Lines

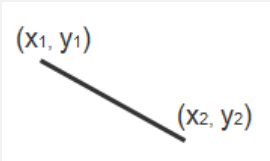
When you specify the type of shape to draw, you must also specify it's location on the image. The table shows the format for the points input for the different shapes.

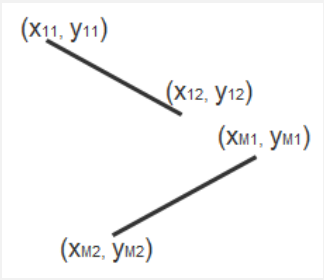
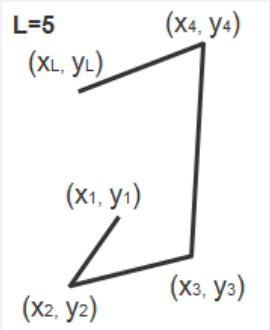
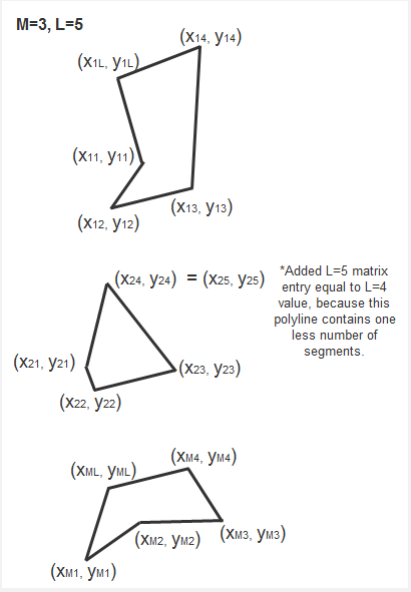
Rectangle

Shape	PTS input	Drawn Shape
Single Rectangle	Four-element row vector `[x y width height]` where • `x` and `y` are the one-based coordinates of the upper-left corner of the rectangle. • `width` and `height` are the width, in pixels, and height, in pixels, of the rectangle. The values of `width` and `height` must be greater than 0.	
M Rectangles	*M*-by-4 matrix $\begin{bmatrix} x_1 & y_1 & width_1 & height_1 \\ x_2 & y_2 & width_2 & height_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_M & y_M & width_M & height_M \end{bmatrix}$ where each row of the matrix corresponds to a different rectangle and is of the same form as the vector for a single rectangle.	

Line and Polyline

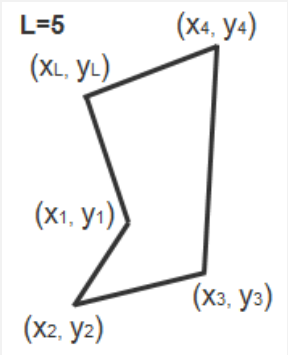
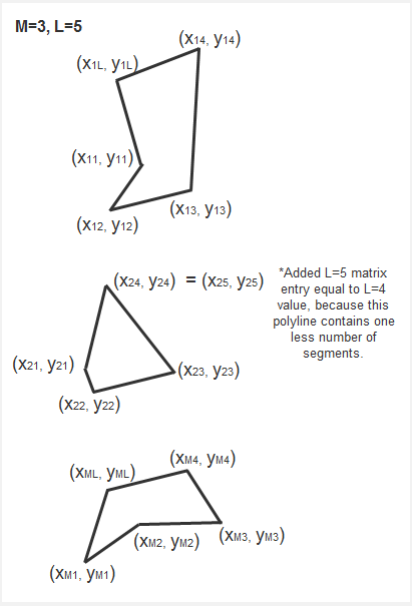
You can draw one or more lines, and one or more polylines. A polyline contains a series of connected line segments.

Shape	PTS input	Drawn Shape
Single Line	Four-element row vector `[x1 y1 x2 y2]` where • x_1 and y_1 are the coordinates of the beginning of the line. • x_2 and y_2 are the coordinates of the end of the line.	

Shape	PTS input	Drawn Shape
M Lines	*M*-by-4 matrix $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} \\ x_{21} & y_{21} & x_{22} & y_{22} \\ \vdots & \vdots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} \end{bmatrix}$ where each row of the matrix corresponds to a different line and is of the same form as the vector for a single line.	
Single Polyline with (*L*-1) Segments	Vector of size 2*L*, where *L* is the number of vertices, with format, `[x1, y1, x2, y2, ..., xL, yL]`. • x_1 and y_1 are the coordinates of the beginning of the first line segment. • x_2 and y_2 are the coordinates of the end of the first line segment and the beginning of the second line segment. • x_L and y_L are the coordinates of the end of the $(L\text{-}1)^{th}$ line segment. The polyline always contains (*L*-1) number of segments because the first and last vertex points do not connect. The block produces an error message when the number of rows is less than two or not a multiple of two.	
M Polylines with (*L*-1) Segments	2*L*-by-*N* matrix $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ where each row of the matrix corresponds to a different polyline and is of the same form as the vector for a single polyline. When you require one polyline to contain less than (*L*–1) number of segments, fill the matrix by repeating the coordinates of the last vertex. The block produces an error message if the number of rows is less than two or not a multiple of two.	

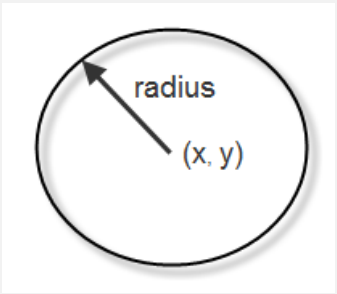
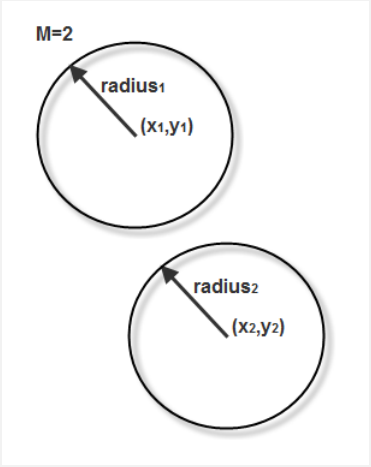
Polygon

You can draw one or more polygons.

Shape	PTS input	Drawn Shape
Single Polygon with L line segments	Row vector of size $2L$, where L is the number of vertices, with format, `[x1 y1 x2 y2 ... xL yL]` where • x_1 and y_1 are the coordinates of the beginning of the first line segment. • x_2 and y_2 are the coordinates of the end of the first line segment and the beginning of the second line segment. • x_L and y_L are the coordinates of the end of the $(L\text{-}1)^{th}$ line segment and the beginning of the L^{th} line segment. The block connects `[x1 y1]` to `[xL yL]` to complete the polygon. The block produces an error if the number of rows is negative or not a multiple of two.	
M Polygons with the largest number of line segments in any line being L	M-by-$2L$ matrix $\begin{bmatrix} x_{11} & y_{11} & x_{12} & y_{12} & \cdots & x_{1L} & y_{1L} \\ x_{21} & y_{21} & x_{22} & y_{22} & \cdots & x_{2L} & y_{2L} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{M1} & y_{M1} & x_{M2} & y_{M2} & \cdots & x_{ML} & y_{ML} \end{bmatrix}$ where each row of the matrix corresponds to a different polygon and is of the same form as the vector for a single polygon. If some polygons are shorter than others, repeat the ending coordinates to fill the polygon matrix. The block produces an error message if the number of rows is less than two or is not a multiple of two.	

Circle

You can draw one or more circles.

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

See Also

`insertObjectAnnotation` | `insertShape` | `insertMarker` | `insertText`

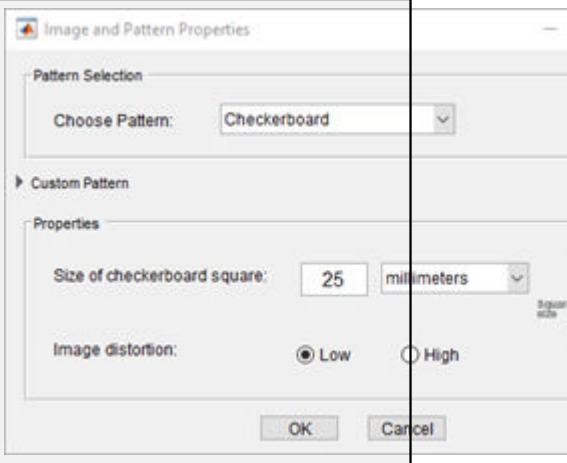
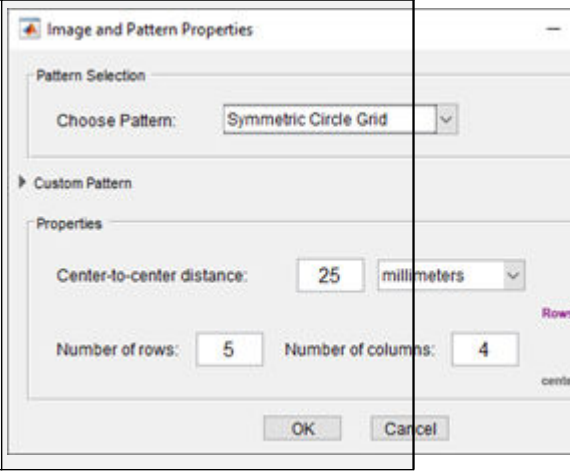
Registration and Stereo Vision

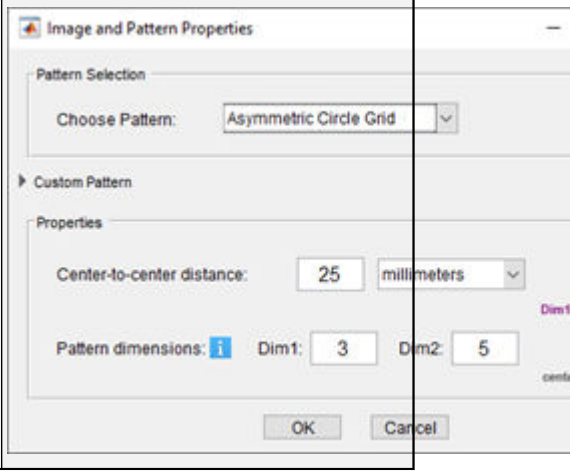
- “Select Calibration Pattern and Set Properties” on page 16-2
- “Prepare Camera and Capture Images” on page 16-4
- “Calibration Patterns” on page 16-6
- “Fisheye Calibration Basics” on page 16-11
- “Using the Single Camera Calibrator App” on page 16-18
- “Using the Stereo Camera Calibrator App” on page 16-31
- “What Is Camera Calibration?” on page 16-43
- “Structure from Motion Overview” on page 16-49

Select Calibration Pattern and Set Properties

After selecting the images you want to use, the **Camera Calibrator** app displays an Image and Pattern Properties dialog box, or the **Stereo Camera Calibrator** app displays a Load Stereo Images dialog box. In the **Pattern Selection** section, select the type of pattern to use for the calibration. The drop-down lists the natively supported patterns and any previously created custom patterns. Alternatively, you can create a custom pattern by using a template. For more details, see “Custom Detector Pattern” on page 16-9.

You must provide calibration pattern properties related to the size and dimension of the pattern structure. For checkerboard patterns, specify the square size. For the **Camera Calibrator** app, you must also select low or high image distortion. Select **High** distortion when the images have been taken using a wide-angle lens, such as a fisheye camera.

Pattern	Calibrator App Support	Example
Checkerboard	• **Camera Calibrator** • **Stereo Camera Calibrator**	
Symmetric Circle Grid	• **Camera Calibrator**	

Pattern	Calibrator App Support	Example
Asymmetric Circle Grid	• **Camera Calibrator** • **Stereo Camera Calibrator**	

Prepare Camera and Capture Images

Camera Setup

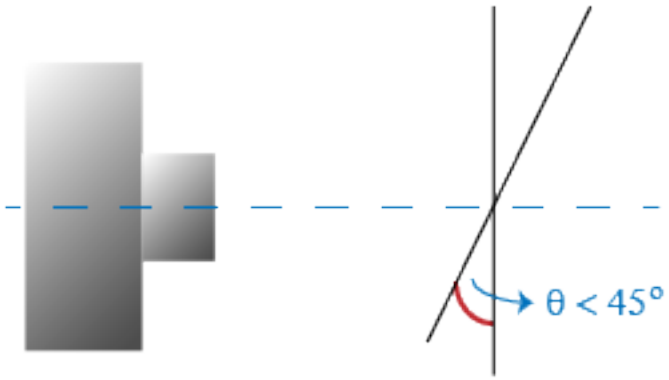
To calibrate your camera, follow these rules and tips:

- Capture the images with a fixed focus by disabling the autofocus of the camera. Set the focus of the camera for your application.
- Place the calibration pattern at such a distance from the camera that at least 20% of the image is covered by pattern.
- If capturing images using a stereo camera pair, place the pattern in different orientations such that it is fully visible from both cameras of the stereo camera pair.
- Keep in mind that changing the zoom setting between images, changes the focal length.

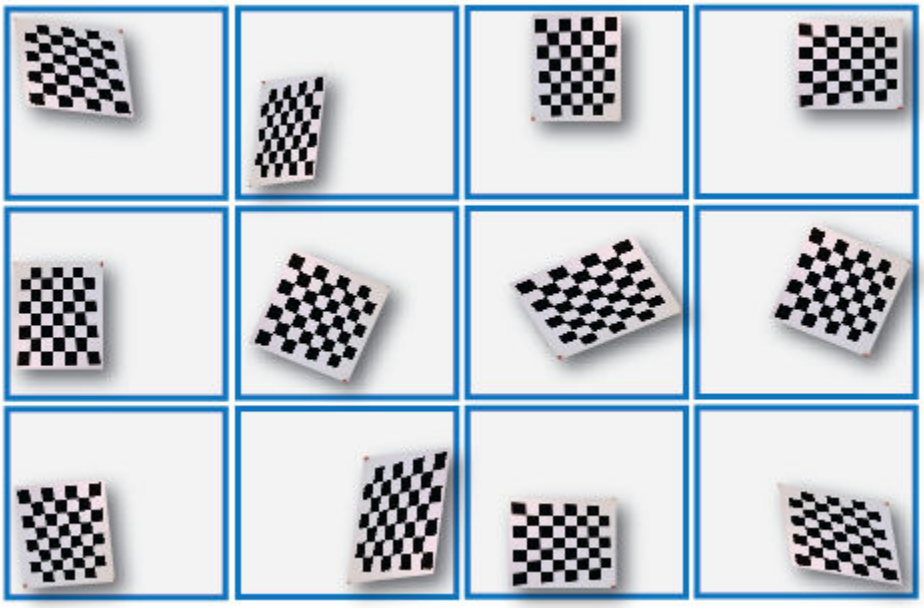
Capture Images

For better results, use a minimum of 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:

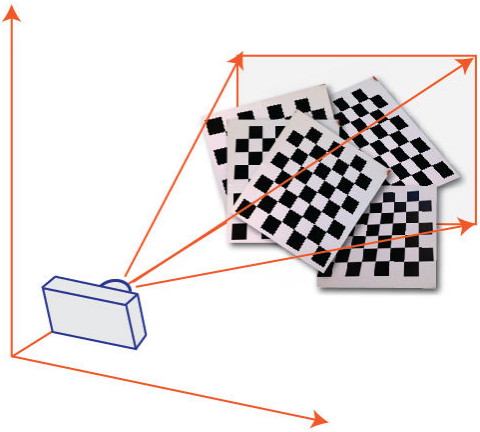
- Capture the images of the pattern at a distance roughly equal to the distance from your camera to the objects of interest. For example, if you plan to measure objects from 2 meters, keep your pattern approximately 2 meters from the camera.
- Place the pattern at an angle less than 45 degrees relative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

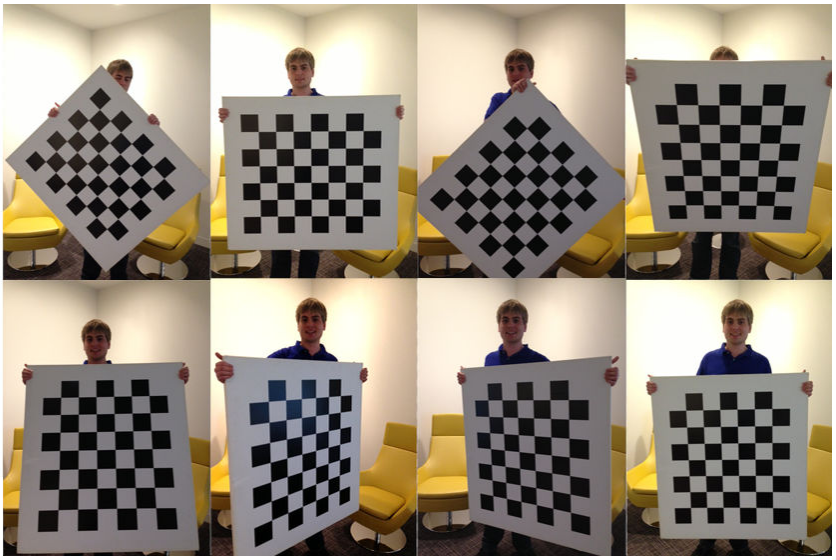


- Do not modify the images For example, do not crop them.
- Do not use autofocus or change the zoom settings between images.
- Capture the images of the calibration pattern at different orientations relative to the camera. For more information on calibration patterns, see “Calibration Patterns” on page 16-6.
- Capture a variety of images of the pattern to account for as much of the image frame as possible. Lens distortion increases radially from the center of the image, and is not always uniform across the image frame. To capture this lens distortion, the pattern must include areas near the edges of the captured images.



The calibrator works with a range of pattern sizes. As a general rule, your pattern should fill at least 20% of the captured image. For example, these images have been taken with a checkerboard square size of 108 mm:





Calibration Patterns

In this section...
“What Are Calibration Patterns?” on page 16-6
“Supported Patterns” on page 16-8
“Checkerboard Pattern” on page 16-8
“Circle Grid Patterns” on page 16-9
“Custom Detector Pattern” on page 16-9

The **Camera Calibrator** and **Stereo Camera Calibrator** app support checkerboard, circle grid, and custom detector patterns to extract keypoints. The apps enable you to select a checkerboard, symmetric circle grid, or asymmetric circle grid detector pattern. The apps also provide the ability for you to add a custom pattern detector, and provide built-in templates to help you create it.

What Are Calibration Patterns?

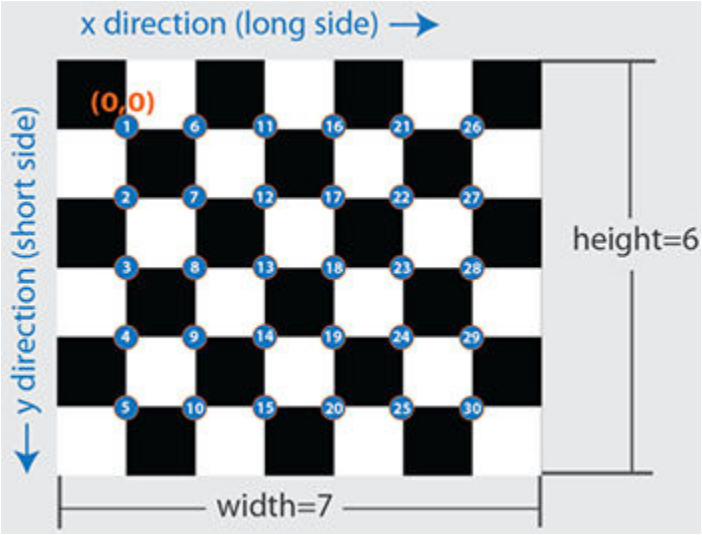
Camera calibration estimates the parameters of a lens, the image sensor of an image, or a video camera. You can use these parameters to estimate structures in a scene and to remove lens distortion. The camera parameters include:

- Intrinsics — These relate to the internal characteristics of a camera, such as the focal length, the optical cente (also known as the principal point), and the skew coefficient.
- Extrinsics — These describe the location (position and orientation) of the camera in the 3-D scene.

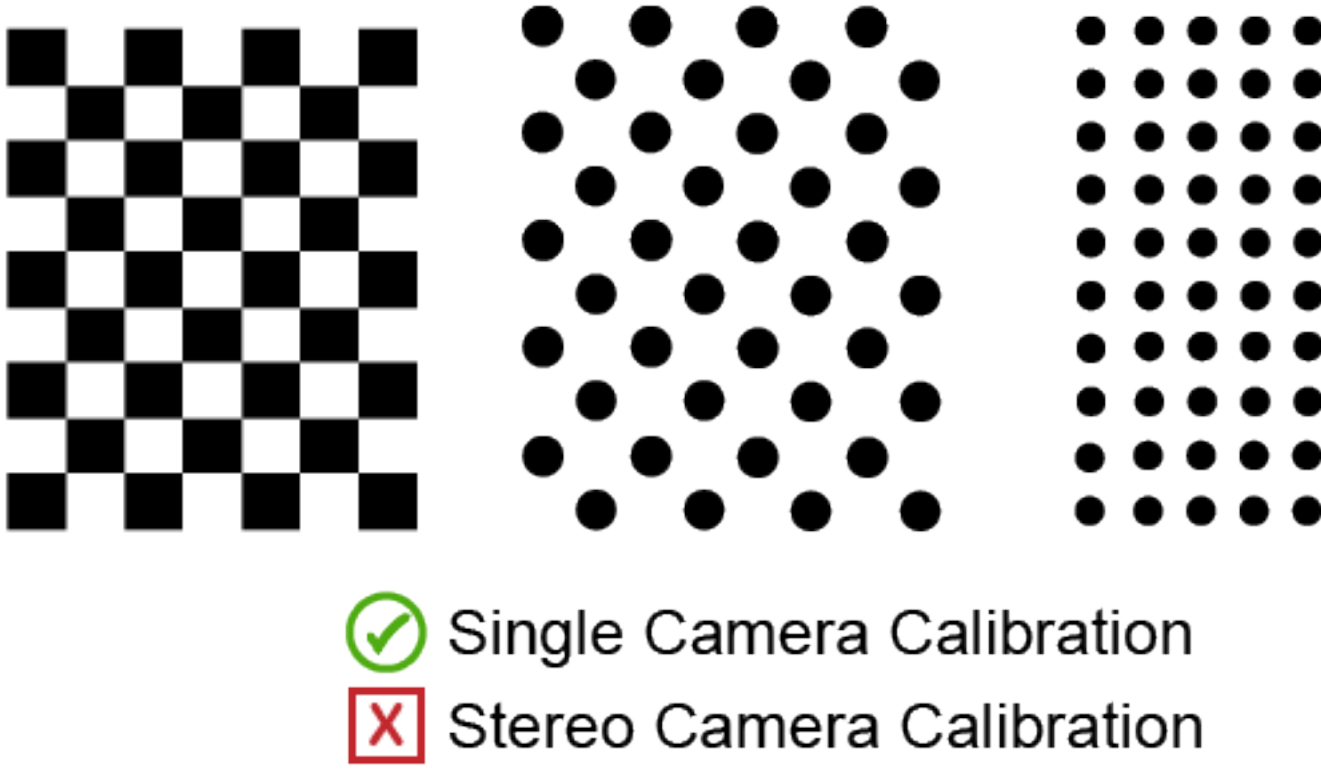
For more detail on how intrinsics and extrinsics are calculated, and explanations of single camera (pinhole) and fisheye camera calibration, see “What Is Camera Calibration?” on page 16-43 and “Fisheye Calibration Basics” on page 16-11, respectively.

To estimate the intrinsics and extrinsics parameters, you need 3-D world points and their corresponding 2-D image points. You can get these correspondences by using multiple images of a calibration pattern. The calibration pattern, sometimes known as a calibration grid or a calibration target, is a repeating pattern of known size and spacing.

For example, a checkerboard pattern consists of alternating white and black squares of equal size. The corners of the squares that lie inside the pattern are used as the control points. These corners can be detected on the 2-D calibration image automatically by using a corner detector algorithm. By assuming the lower-right corner point of the top-left square of the checkerboard is the origin, we can also determine their 3-D world coordinates of the points by using the square size of the checkerboard.



Similar assumptions are used to identify the origin in different types of calibration patterns, so that algorithms can automatically identify them based on their structure. When calibrating stereo cameras, the origin of the pattern must be uniquely identifiable, and thus should not have 180-degree ambiguity. This means, the pattern should not look the same when rotated by 180 degrees. These are examples of calibration patterns that contain 180-degree ambiguity, and should be avoided when calibrating stereo cameras:



Supported Patterns

Computer Vision Toolbox contains an example pattern for each of the natively supported pattern types. To open a PDF file for any of these supported patterns, select the link or type the corresponding command at the MATLAB prompt:

- Checkerboard pattern:

  ```
  open checkerboardPattern.pdf
  ```

- Symmetric Circles Grid Pattern:

  ```
  open symmetricCirclesGridPattern.pdf
  ```

- Asymmetric Circles Grid Pattern:

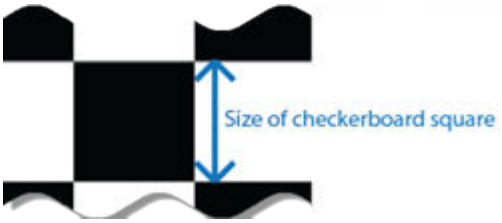
  ```
  open asymmetricCirclesGridPattern.pdf
  ```

Checkerboard Pattern

The checkerboard pattern is the most commonly used calibration pattern for camera calibration. The control points for this pattern are the corners that lie inside the checkerboard. Because corners are extremely small, they are often invariant to perspective and lens distortion. The calibrator apps can also detect partial checkerboards, which can be useful when calibrating cameras with wide-angle lenses. Use a checkerboard that contains an even number of squares along one edge and an odd number of squares along the other edge, with two black corner squares along one side and two white corner squares on the opposite side. This enables the app to determine the orientation of the pattern and the origin. The calibrator assigns the longer side as the *x*-direction. A square checkerboard pattern can produce unexpected results for camera extrinsics.

To prepare the checkerboard pattern:

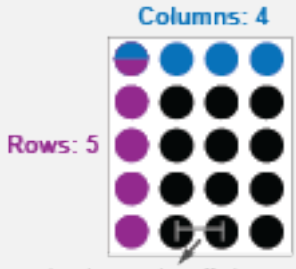
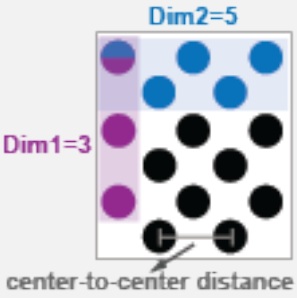
1. Attach the checkerboard printout to a flat surface. Imperfections on the surface can affect the accuracy of the calibration.
2. Measure one side of a checkerboard square. You need this measurement for calibration. The size of the squares can vary depending on printer settings.



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

Circle Grid Patterns

Circle grid patterns, sometimes referred to as a grid of circles, are a class of calibration patterns that use evenly spaced circles to form a grid structure. They are broadly classified into two types: symmetric and asymmetric patterns.

Pattern	Example	Description
Symmetric		• Circles are arranged evenly in rows and columns • Dimensions are measured in number of circles as [*height width*], where *height* is the number of circles in one row and *width* is the number of circles in one column. • Cannot be used to calibrate stereo cameras due to 180-degree ambiguity.
Asymmetric		• Every second row of circles is offset by half the column distance between neighboring row elements. • Dimensions are measured in number of circles as [*dim1 dim2*], where *dim1* is the number of circles along the dimension that contains the same number of circles in each row or column and *dim2* is the number of circles across two adjacent columns (or rows) in the dimension where the two columns (or rows) contain an unequal number of circles. • Greater density of points for the same circle radius. • Can be used to calibrate stereo cameras. No 180-degree ambiguity.

Custom Detector Pattern

You can create a custom pattern by using a template. When loading your images into the calibrator, in the properties dialog box, expand the **Custom Pattern** section and select **Import Pattern Detector** to open the template. For an example of creating and using a custom template, see“Camera Calibration Using AprilTag Markers” on page 1-50.

See Also

Apps
Camera Calibrator | Stereo Camera Calibrator

Related Examples

- Checkerboard pattern
- Symmetric Circles Grid Pattern
- Asymmetric Circles Grid Pattern

More About

- “Using the Stereo Camera Calibrator App” on page 16-31

Fisheye Calibration Basics

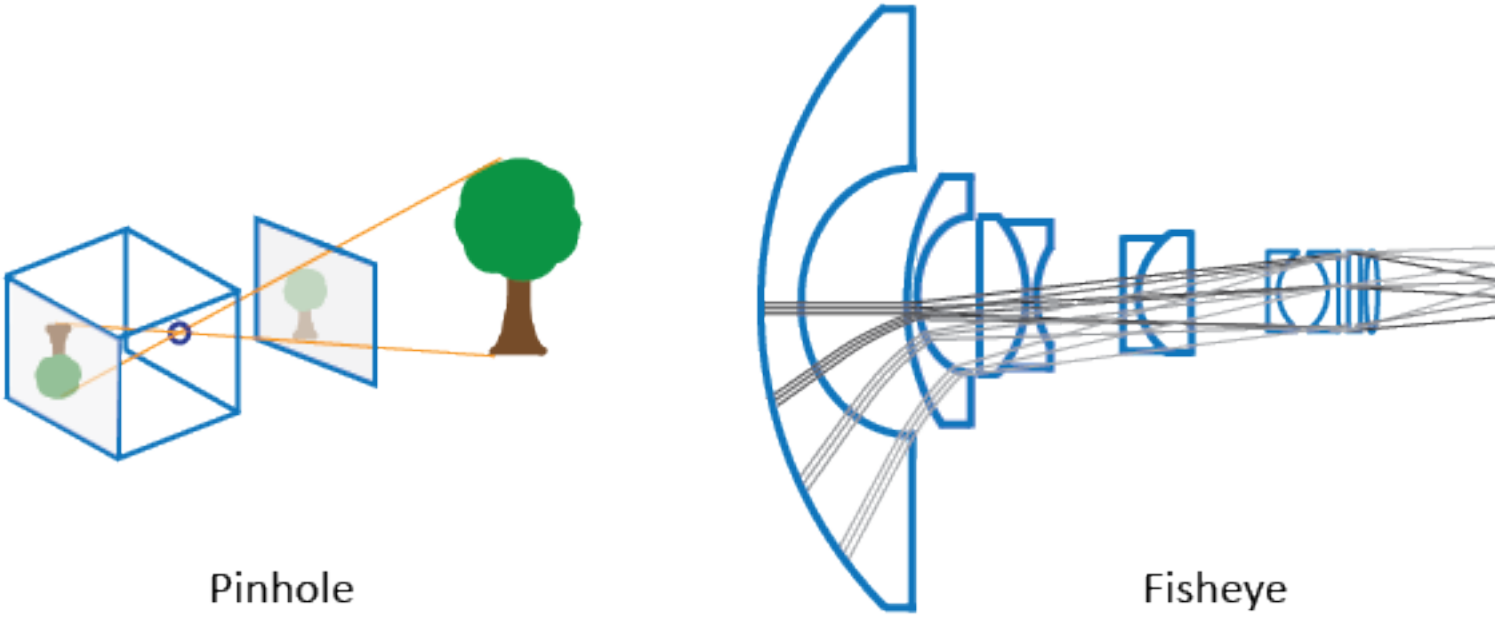
Camera calibration is the process of computing the extrinsic and intrinsic parameters of a camera. Once you calibrate a camera, you can use the image information to recover 3-D information from 2-D images. You can also undistort images taken with a fisheye camera. The Computer Vision Toolbox contains calibration algorithms for the pinhole camera model and the fisheye camera model. You can use the fisheye model with cameras up to a field of view (FOV) of 195 degrees.

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

Fisheye image

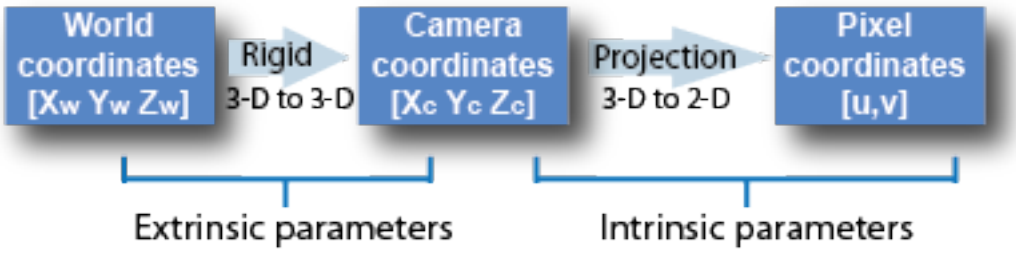
Undistorted fisheye image

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



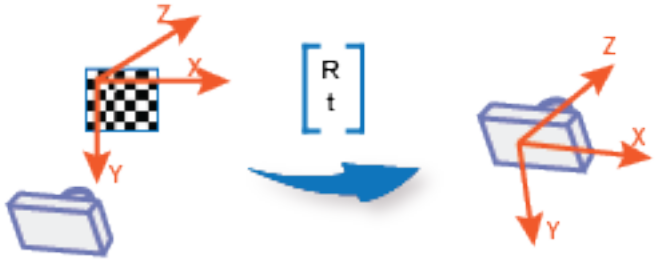
Fisheye Camera Model

The Computer Vision Toolbox calibration algorithm uses the fisheye camera model proposed by Scaramuzza [1] on page 16-16. The model uses an omnidirectional camera model. The process treats the imaging system as a compact system. In order to relate a 3-D world point on to a 2-D image, you must obtain the camera extrinsic and intrinsic parameters. World points are transformed to camera coordinates using the extrinsics parameters. The camera coordinates are mapped into the image plane using the intrinsics parameters.



Extrinsic Parameters

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



The transformation from world points to camera points is:

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = R\begin{pmatrix} X_w \\ Y_w \\ Z_w \end{pmatrix} + T$$

Rotation

Translation

Camera points

World points

Intrinsic Parameters

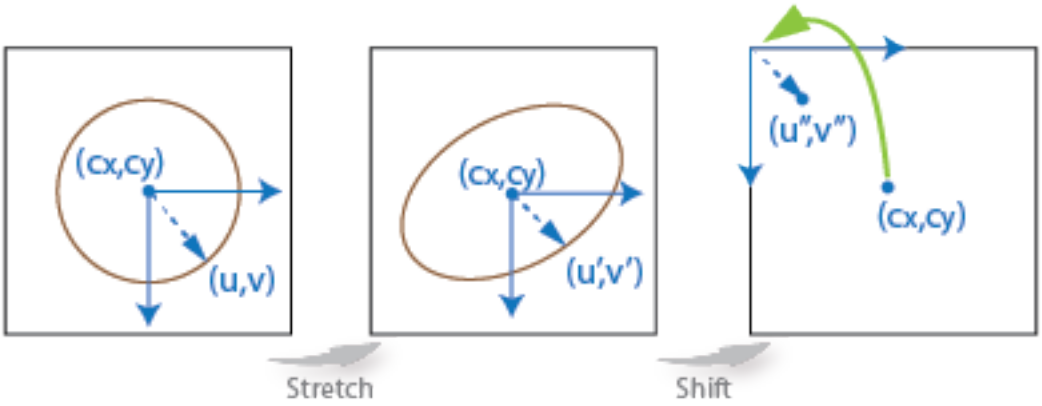
For the fisheye camera model, the intrinsic parameters include the polynomial mapping coefficients of the projection function. The alignment coefficients are related to sensor alignment and the transformation from the sensor plane to a pixel location in the camera image plane.

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

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

- (u,v) are the ideal image projections of the real-world points.
- λ represents a scalar factor.
- a_0, a_2, a_3, a_4 are polynomial coefficents described by the Scaramuzza model, where $a_1 = 0$.
- ρ is a function of (u,v) and depends only on the distance of a point from the image center: $\rho = \sqrt{u^2 + v^2}$.

The intrinsic parameters also account for stretching and distortion. The stretch matrix compensates for the sensor-to-lens misalignment, and the distortion vector adjusts the (0,0) location of the image plane.



The following equation relates the real distorted coordinates (u'',v'') to the ideal distorted coordinates (u,v).

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

Image pixels — Stretch matrix — Hypothetical image plane — Distortion center

Fisheye Camera Calibration in MATLAB

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

Correct Fisheye Image for Lens Distortion

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

Gather a set of checkerboard calibration images.

```
images = imageDatastore('calibrationImages');
```

Detect the calibration pattern from the images. The 'PartialDetections' Name-Value argument is set to true by default allowing detection of partial checkerboards.

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

Generate world coordinates for the corners of the checkerboard squares.

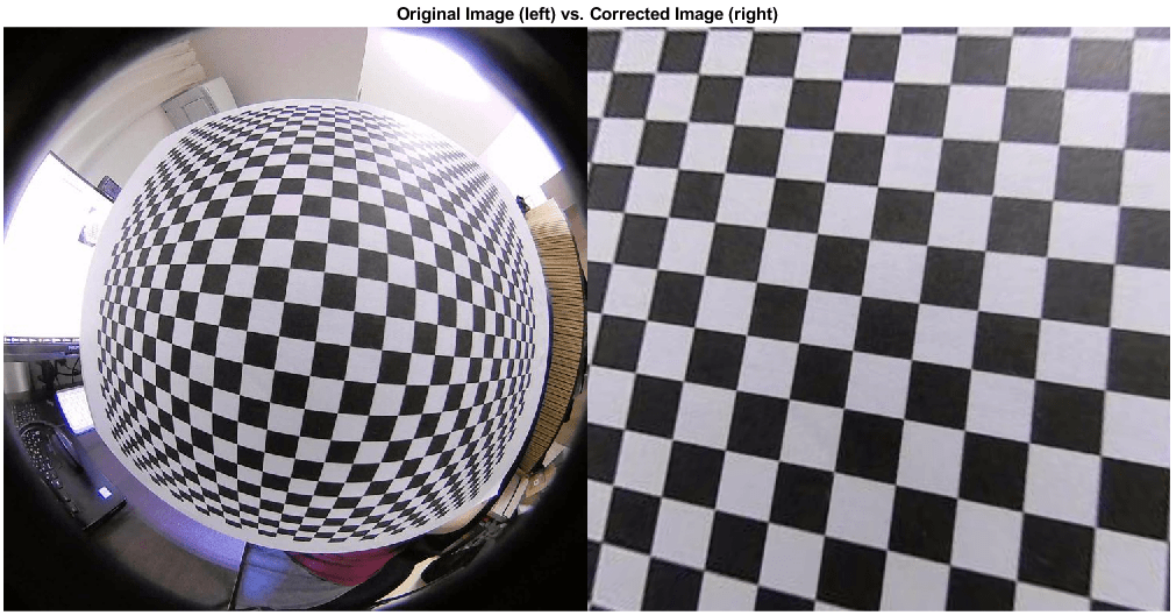
```
squareSize = 20; % millimeters
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

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

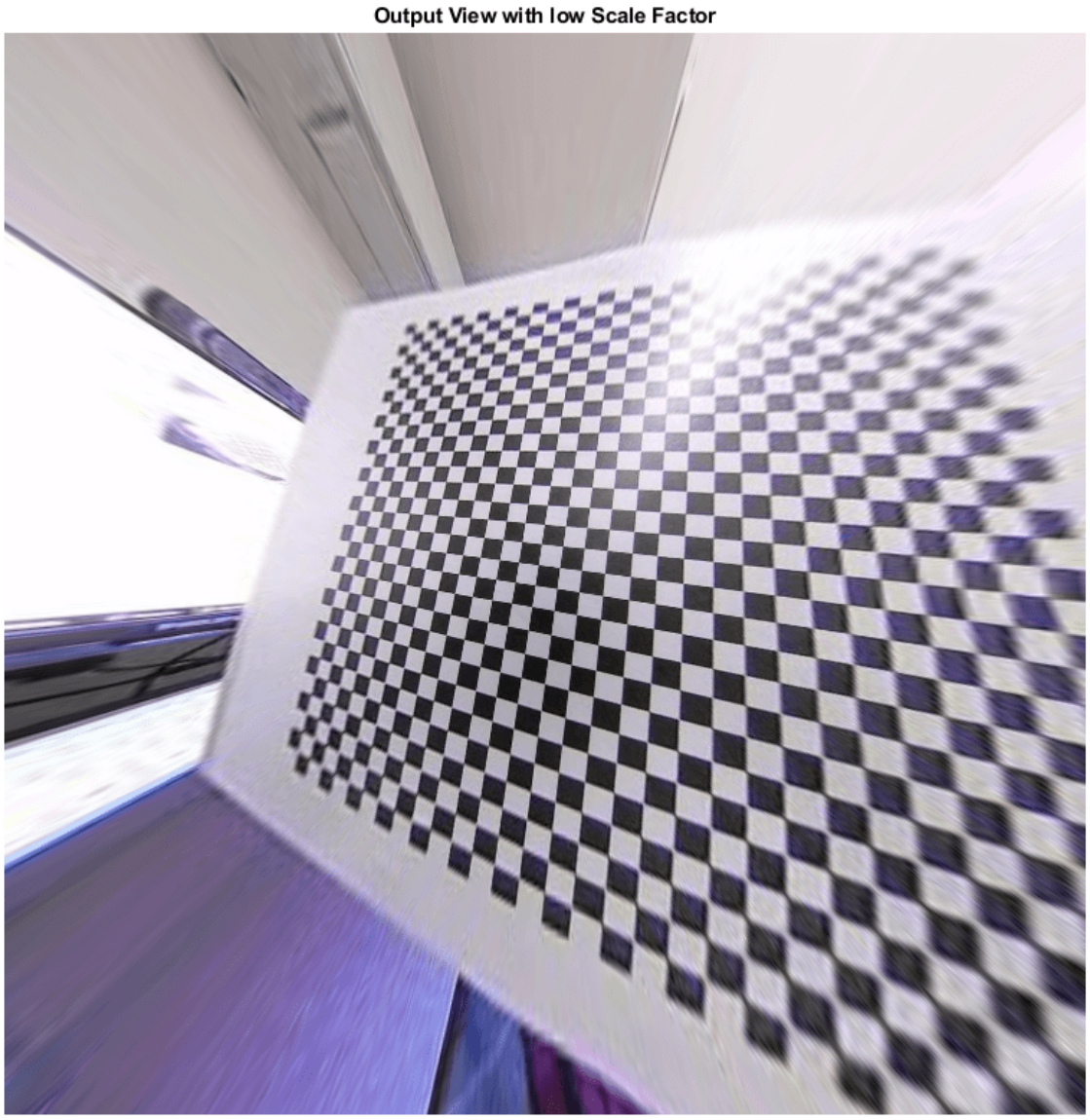
```
I = readimage(images,10);
imageSize = [size(I,1) size(I,2)];
params = estimateFisheyeParameters(imagePoints,worldPoints,imageSize);
```

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

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




```
J2 = undistortFisheyeImage(I,params.Intrinsics,'OutputView','same', 'ScaleFactor', 0.2);
figure
imshow(J2)
title('Output View with low Scale Factor')
```



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 1-92
- “Structure From Motion From Multiple Views” on page 1-141
- “Configure Monocular Fisheye Camera” (Automated Driving Toolbox)

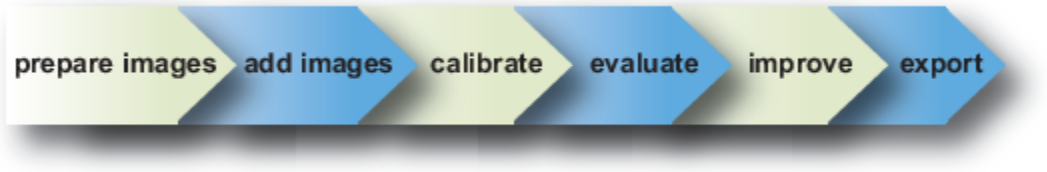
Using the Single Camera Calibrator App

In this section...
“Camera Calibrator Overview” on page 16-18
“Choose a Calibration Pattern” on page 16-18
“Capture Calibration Images” on page 16-18
“Using the Camera Calibrator App” on page 16-19

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, such as 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 “Camera Calibration”.



Follow this workflow to calibrate your camera using the app.

1. Prepare the images, camera, and calibration pattern.
2. Add the images and select standard or fisheye camera model.
3. Calibrate the camera.
4. Evaluate the calibration accuracy.
5. Adjust the parameters to improve the accuracy (if necessary).
6. Export the parameters object.

If the default values work well, then you do not need to make any adjustments before exporting the parameters.

Choose a Calibration Pattern

The **Camera Calibrator** app supports checkerboard, circle grid, and custom detector patterns. For details on each of these patterns and PDF files containing printable patterns, see “Calibration Patterns” on page 16-6.

Capture Calibration Images

For best calibration 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 more details on camera setup and capturing images, see “Prepare Camera and Capture Images” on page 16-4.

Using the Camera Calibrator App

Open the App

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

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 of theselected pattern. For details on camera setup and capturing images, see “Prepare Camera and Capture Images” on page 16-4.

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 repeating the process for each folder.

Acquire Live Images

You can acquire live images from a webcam using MATLAB Webcam support. To use this feature, you must install the MATLAB Support Package for USB Webcams. To add live images, follow these steps.

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

   This opens the **Camera** tab. If only one webcam is connected to your system, the app selects it by default and a live preview pane opens. If you have multiple cameras connected and want to use one other than the default, select that camera in the **Camera** list.
2. (Optional) Set properties for the camera to control the image. Select the **Camera Properties** to open the Camera Properties dialog box for the selected camera. The available properties vary depending on your device.

   Use the sliders and lists to change the available property settings. The preview pane updates dynamically when you change a setting. When you are done setting properties, click anywhere outside of the dialog box to dismiss it.
3. Enter a location to save the acquired image files in the **Save Location** box. You can type the path to a folder or use the **Browse** button. You must have permission to write to the folder you specify.
4. Set the capture parameters.

   - To set the number of seconds between image captures, use the **Capture Interval (sec)** 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.

   The default configuration captures a total of 20 images, one every 5 seconds.

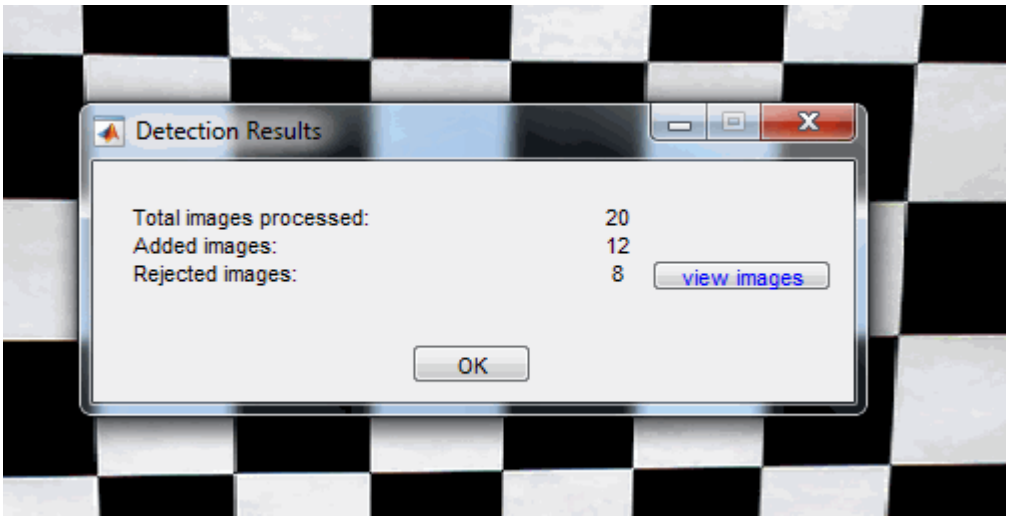
5 The preview pane shows the live images streamed as RGB data. After you adjust any device properties and capture settings, use the Preview window as a guide to line up the camera to acquire the pattern image you want to capture.

6 Select **Capture**. The app captures the specified number of images, 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 stop the image capture before the designated number of images are captured by selecting **Stop Capture**.

When you are capturing images of a pattern, after the designated number of images are captured, the app displays the Image and Pattern Properties dialog box. Select the calibration pattern in the image and specify the pattern properties. Click **OK**.

The app then calculates and displays the detection results.



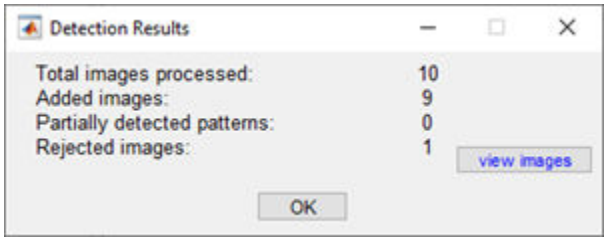
7 To dismiss the Detection Results dialog box, click **OK**.

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

After you add images, the Image and Pattern Properties dialog box to your session, appears. Before the calibrator can analyze the calibration patterns, you must select the calibration pattern to detect and set image properties for the pattern structure. For more details on this dialog, see “Select Calibration Pattern and Set Properties” on page 16-2.

Analyze Images

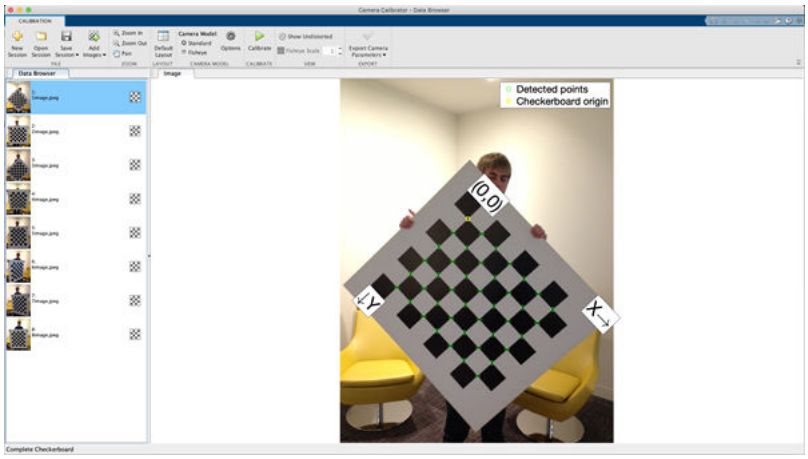
The calibrator attempts to detect a pattern in each of the added images, and displays a progress bar dialog box, that indicates detection progress. If any of the images are rejected, the Detection Results dialog box appears and displays diagnostic information. The results indicate how many total images have been processed and, of those processed, how many have been accepted, rejected, or skipped. The calibrator skips duplicate images.



To view the rejected images, click **view images**. In addition to duplicate images, the calibrator also rejects images where it could not detect the entire pattern. Possible reasons for no detection are a blurry image or an extreme angle of the pattern. Detection takes longer for larger images and patterns that contain a large number of elements.

View Images and Detected Points

The **Data Browser** pane displays a list of images with their IDs. Each image contains a detected pattern. To view an image, select it from the **Data Browser** pane.



The **Image** pane displays the selected checkerboard image with green circles overlaid to indicate detected points. You can verify that the corners have been detected correctly by 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, on the **Calibration** tab, select **Calibrate**. The default calibration settings use a 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 or adding or removing images, and then calibrating again. If you switch between the standard and fisheye camera models, 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 recalibrate after changing the camera model. Select **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 specifications for your camera and know the pixel size, focal length, or lens characteristics, you can manually set initial guesses for the camera intrinsics and radial distortion. To set initial guesses, select **Options > Optimization Options**.

- Select **Specify initial intrinsics as a 3-by-3 matrix of the form [fx 0 0; s fy 0; cx cy 1]**, 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 **Specify initial radial distortion as 2- or 3-element vector**, 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.

For more details on calibration parameters, see “What Is Camera Calibration?” on page 16-43.

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 16-11.

Calibration Algorithm

For fisheye camera model calibration, see “Fisheye Calibration Basics” on page 16-11.

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 `0`. The matrix elements are defined as:

  - $f_x = F*s_x$, expressed in pixels.
  - $f_y = F*s_y$, expressed in pixels.
  - F is the focal length in world units, typically expressed in millimeters.
  - s_x and s_y are the number of pixels per world unit in the x- and y- respectively.

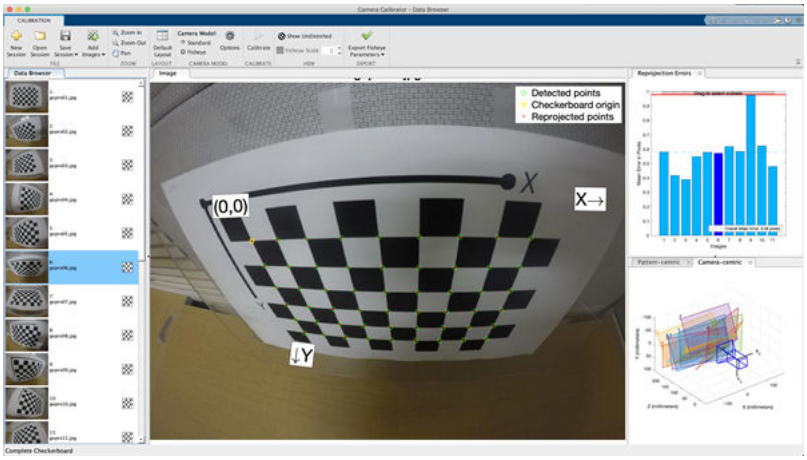
- R — Matrix representing the 3-D rotation of the camera.
- t — Translation of the camera relative to the world coordinate system.

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]

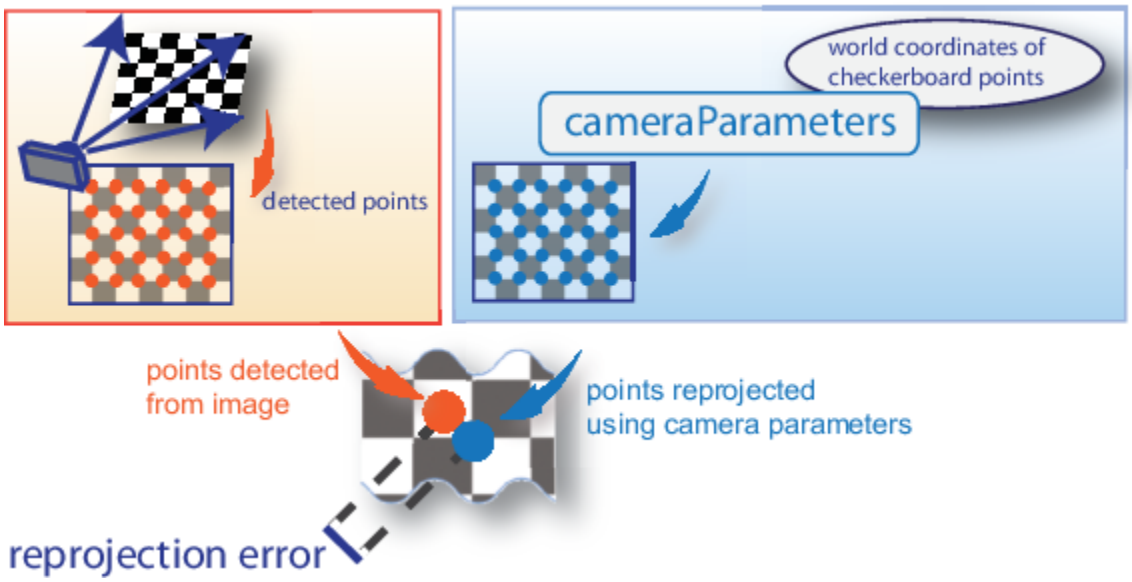
Evaluate Calibration Results

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.



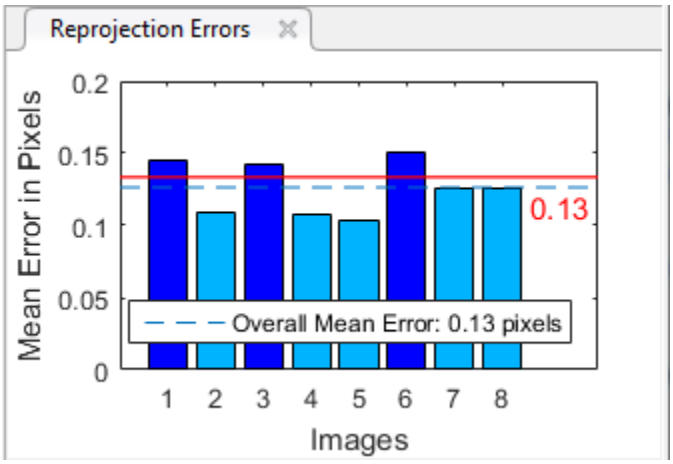
Examine Reprojection Errors

The reprojection errors are the distances, in pixels, between the detected and the corresponding reprojected points. The **Camera Calibrator** app calculates reprojection errors by projecting points from the world coordinates defined by the pattern 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 identify which images adversely impact the calibration. You can select a bar graph entry to select an image, and then 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.

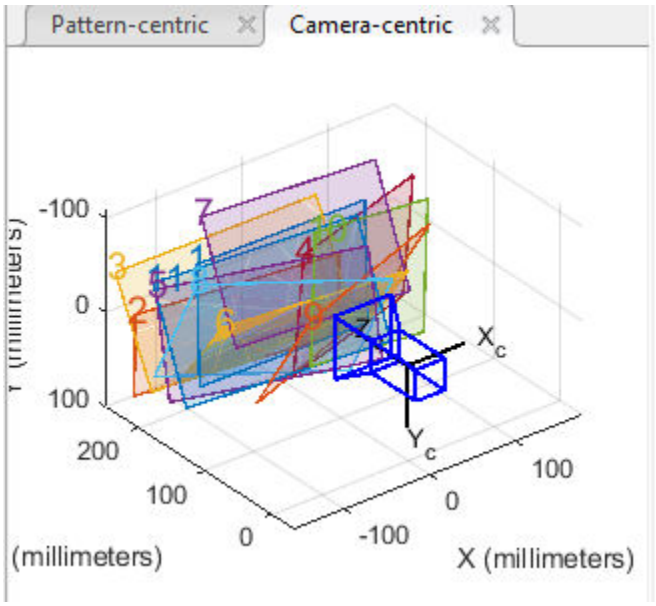


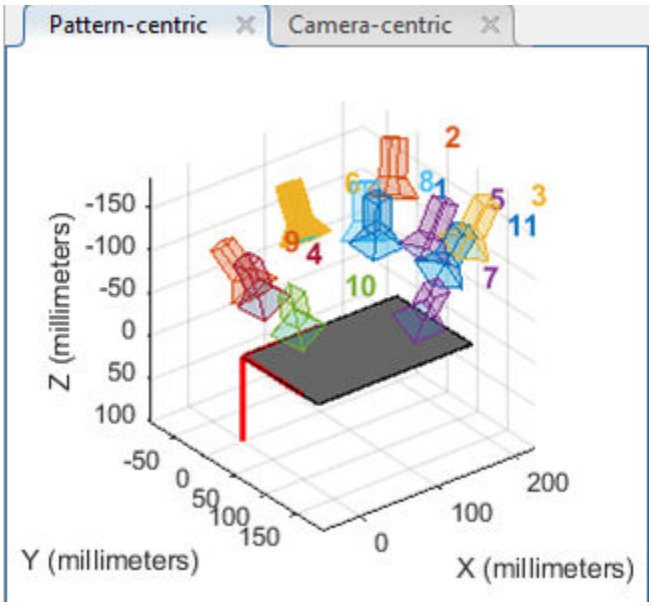
Select an image in one of these ways:

- 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. Slide the red line up or down to automatically select all images with a mean error greater than the specified value.

Examine Extrinsic Parameter Visualization

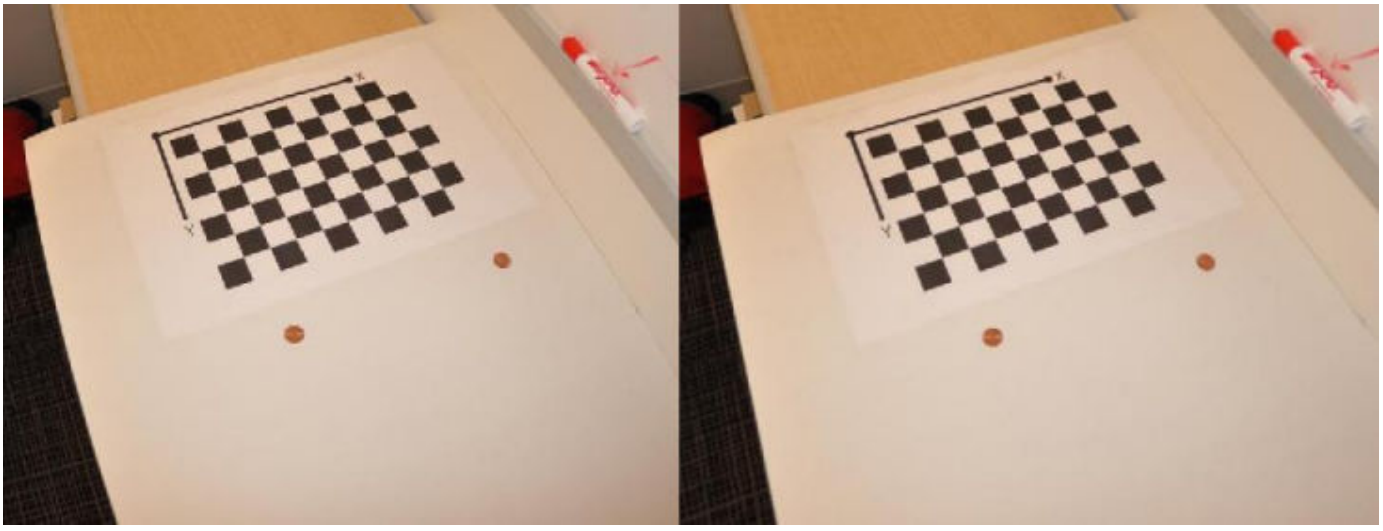
The 3-D extrinsic parameters plot provides a camera-centric view of the patterns and a pattern-centric view of the camera. The camera-centric view is helpful if the camera is stationary when the images are captured. The pattern-centric view is helpful if the pattern is stationary. You can click and drag the figure to rotate it. Click a checkerboard (or camera) to select it. The highlighted data in the visualizations corresponds 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.





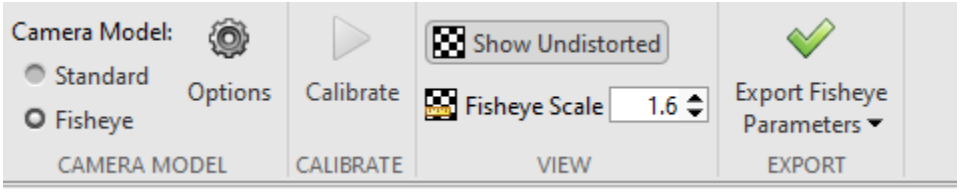
View Undistorted Image

To view the effects of removing lens distortion, on the **Calibration** tab, in the **View** section, select **Show Undistorted** in the **View** section of the **Calibration** tab. If the calibration is accurate, the distorted lines in the image preview become straight.



Note 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 can be incorrect, even though the calibration resulted in few reprojection errors. This image shows an example of this type of incorrect estimation for a single camera calibration.

For the fisheye camera model, while viewing the undistorted images, you can examine the fisheye images more closely by, on the **Calibration** tab, in the **View** section, specifying the **Fisheye Scale**. Enter a value in the **Fisheye Scale** box, or use the arrows to adjust the scale up or down.



Improve Calibration

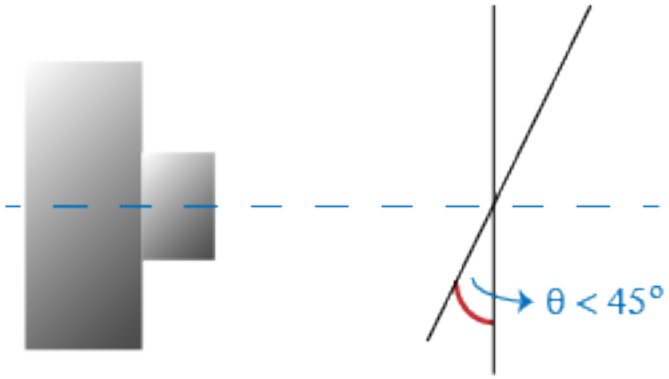
To improve the calibration, you can remove high-error images, add more images, or modify the calibrator settings.

Consider adding more images if:

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

Consider removing images if the images:

- Have a high mean reprojection error.
- Are blurry.
- Contain a calibration pattern at an angle greater than 45 degrees relative to the camera plane.



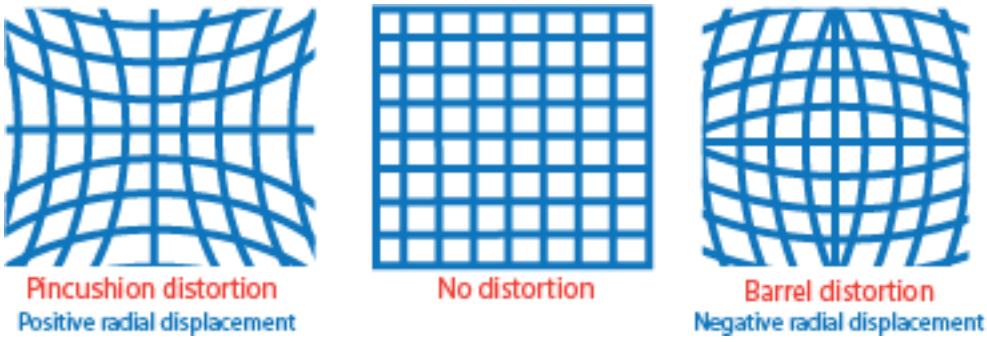
- Incorrectly detected calibration pattern points.

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**. Specify the **Radial Distortion** as either two or three coefficients by selecting **2 Coefficients** or **3 Coefficients**, respectively.

Radial distortion is the displacement of image points along radial lines extending from the principal point.

- As image points move away from the principal point (positive radial displacement), image magnification decreases and a pincushion-shaped distortion occurs on the image.
- As image points move toward the principal point (negative radial displacement), image magnification increases and a barrel-shaped distortion occurs on the image.



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

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

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

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

- $r^2 = x^2 + y^2$

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

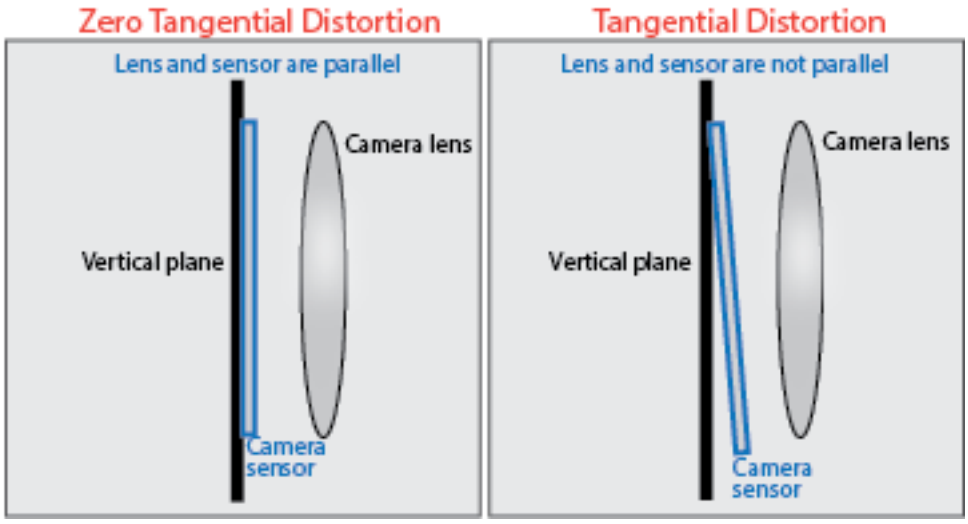
The undistorted pixel locations are in normalized image coordinates, with the origin at the optical center. The coordinates are expressed in world units.

Standard Model: Compute Skew

To estimate the skew of the image axes, on the **Calibration** tab, in the **Camera Model** section, select **Options > Compute > 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 this option, the image axes are perpendicular, which is true 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 $(x_{distorted}, y_{distorted})$:

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

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

- 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.
- $r^2 = x^2 + y^2$

To estimate the tangential distortion coefficients, on the **Calibration** tab, in the **Camera Model** section, select **Options > Compute > Tangential Distortion**. Otherwise, the calibrator sets the tangential distortion coefficients to zero.

Fisheye Model: Estimate Alignment

On the**Calibration** tab, in **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 your calibration accuracy, select **Export Camera Parameters** for a standard camera model or **Export Camera Parameters** for a fisheye camera model. You can either export the camera parameters to an object in the MATLAB workspace or generate the camera parameters as a MATLAB script.

Export Camera Parameters

Under **Export Camera Parameters**, for a standard camera model, or **Export Fisheye Parameters**, for a fisheye camera model, select **Export Parameters to Workspace** to create a `cameraParameters` object in your workspace. The object contains the intrinsic and extrinsic parameters of the camera and its distortion coefficients. You can use this object for various computer vision tasks, such as image undistortion, measuring planar objects, and 3-D reconstruction. For more information on measuring planar objects, see “Measuring Planar Objects with a Calibrated Camera” on page 1-122. You can optionally export a `cameraCalibrationErrors` object that contains the standard errors of the estimated camera parameters by selecting **Export estimation errors**.

Generate MATLAB Script

Under **Export Camera Parameters**, for a standard camera model, or **Export Fisheye Parameters**, for a fisheye camera model, select **Generate MATLAB script** to save your camera parameters to a MATLAB script, enabling you to reproduce the steps from your calibration session.

Note You cannot generate a MATLAB script for custom pattern camera parameters defined using the `vision.calibration.PatternDetector` class.

References

[1] Zhang, Z. “A Flexible New Technique for Camera Calibration.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 22, no. 11 (November 2000): 1330–34. https://doi.org/10.1109/34.888718.

[2] Heikkila, J., and O. Silven. “A Four-step Camera Calibration Procedure with Implicit Image Correction.” In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1106–12. San Juan, Puerto Rico: IEEE Comput. Soc, 1997. https://doi.org/10.1109/CVPR.1997.609468.

[3] Scaramuzza, Davide, Agostino Martinelli, and Roland Siegwart. "A Toolbox for Easily Calibrating Omnidirectional Cameras." In *Proceedings of IEEE International Workshop on Intelligent Robots and Systems 2006 (IROS 2006)*, 5695–701. Bejing, China: IEEE, 2006. https://doi.org/10.1109/IROS.2006.282372

[4] Urban, Steffen, Jens Leitloff, and Stefan Hinz. “Improved Wide-Angle, Fisheye and Omnidirectional Camera Calibration.” *ISPRS Journal of Photogrammetry and Remote Sensing* 108 (October 2015): 72–79. https://doi.org/10.1016/j.isprsjprs.2015.06.005.

See Also

Apps
Camera Calibrator | **Stereo Camera Calibrator**

Functions
`showReprojectionErrors` | `showExtrinsics` | `undistortImage` | `detectCheckerboardPoints` | `generateCheckerboardPoints` | `estimateCameraParameters`

Objects
`cameraParameters` | `stereoParameters`

Related Examples

- “Prepare Camera and Capture Images” on page 16-4
- “Select Calibration Pattern and Set Properties” on page 16-2
- “Evaluating the Accuracy of Single Camera Calibration” on page 1-117
- “Measuring Planar Objects with a Calibrated Camera” on page 1-122
- “Camera Calibration Using AprilTag Markers” on page 1-50
- “Structure From Motion From Two Views” on page 1-92
- “Structure From Motion From Multiple Views” on page 1-141
- “Depth Estimation From Stereo Video” on page 1-131
- “3-D Point Cloud Registration and Stitching” on page 5-71
- “Uncalibrated Stereo Image Rectification” on page 1-149
- Checkerboard pattern
- Symmetric Circles Grid Pattern
- Asymmetric Circles Grid Pattern

More About

- “Using the Single Camera Calibrator App” on page 16-18
- “Using the Stereo Camera Calibrator App” on page 16-31
- “Coordinate Systems”
- “Implement Visual SLAM in MATLAB” on page 11-8
- Camera Calibration with MATLAB

Using the Stereo Camera Calibrator App

In this section...
“Stereo Camera Calibrator Overview” on page 16-31
“Choose a Calibration Pattern” on page 16-32
“Capture Calibration Images” on page 16-32
“Using the Stereo Camera Calibrator App” on page 16-32

Stereo Camera Calibrator Overview

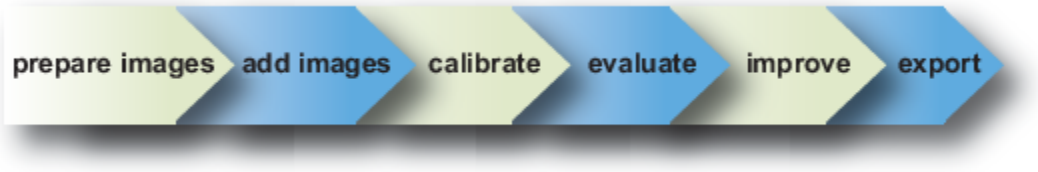
You can use the **Stereo Camera Calibrator** app to calibrate a stereo camera, which you can then use to recover depth from images. A stereo system consists of two cameras: camera 1 and camera 2. The app can either estimate or import the parameters of individual cameras. The app also calculates the position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

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



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 “Camera Calibration”.

Choose a Calibration Pattern

The **Stereo Camera Calibrator** app supports checkerboard, circle grid, and custom detector patterns. For details on each of these patterns, see “Calibration Patterns” on page 16-6.

Capture Calibration Images

For best calibration 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 more details on camera setup and capturing images, see “Prepare Camera and Capture Images” on page 16-4.

Using the Stereo Camera Calibrator App

Open the App

- 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`

Add Image Pairs 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 pattern. For details on camera setup and capturing images, see “Prepare Camera and Capture Images” on page 16-4

Add Images from File

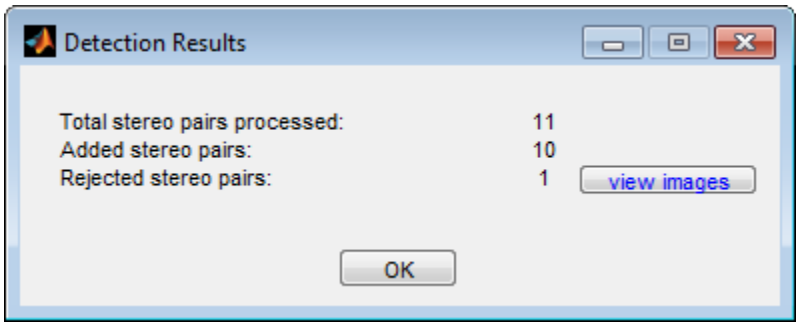
On the **Calibration** tab, in the **File** section, click **Add Images**, and then select `From file`. You can add images from multiple folders by clicking **Add images** for each folder. Enter the location for the images corresponding to camera 1, or select it using the **Browse** button, and then do the same for camera 2. Specify the calibration pattern by selecting one from the **Choose Pattern** list, or, in the **Custom Pattern** section, select **Import Pattern Detector**. In the **Properties** section, specify the properties for your detector, and then select **OK** to add your images.

After you load images, the **Image and Pattern Properties** dialog appears. Before the calibrator can analyze the calibration patterns, you must select the calibration pattern to detect and set image properties for the pattern structure. For more details on this dialog, see “Select Calibration Pattern and Set Properties” on page 16-2.

Analyze Images

The calibrator attempts to detect a pattern in each of the added stereo pairs, displaying a 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

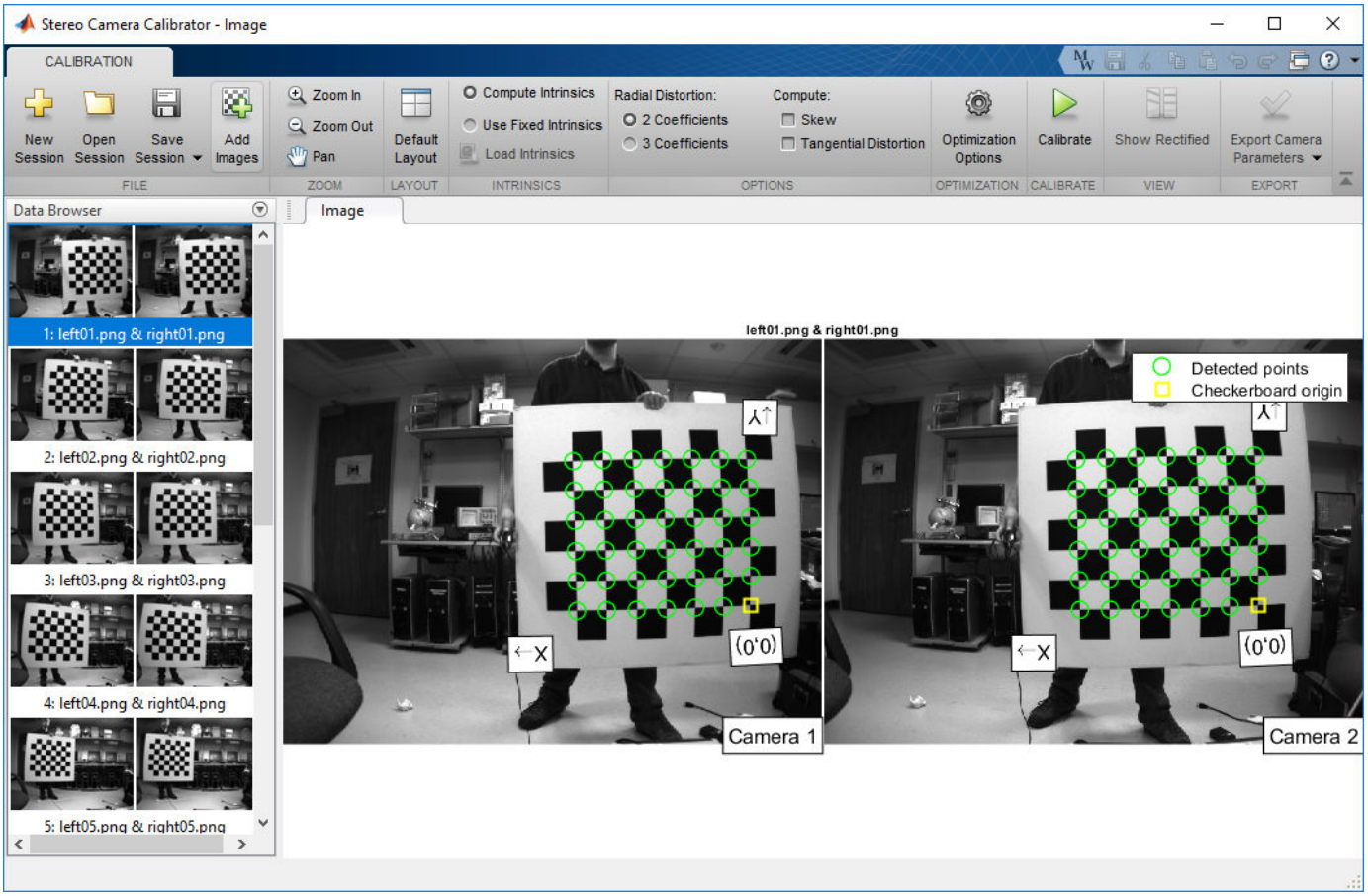
images were processed, and of those processed, how many were accepted, rejected, or skipped. The calibrator skips duplicate images.



To view the rejected images, click **View images**. The calibrator rejects duplicate images. It also rejects images where the entire pattern 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 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 pattern axes orientation.

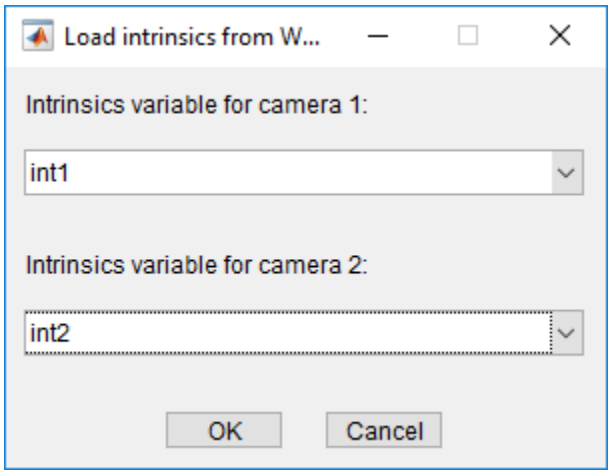
Intrinsics

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

To load intrinsics as variables from your workspace, select **Load Intrinsics**. For example, if the `wideBaselineStereo` structure contains the intrinsics for both cameras, enter this code at the MATLAB command prompt.

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

Then, select **Load Intrinsics** to specify these variables in the Load intrinsics from Workspace dialog box.



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

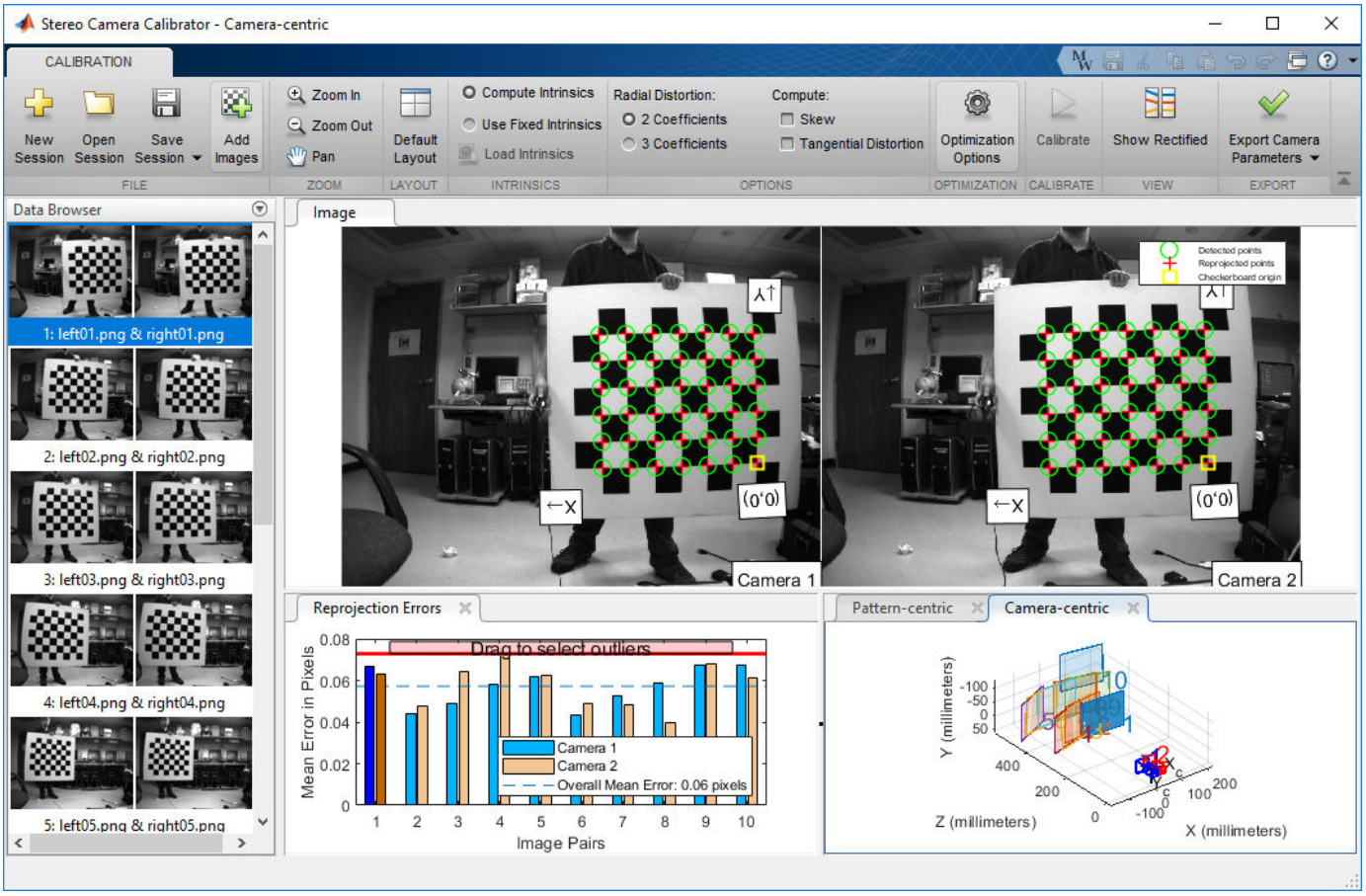
When the camera has severe lens distortion, the app can fail to compute the initial values for the camera intrinsics. If you have the manufacturer specifications for your camera and know the pixel size, focal length, or lens characteristics, you can manually set initial guesses for the camera intrinsics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

- Select **Specify initial intrinsics as a 3-by-3 matrix of the form [fx 0 0; s fy 0; cx cy 1]**, 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 **Specify initial radial distortion as 2- or 3-element vector**, 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.

For more details on calibration parameters, see “What Is Camera Calibration?” on page 16-43.

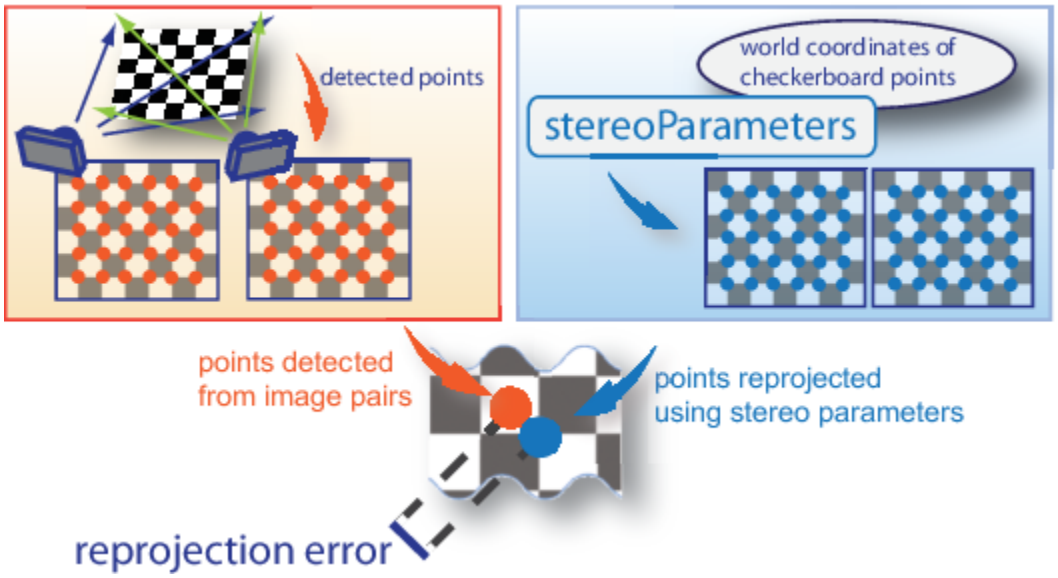
Evaluate Calibration Results

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.



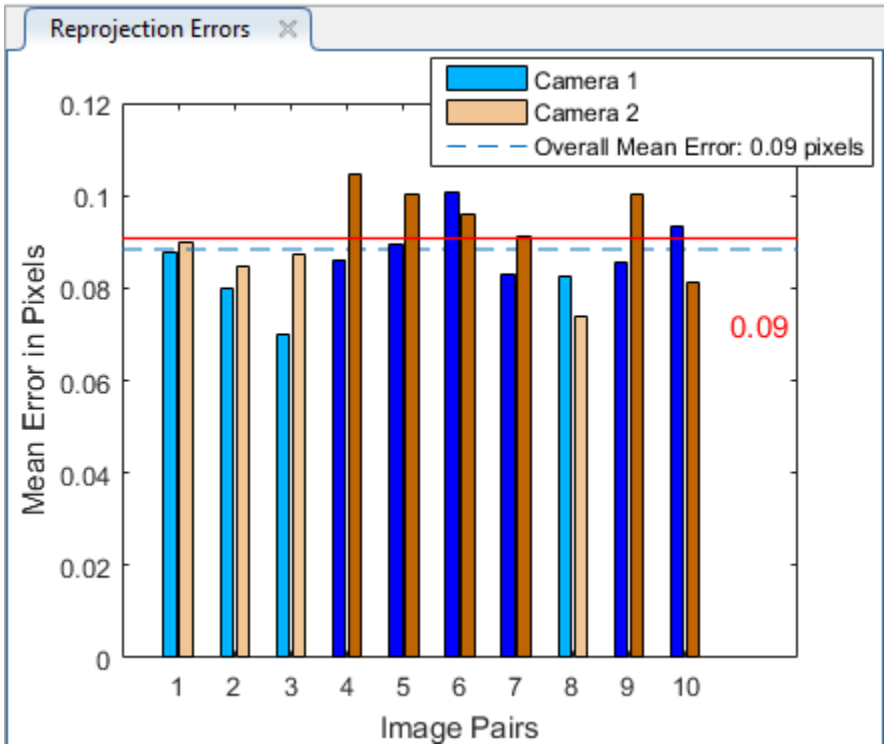
Examine Reprojection Errors

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 points from world coordinates, defined by the pattern, 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 pair IDs. The highlighted bars correspond to the selected image pair.

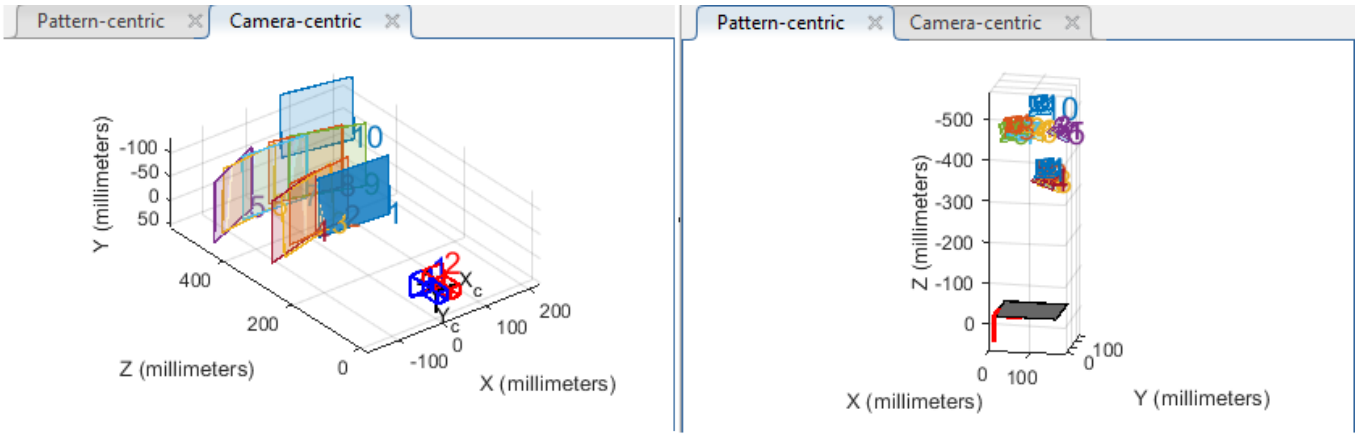


Select an image pair in one of these ways:

- Click a corresponding bar in the graph.
- Select an 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 pairs containing an image with a mean error greater than the specified value.

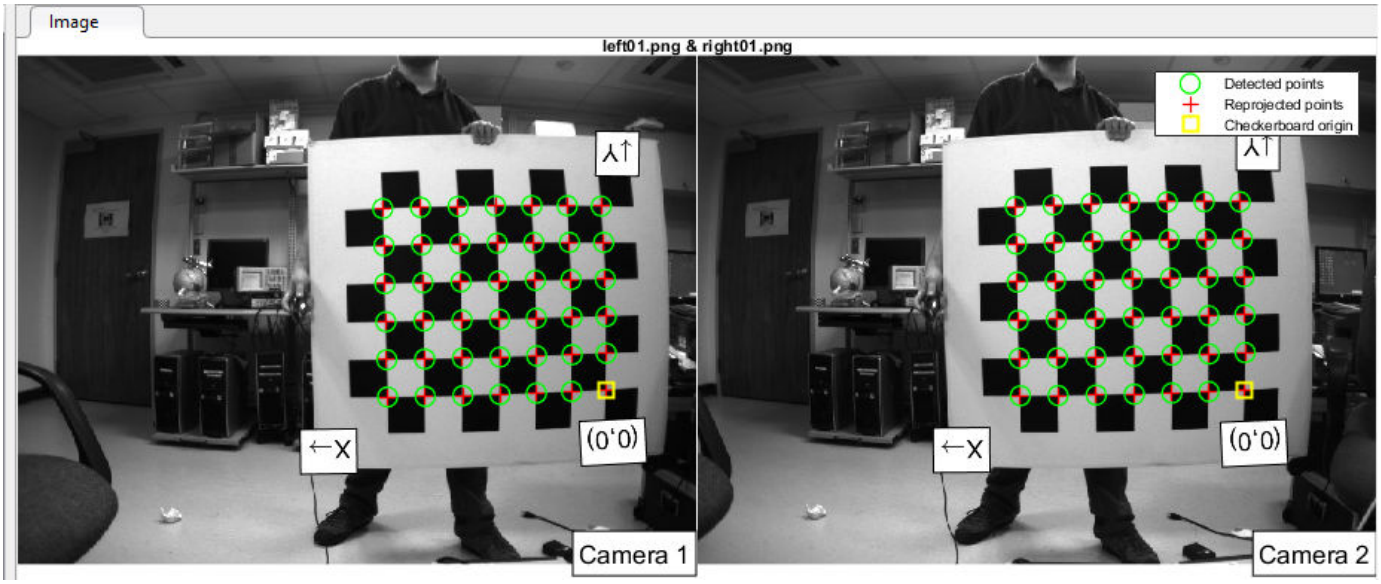
Examine Extrinsic Parameter Visualization

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 pattern (or the camera) in the display to select it. The highlighted data in the visualizations correspond to the selected image pair 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, on the **Calibration** tab, in the **View** section, select **Show Rectified**. If the calibration is accurate, the images become undistorted and row-aligned.



Note 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

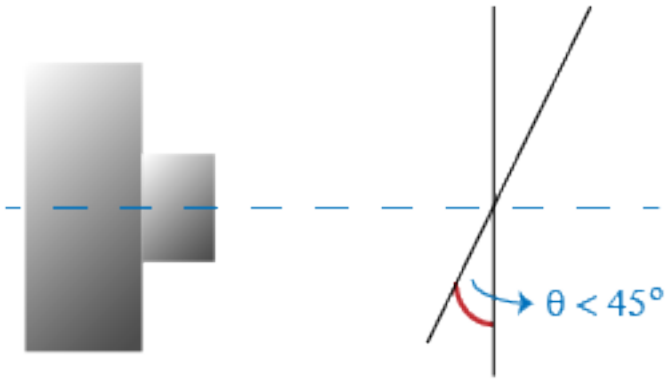
To improve the calibration, you can remove high-error images, add more images, or modify the calibrator settings.

Consider adding more image pairs if:

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

Consider removing image pairs if the images:

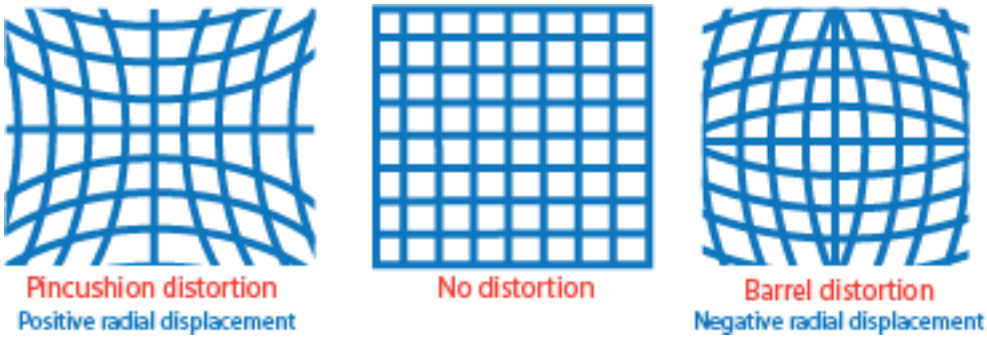
- Have a high mean reprojection error.
- Are blurry.
- Contain a calibration pattern at an angle greater than 45 degrees relative to the camera plane.



- Incorrectly detected calibration pattern points.

Change the Number of Radial Distortion Coefficients

You can specify two or three radial distortion coefficients by selecting the corresponding option 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 $(x_{distorted}, y_{distorted})$:

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

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

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

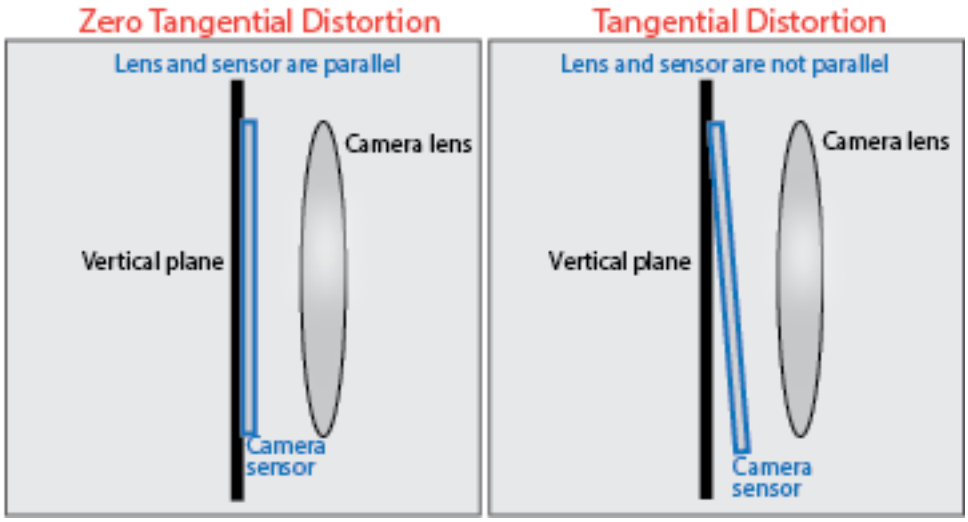
Typically, two coefficients are sufficient for calibration. For severe distortion, such as in wide-angle lenses, you can select three coefficients to include k_3.

Compute Skew

To estimate the skew of the image axes, on the **Calibration** tab, in the **Camera Model** section, select **Options > Compute > 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 this option, the image axes are perpendicular, which is true 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 $(x_{distorted}, y_{distorted})$:

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

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

- 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 your calibration accuracy, select **Export Camera Parameters**. You can either save and export the camera parameters to an object in the MATLAB workspace, 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 its distortion coefficients. You can use this object for various computer vision tasks, such as image undistortion, measuring planar objects, and 3-D reconstruction. For more information on measuring planar objects, see "Measuring Planar Objects with a Calibrated Camera" on page 1-122. You can optionally export the `stereoCalibrationErrors` object, which contains the standard errors of estimated stereo camera parameters, by selecting **Export estimation errors**.

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.

Note You cannot generate a MATLAB script for custom pattern camera parameters defined using the `vision.calibration.PatternDetector` class.

References

[1] Zhang, Z. "A Flexible New Technique for Camera Calibration". *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 22, no. 11 (November 2000): 1330–34. https://doi.org/10.1109/34.888718.

[2] Heikkila, J., and O. Silven. "A Four-step Camera Calibration Procedure with Implicit Image Correction." In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1106–12. San Juan, Puerto Rico: IEEE Comput. Soc, 1997. https://doi.org/10.1109/CVPR.1997.609468.

See Also

Apps
Camera Calibrator | **Stereo Camera Calibrator**

Functions
`showReprojectionErrors` | `showExtrinsics` | `undistortImage` | `detectCheckerboardPoints` | `generateCheckerboardPoints` | `estimateCameraParameters`

Objects
`cameraParameters` | `stereoParameters`

Related Examples

- "Evaluating the Accuracy of Single Camera Calibration" on page 1-117
- "Measuring Planar Objects with a Calibrated Camera" on page 1-122
- "Camera Calibration Using AprilTag Markers" on page 1-50
- "Structure From Motion From Two Views" on page 1-92
- "Structure from Motion from Multiple Views" on page 16-50
- "Depth Estimation From Stereo Video" on page 1-131
- "3-D Point Cloud Registration and Stitching" on page 5-71
- "Uncalibrated Stereo Image Rectification" on page 1-149
- Checkerboard pattern

- Symmetric Circles Grid Pattern
- Asymmetric Circles Grid Pattern

More About

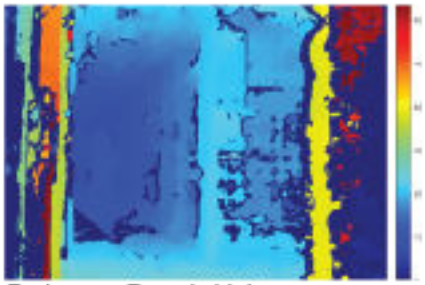
- “Using the Single Camera Calibrator App” on page 16-18
- “Implement Visual SLAM in MATLAB” on page 11-8
- “Coordinate Systems”
- Camera Calibration with MATLAB

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:

Remove Lens Distortion


Estimate Depth Using a Stereo Camera

Measure Planar Objects


Estimate 3-D Structure from Camera Motion

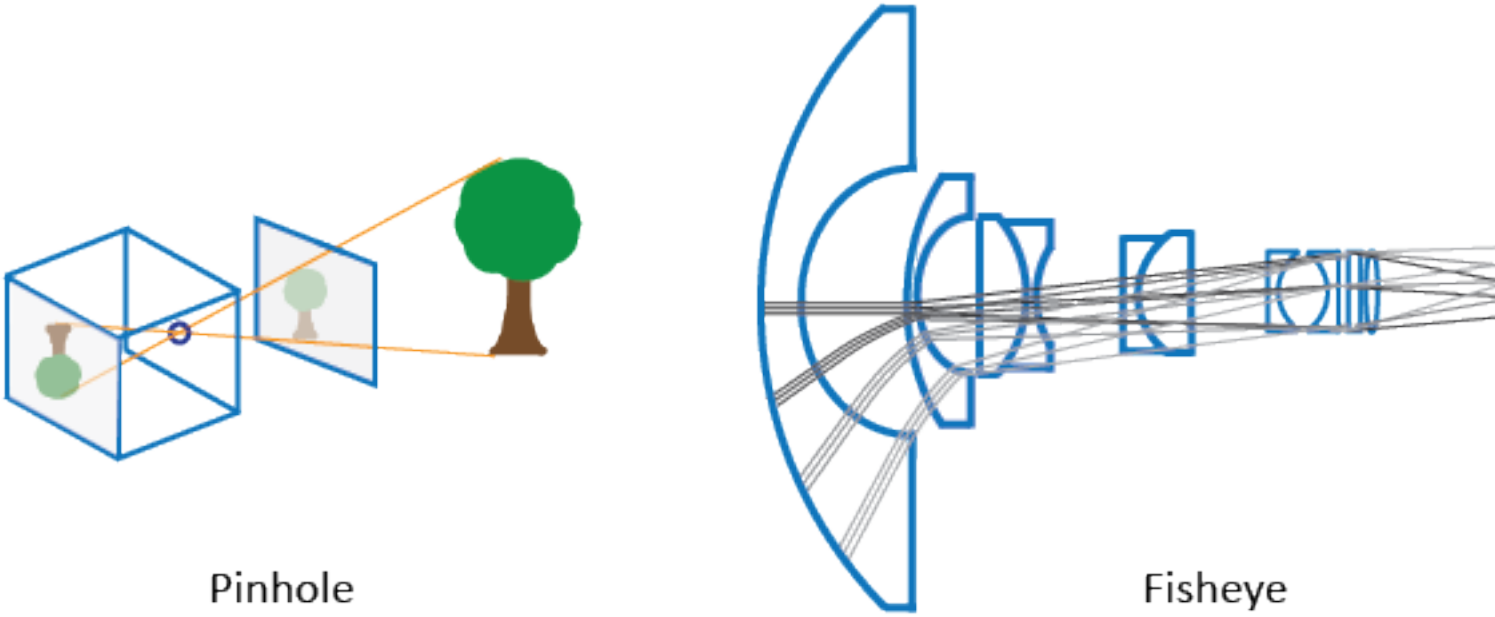
Camera parameters include intrinsics, extrinsics, and distortion coefficients. To estimate the camera parameters, you need to have 3-D world points and their corresponding 2-D image points. You can get these correspondences using multiple images of a calibration pattern, such as a checkerboard. Using 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. You can use the fisheye model with cameras up to a field of view (FOV) of 195 degrees.

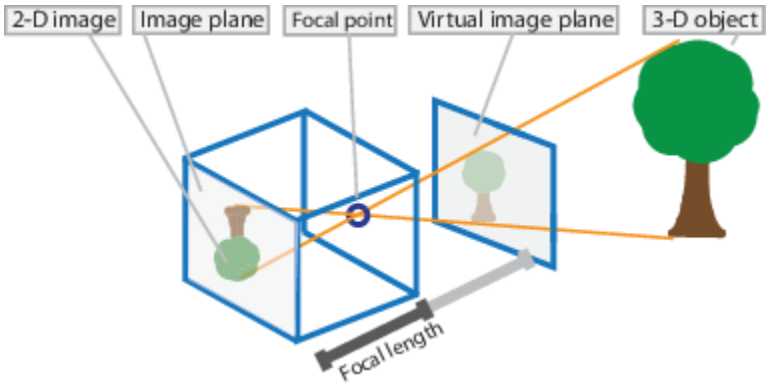


The pinhole calibration algorithm is based on the model proposed by Jean-Yves Bouguet [3]. The model includes, the pinhole camera model [1] and lens distortion [2].The pinhole camera model does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent 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 16-11.

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.

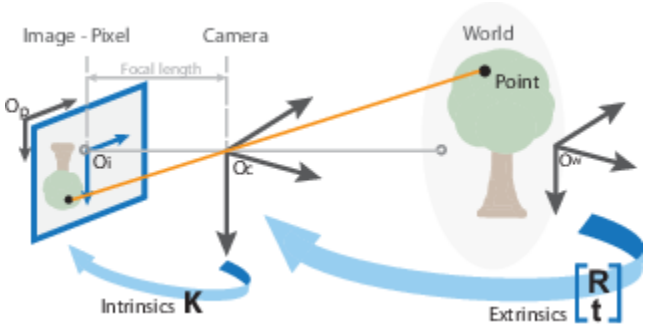
$$w\,[x\ y\ 1] = [X\ Y\ Z\ 1]\,P$$

Scale factor — Image points — World points

$$P = \begin{bmatrix} R \\ t \end{bmatrix} K$$

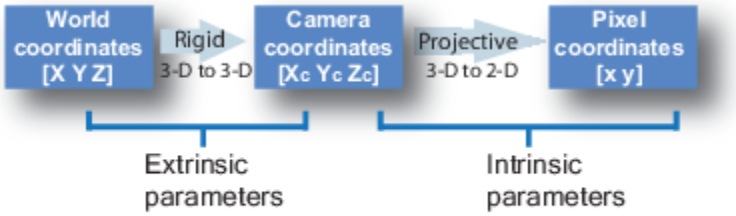
Camera matrix — Extrinsics: Rotation and translation — Intrinsic matrix

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

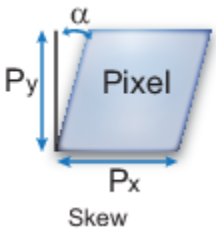
The extrinsic parameters consist of a rotation, R, and a translation, t. The origin of the camera's coordinate system is at its optical center and its x- and y-axis define the image plane.

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:



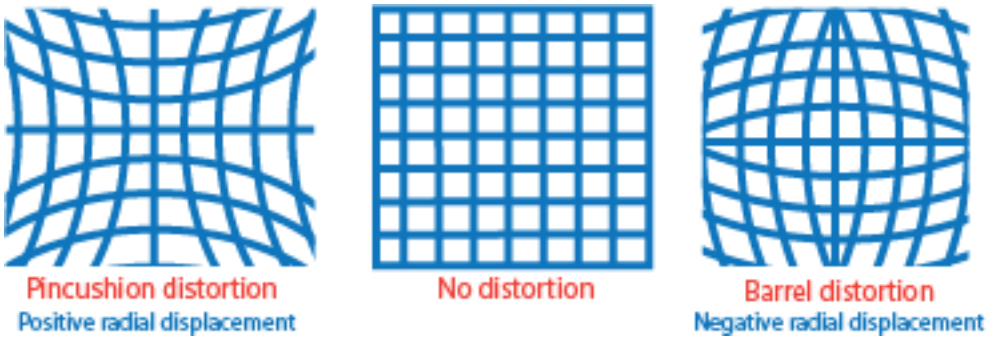
$[c_x\ c_y]$ — Optical center (the principal point), in pixels.
$(f_x,\ f_y)$ — Focal length in pixels.
$f_x = F/p_x$
$f_y = F/p_y$
F — Focal length in world units, typically expressed in millimeters.
$(p_x,\ p_y)$ — Size of the pixel in world units.
s — Skew coefficient, which is non-zero if the image 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 $(x_{distorted}, y_{distorted})$:

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

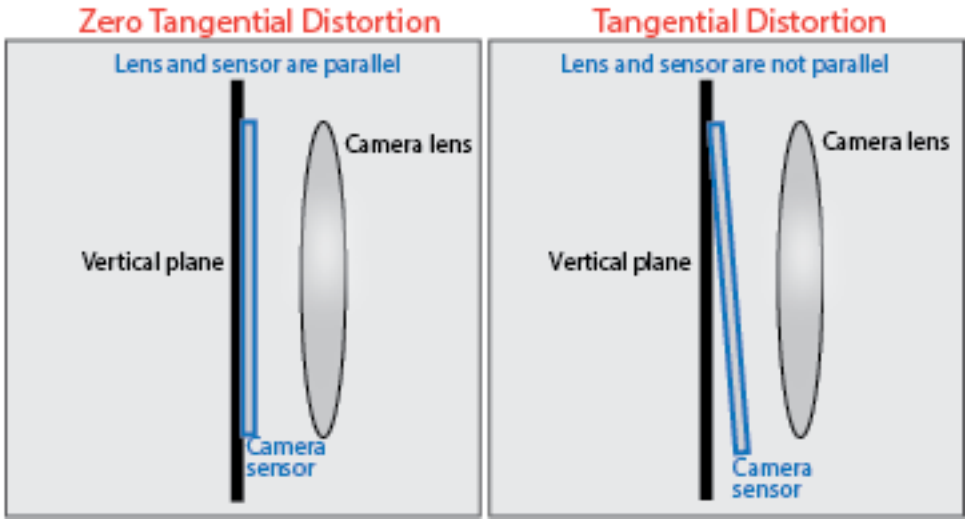
$$y_{distorted} = y(1 + k_1*r^2 + k_2*r^4 + k_3*r^6)$$

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

Typically, two coefficients are sufficient for calibration. For severe distortion, such as in wide-angle lenses, you can select three 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 ($x_{distorted}$, $y_{distorted}$):

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

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

- 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

- "Implement Visual SLAM in MATLAB" on page 11-8
- "Using the Single Camera Calibrator App" on page 16-18
- "Using the Stereo Camera Calibrator App" on page 16-31
- "Evaluating the Accuracy of Single Camera Calibration" on page 1-117
- "Fisheye Calibration Basics" on page 16-11
- "Configure Monocular Fisheye Camera" (Automated Driving Toolbox)
- "Calibrate a Monocular Camera" (Automated Driving Toolbox)
- "Measuring Planar Objects with a Calibrated Camera" on page 1-122
- "Structure From Motion From Two Views" on page 1-92
- "Structure From Motion From Multiple Views" on page 1-141

Structure from Motion Overview

In this section...
“Structure from Motion from Two Views” on page 16-49 “Structure from Motion from Multiple Views” on page 16-50

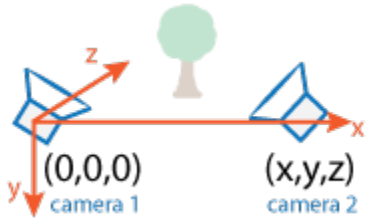
Structure from motion (SfM) is the process of estimating the 3-D structure of a scene from a set of 2-D images. SfM is used in many applications, such as 3-D scanning , augmented reality, and visual simultaneous localization and mapping (vSLAM).

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:

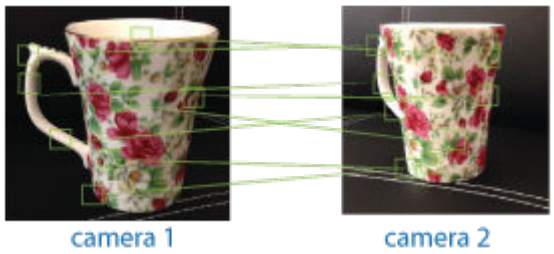
- The size of an object in the scene
- Information from another sensor, for example, an odometer.

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.

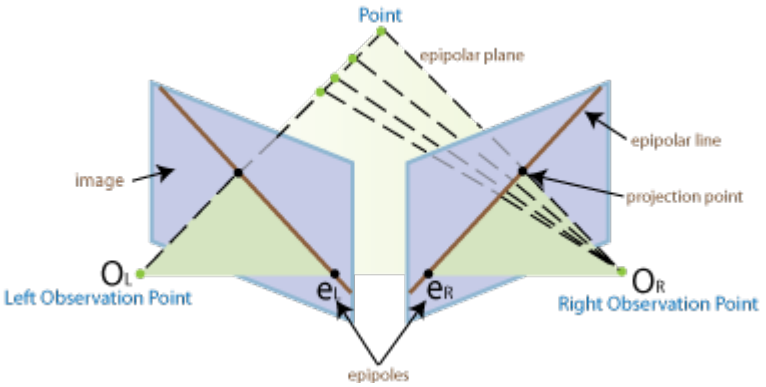


1. SfM requires point correspondences between images. Find corresponding points either by matching features or tracking points from image 1 to image 2. Feature tracking techniques, such as Kanade-Lucas-Tomasi (KLT) algorithm, work well when the cameras are close together. As cameras move further apart, 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 `matchFeatures`	"Find Image Rotation and Scale Using Automated Feature Matching" on page 4-29
Narrow	Track features using `vision.PointTracker`	"Face Detection and Tracking Using the KLT Algorithm" on page 8-20

2 To find the pose of the second camera relative to the first camera, you must compute the fundamental matrix. Use the corresponding points found in the previous step for the computation. The fundamental matrix describes the epipolar geometry of the two cameras. It relates a point in one camera to an epipolar line in the other camera. Use the `estimateFundamentalMatrix` function to estimate the fundamental matrix.



3 Input the fundamental matrix to the `relativeCameraPose` function. `relativeCameraPose` returns the orientation and the location of the second camera in the coordinate system of the first camera. The location can only be computed up to scale, so the distance between two cameras is set to 1. In other words, the distance between the cameras is defined to be 1 unit.

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

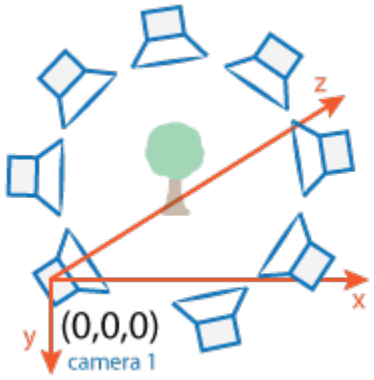
The `triangulate` function takes two camera matrices, which you can compute using `cameraMatrix`.

5 Use `pcshow` or `pcplayer` to display the reconstruction. Use `plotCamera` to visualize the camera poses.

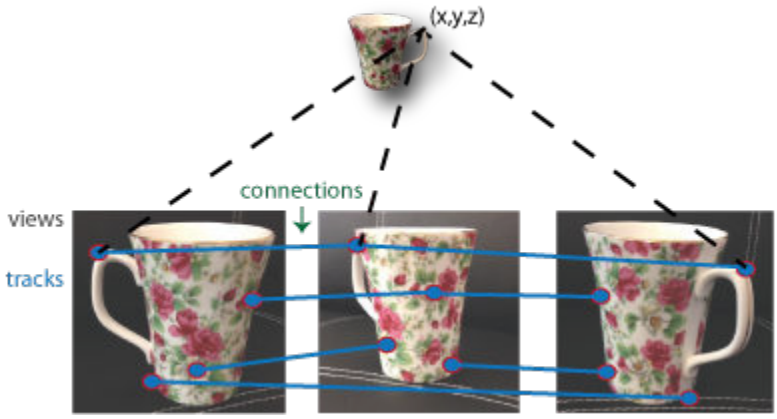
To recover the scale of the reconstruction, you need additional information. One method to recover the scale is to detect an object of a known size in the scene. The "Structure From Motion From Two Views" on page 1-92 example shows how to recover scale by detecting a sphere of a known size in the point cloud of the scene.

Structure from Motion from Multiple Views

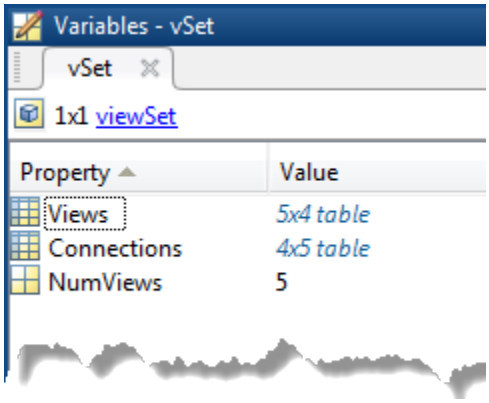
For most applications, such as robotics and autonomous driving, SfM uses more than two views.



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.

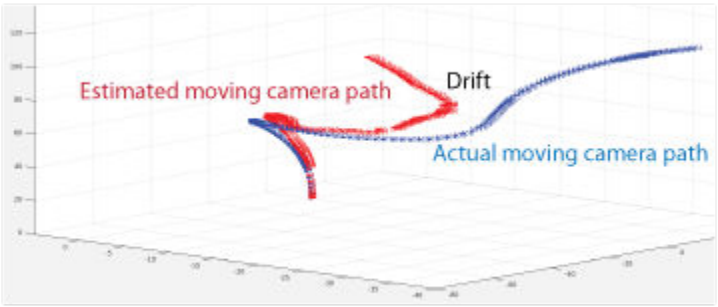


Using the approach in SfM from two views, you can find the pose of camera 2 relative to camera 1. To extend this approach to the multiple view case, find the pose of camera 3 relative to camera 2, and so on. The relative poses must be transformed into a common coordinate system. Typically, all camera 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 anew 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 1-92 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 1-72 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
Stereo Camera Calibrator | Camera Calibrator

Functions
`bundleAdjustment` | `bundleAdjustmentStructure` | `bundleAdjustmentMotion` | `relativeCameraPose` | `triangulateMultiview` | `estimateFundamentalMatrix` | `matchFeatures` | `pointTrack` | `cameraMatrix`

Objects
`imageviewset` | `worldpointset` | `vision.PointTracker`

See Also

Related Examples

- “Structure From Motion From Two Views” on page 1-92
- “Structure From Motion From Multiple Views” on page 1-141
- “Monocular Visual Simultaneous Localization and Mapping” on page 1-72

Object Detection

- “Getting Started with Anomaly Detection Using Deep Learning” on page 17-2
- “Getting Started with Video Classification Using Deep Learning” on page 17-4
- “Choose an Object Detector” on page 17-14
- “Getting Started with SSD Multibox Detection” on page 17-21
- “Getting Started with Object Detection Using Deep Learning” on page 17-24
- “How Labeler Apps Store Exported Pixel Labels” on page 17-27
- “Anchor Boxes for Object Detection” on page 17-32
- “Getting Started with YOLO v2” on page 17-37
- “Getting Started with YOLO v3” on page 17-41
- “Getting Started with YOLO v4” on page 17-44
- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 17-49
- “Getting Started with Mask R-CNN for Instance Segmentation” on page 17-55
- “Getting Started with Semantic Segmentation Using Deep Learning” on page 17-62
- “Point Feature Types” on page 17-65
- “Local Feature Detection and Extraction” on page 17-71
- “Get Started with Cascade Object Detector” on page 17-89
- “Train Optical Character Recognition for Custom Fonts” on page 17-102
- “Troubleshoot ocr Function Results” on page 17-106
- “Create a Custom Feature Extractor” on page 17-107
- “Image Retrieval with Bag of Visual Words” on page 17-110
- “Image Classification with Bag of Visual Words” on page 17-113

Getting Started with Anomaly Detection Using Deep Learning

Anomaly detection using deep learning is an increasingly popular approach to automating visual inspection tasks. The goal of anomaly detection is to perform a binary classification of images as normal or anomalous. For example, in the manufacturing industry, anomaly detection can determine which manufactured units meet quality standards and which units are defective.

You can train an anomaly detector using semi-supervised training. A complete semi-supervised workflow consists of training a model on normal image data and determining an anomaly threshold that separates normal images from anomalous images. Anomaly detection techniques generally follow these steps:

1. Prepare training data and calibration data.
2. Train a model to produce anomaly scores.
3. Calibrate the model by determining an anomaly score threshold.
4. Classify test images using the trained model and the chosen anomaly threshold.
5. Deploy the model onto a production line.

Note An alternative to anomaly detection is anomaly classification. Classification can differentiate between normal images and multiple known anomalous patterns. Classification can provide more insight on the type of anomaly, but it requires a greater amount of training data of the anomaly classes. For an example, see “Classify Defects on Wafer Maps Using Deep Learning” on page 3-24.

Prepare Training and Calibration Data

You can train an anomaly detector using one-class learning, which means that the network trains on data consisting only of normal images. Despite training on samples only of normal images, the model learns how to distinguish between normal and anomalous images. One-class learning offers many advantages for anomaly detection problems:

- Representations of anomalies can be scarce.
- Anomalies can represent expensive or catastrophic outcomes.
- There can be many types of anomalies, and the types of anomalies can change over the lifetime of the model. Describing what a normal image looks like is often more feasible than providing data that represents all possible anomalies in real world settings.

Some techniques, such as Outlier Exposure, add a small number of anomalous images to the training data set.

In semi-supervised learning, you can tune the performance of the trained model using calibration data. The calibration data set consists of labeled samples of normal and anomalous images. Avoid overfitting the model by using different images in the training and calibration data sets. You can use the **Image Labeler** app to interactively label images with scene labels. For more information, see “Get Started with the Image Labeler” on page 9-30.

Train Model

Train a model that accepts images as input and returns scalar valued anomaly scores. The score represents the likelihood that an image is anomalous. Larger scores indicate a higher probability that an image in anomalous samples.

Optionally, you can design a network that returns localized anomaly scores in addition to a scalar aggregate score for the image. Localized score information enables you to subjectively understand the network classification decisions.

Calibrate Model

Calibrate the model by determining an anomaly score threshold that separates the scores of good samples from the scores of anomaly samples. Unfortunately, most data sets cannot be cleanly separated into these two categories, and will have misclassifications in the form of false positives (normal images classified as anomalous) and false negatives (anomalous images classified as normal). By adjusting the threshold, you can tune the false positive and false negative rates to satisfy your operating requirements.

To select a threshold, you can use statistical techniques such as receiver operating characteristic (ROC) curves. A ROC curve plots the true positive rate versus false positive rate for different thresholds. If you have Statistics and Machine Learning Toolbox™, then you can use the `rocmetrics` function to calculate metrics such as the area under the ROC curve (AUC) and accuracy.

If your model also returns localized information about anomaly scores, then you can visualize the anomaly scores to understand the classification results. For example, you can display the anomaly score map as a heatmap overlayed on a calibration image. This approach is useful for identifying patterns in false negatives and false positives. You can use these patterns to identify strategies to improve the classifier performance.

Perform Classification Using Model

Classify test images as normal or anomalous by passing the images to the trained model, then comparing the score returned by the trained model against the anomaly score threshold.

Deploy Model

The end goal of visual inspection is often to deploy trained models for use in making business decisions in real-time production systems on assembly lines. The model determines acceptance or rejection given an image. The MATLAB Coder and GPU Coder™ products are effective tools for deploying visual inspection systems to CPU and NVIDIA® GPU targets.

See Also

Related Examples

- “Detect Image Anomalies Using Pretrained ResNet-18 Feature Embeddings” on page 3-59
- “Detect Image Anomalies Using Explainable One-Class Classification Neural Network” on page 3-40

Getting Started with Video Classification Using Deep Learning

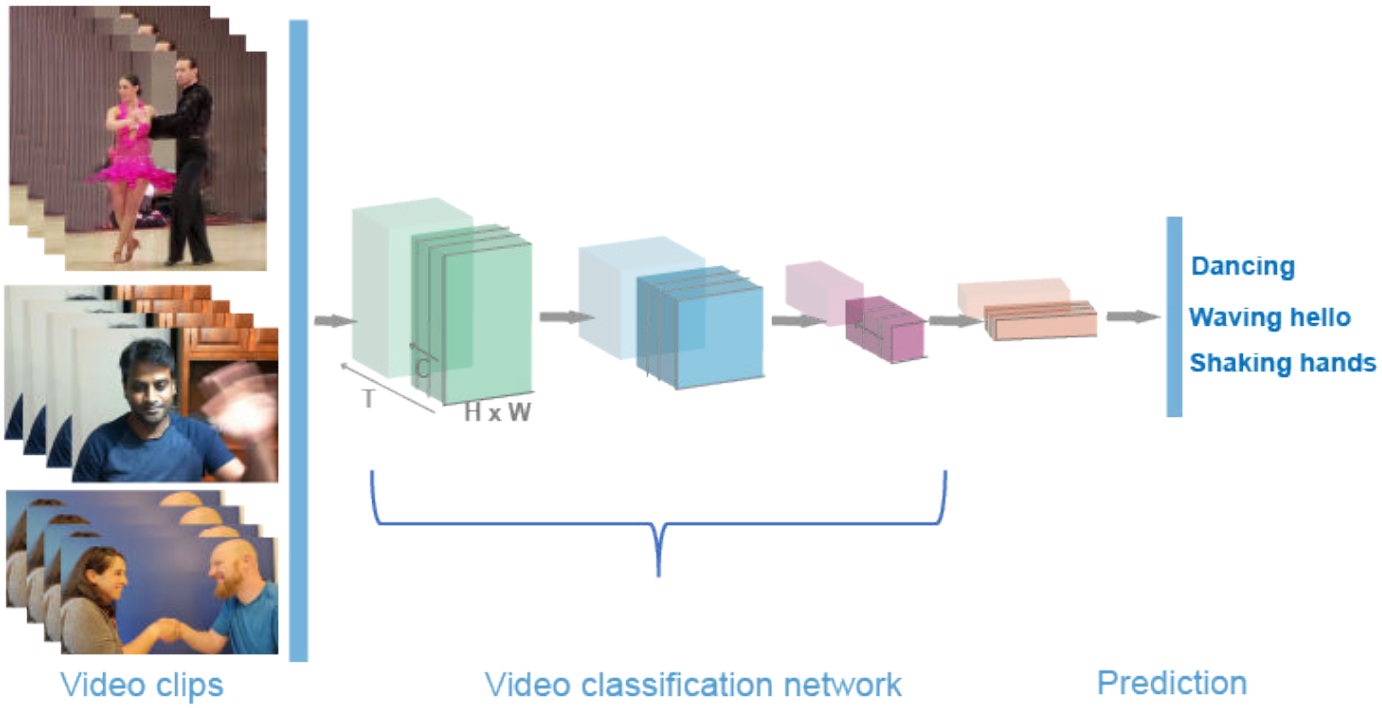
Video classification is similar to image classification, in that the algorithm uses feature extractors, such as convolutional neural networks (CNNs), to extract feature descriptors from a sequence of images and then classify them into categories. Video classification using deep learning provides a means to analyze, classify, and track activity contained in visual data sources, such as a video stream. Video classification has many applications, such as human activity recognition, gesture recognition, anomaly detection, and surveillance.

Video classification methodology includes these steps:

1. Prepare training data
2. Choose a video classifier
3. Train and evaluate the classifier
4. Use the classifier to process video data

You can train a classifier using a video classifier pretrained on a large activity recognition video data set, such as the Kinetics-400 Human Action Dataset, which is a large-scale and high-quality data set collection. Start by providing the video classifier with labeled video or video clips. Then, using a deep learning video classifier that consists of convolution neural networks that match the nature of the video input, you can predict and classify the videos. Ideally, your workflow should include the evaluation of your classifier. Finally, you can use the classifier to classify activity in a collection of videos or a streaming video from a webcam.

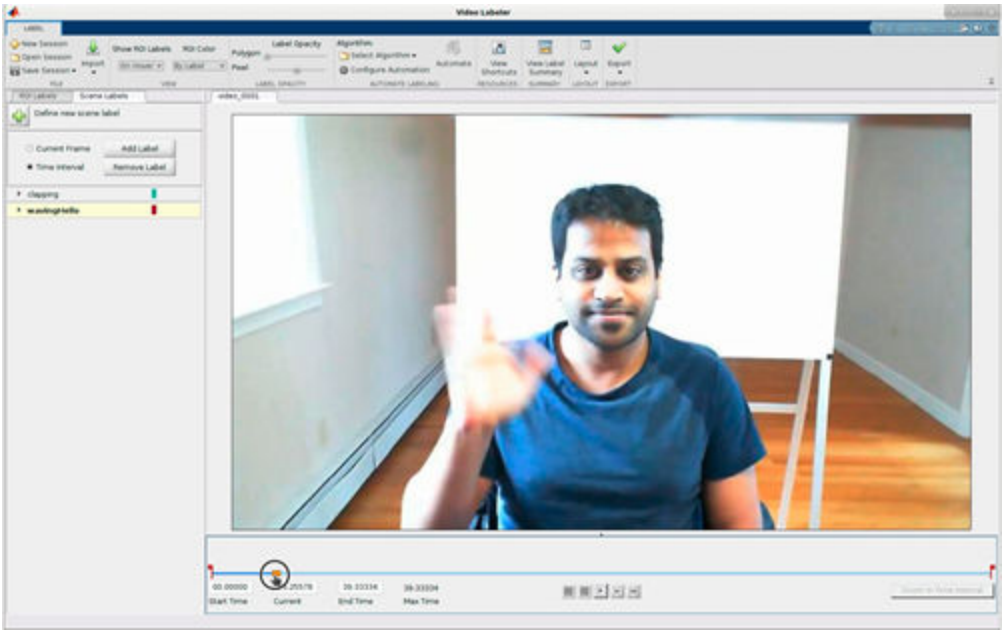
Computer Vision Toolbox provides the slow and fast pathway (SlowFast), ResNet with (2+1)D convolutions, and two-stream Inflated-3D techniques for training a classifier of video classification.



Create Training Data for Video Classification

To train a classifier network, you need a collection of videos and its corresponding collection of scene labels. A scene label is a label applied to a time range in a video. For example, you could label a range of frames `"jumping"`.

You can use the **Video Labeler** or **Ground Truth Labeler** to interactively label ground truth data in a video, image sequence, or custom data source with scene labels. For a summary all labelers, see “Choose an App to Label Ground Truth Data” on page 9-40.



The labeler apps export labeled data into MAT files that contain `groundTruth` objects. For an example showing how to extract training data from ground truth objects, see “Extract Training Data for Video Classification” on page 3-140.

Augment and Preprocess Data

Data augmentation provides a way to use limited data sets for training. Minor changes, such as translating, cropping, or transforming an image, provide new, distinct, and unique images that you can use to train a robust video classifier. Datastores are a convenient way to read and augment collections of data. Use the `fileDatastore` function with a `read` function that uses the `VideoReader` to read video files, to create datastores for videos and labeled scene label data. For an example that augments and preprocesses data, see “Gesture Recognition using Videos and Deep Learning” on page 3-147.

To learn how to augment and preprocess data, see “Perform Additional Image Processing Operations Using Built-In Datastores” (Deep Learning Toolbox) and “Datastores for Deep Learning” (Deep Learning Toolbox).

Create Video Classifier

Choose one of the listed video classifier objects to create deep learning classification networks using models pretrained models using the Kinetics-400 data set (which contains 400 class labels):

- The `slowFastVideoClassifier` model is pretrained on the Kinetics-400 data set which contains the residual network ResNet-50 model as the backbone architecture with slow and fast pathways. This functionality requires the Computer Vision Toolbox Model for SlowFast Video Classification.
- The `r2plus1dVideoClassifier` model is pretrained on the Kinetics-400 data set which contains 18 spatio-temporal (ST) residual layers. This functionality requires the Computer Vision Toolbox Model for R(2+1)D Video Classification.
- The `inflated3dVideoClassifier` model contains two subnetworks: the video network and the optical flow network. These networks are trained on the Kinetics-400 data set with RGB data and optical flow data, respectively. This functionality requires the Computer Vision Toolbox Model for Inflated-3D Video Classification.

The table provides a comparison of the these deep learning supported classifiers:

Model	Data Sources	Classifier Model Size (Pretained on Kinetics-400 Dataset)	GPU Support	Multiple Class Support	Description
SlowFast	Video data	124 MB	Yes	Yes	• Faster convergence during training than the Inflated-3D video classifier. Transfer learning on your data set can be slower than the R(2+1)D video classifier because of the two pathways in the 3-D convolutional neural network. • The 3-D convolutional neural network is deeper than those of the Inflated-3D and R(2+1)D video classifiers. • Does not require optical flow data, in addition to video data • You must use a low `MiniBatchSize` value

Model	Data Sources	Classifier Model Size (Pretained on Kinetics-400 Dataset)	GPU Support	Multiple Class Support	Description
					per GPU because of the depth of the residual layers. The value must be much lower than for a corresponding R(2+1)D classifier because of the two pathways (roughly half the value you would use for R(2+1)D). • Choose this classifier to obtain good classification accuracy results for your data set, and for faster convergence during transfer learning at the expense of greater GPU memory requirements.

Model	Data Sources	Classifier Model Size (Pretained on Kinetics-400 Dataset)	GPU Support	Multiple Class Support	Description
R(2+1)D	Video data	112 MB	Yes	Yes	• Faster convergence during training than the Inflated-3D video classifier. • The 3-D convolutional neural network is deeper than the Inflated-3D CNN. • Does not require optical flow data or RGB data. • Choose this classifier to obtain good classification accuracy results for your data set, and for faster convergence during transfer learning at the expense of greater GPU memory requirements. • You must reduce the `MiniBatchSize` per GPU,

Model	Data Sources	Classifier Model Size (Pretained on Kinetics-400 Dataset)	GPU Support	Multiple Class Support	Description
					because of the depth of the residual layers.

Model	Data Sources	Classifier Model Size (Pretained on Kinetics-400 Dataset)	GPU Support	Multiple Class Support	Description
Inflated-3D	• Optical flow data • Video data	91 MB	Yes	Yes	• Accuracy of the classifier improves when combining optical flow and RGB data. • Slower convergence during training compared to R(2+1)D and SlowFast video classifiers. • Use with optical flow data to capture motion information, as the accuracy of the classifier improves with optical flow data and video data. • Commonly used as a baseline when comparing video classifiers. Choose this classifier to obtain baseline

Model	Data Sources	Classifier Model Size (Pretained on Kinetics-400 Dataset)	GPU Support	Multiple Class Support	Description
					results for your data set through transfer learning, and to train while using less GPU memory. • You can set `MiniBatchSize` to a value greater than for either R(2+1)D or SlowFast.

This table shows sample code you can use to create a video classifier using each of the listed video classifiers:

Video Classifier	Sample Creation Code

SlowFast	`inputSize = [112 112 64 3];`
	`classes = ["wavingHello","clapping"];`
	`sf = slowFastVideoClassifier("resnet50-3d",classes,In`

R(2+1)D	`inputSize = [112 112 64 3];`
	`classes = ["wavingHello","clapping"];`
	`rd = r2plus1dVideoClassifier("resnet-3d-18",classes,I`

Inflated 3-D	`inputSize = [112 112 64 3];`
	`classes = ["wavingHello","clapping"];`
	`i3d = inflated3dVideoClassifier("googlenet-video-flow`

Train Video Classifier and Evaluate Results

To learn how to train and evaluate the results for the listed video classifiers, see these examples:

- “Gesture Recognition using Videos and Deep Learning” on page 3-147 — Train and evaluate a SlowFast video classifier
- “Activity Recognition Using R(2+1)D Video Classification” on page 3-86 — Train and evaluate an R(2+1)D video classifier
- “Activity Recognition from Video and Optical Flow Data Using Deep Learning” on page 3-108 — Train and evaluate a two-stream Inflated-3D video classifier

Classify Using Deep Learning Video Classifiers

To learn how to classify videos using a video classifier, see these examples:

- “Evaluate a Video Classifier” on page 3-136
- “Classify Streaming Webcam Video Using SlowFast Video Classifier” on page 3-144

See Also

Apps
Video Labeler | Ground Truth Labeler

More About

- “Classify Videos Using Deep Learning” (Deep Learning Toolbox)
- “Get Started with the Video Labeler” on page 9-44

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 17-24.

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 3-364	• 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. 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. "Comparison of R-CNN Object Detectors" on page 17-51
`fastRCNNObjectDetector`	Yes	Yes	No	Yes	Train Fast R-CNN Stop Sign Detector on page 3-415	• 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. "Comparison of R-CNN Object Detectors" on page 17-51

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 3-351	• Consider starting with the `fasterRCNNObjectDetector`. • Requires GPU for optimal performance. • Slowest of the R-CNN-based detectors. 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. "Comparison of R-CNN Object Detectors" on page 17-51
`yolov2ObjectDetector`	Yes	Yes	Yes	Yes	"Object Detection Using YOLO v2 Deep Learning" on page 3-338	• 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. YOLO v2 uses a single stage network to perform object detection.

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 3-201	• 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. Single shot detector (SSD) uses a single stage detection network to detects objects using multi-scale features.

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 3-412	• 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. 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. ACF would not work well for detecting vehicles from various viewpoints, such as front, side, and rear.
`peopleDetectorACF`	Pretrained	No	Yes	No	"Tracking Pedestrians from a Moving Car" on page 8-40	Use this pretrained detector to detect upright positioned people.
`vision.PeopleDetector`	Pretrained	No	Yes	No	"Depth Estimation From Stereo Video" on page 1-131	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"	• 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 17-55	Use this detector when you need to segment individual objects.
`yolov3ObjectDetector`	Yes	Yes	Yes	Yes	"Object Detection Using YOLO v3 Deep Learning" on page 3-323	YOLO v3 is a single stage network that uses multi-scale features to better handle detection of objects of various sizes.
`yolov4ObjectDetector`	Yes	Yes	Yes	Yes	"Object Detection Using YOLO v4 Deep Learning" on page 3-423	YOLO v4 is a single stage object detector that is fast and accurate than YOLO v3. The detector uses spatial pyramid pooling and path aggregation network for computing aggregated features and is capable of detecting small objects of different sizes.
`vehicleDetectorACF`	Pretrained	No	Yes	No	"Track Multiple Vehicles Using a Camera" (Automated Driving Toolbox)	Pretrained ACF detector

Detector	Multiple Classes Support	Deep Learning Support	Code Generation Support	GPU Support	Example	Description
`vehicleDetectorFasterRCNN`	Pretrained	Yes	No	Yes	"Train a Deep Learning Vehicle Detector" (Automated Driving Toolbox)	Pretrained Faster R-CNN detector
`vehicleDetectorYOLOv2`	Pretrained	Yes	Yes	Yes	"Detect Vehicles Using Monocular Camera and YOLO v2" (Automated Driving Toolbox)	Pretrained YOLO v2 detector

See Also

Apps
Image Labeler | **Video Labeler** | **Ground Truth Labeler**

Objects
`ssdObjectDetector` | `acfObjectDetector` | `rcnnObjectDetector` | `vision.CascadeObjectDetector` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `yolov2ObjectDetector` | `vehicleDetectorYOLOv2`

Functions
`trainACFObjectDetector` | `trainRCNNObjectDetector` | `trainCascadeObjectDetector` | `trainSSDObjectDetector` | `trainFasterRCNNObjectDetector` | `trainYOLOv2ObjectDetector` | `trainFastRCNNObjectDetector` | `vehicleDetectorACF` | `vehicleDetectorFasterRCNN` | `acfObjectDetectorMonoCamera`

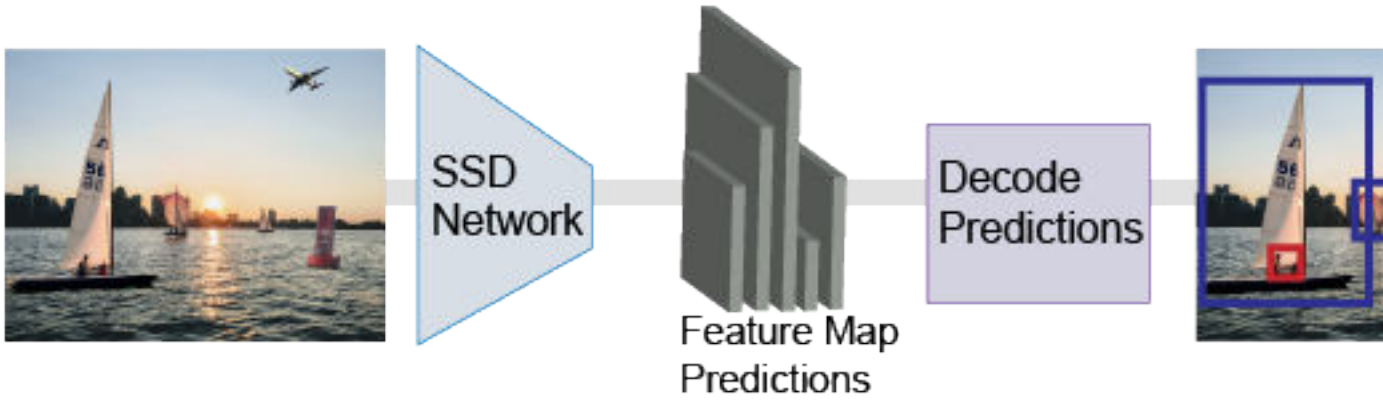
More About

- "Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN" on page 17-49
- "Getting Started with SSD Multibox Detection" on page 17-21
- "Getting Started with Object Detection Using Deep Learning" on page 17-24
- "Getting Started with YOLO v2" on page 17-37
- "Getting Started with Mask R-CNN for Instance Segmentation" on page 17-55

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.

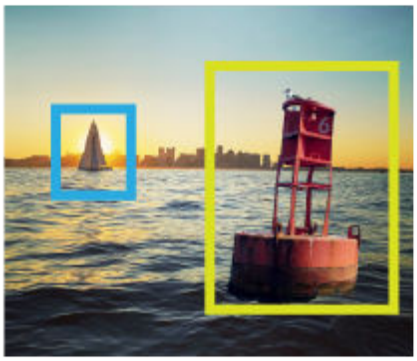


Predict Objects in the Image

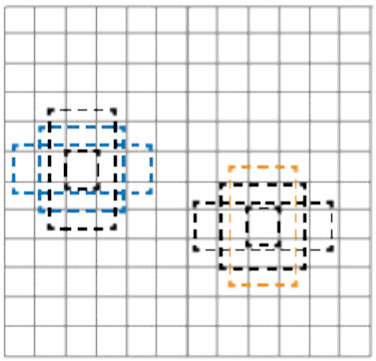
SSD uses anchor boxes to detect classes of objects in an image. For more details, see “Anchor Boxes for Object Detection” on page 17-32. The SSD predicts these two attributes for each anchor box.

- Anchor box offsets — Refine the anchor box position.
- Class probability — Predict the class label assigned to each anchor box.

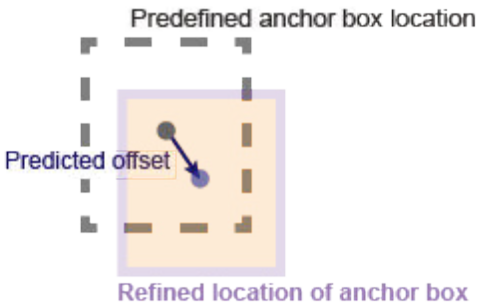
This figure shows predefined anchor boxes (the dotted lines) at each location in a feature map and the refined location after offsets are applied. Matched boxes with a class are in blue and orange.



Ground truth image and bounding boxes

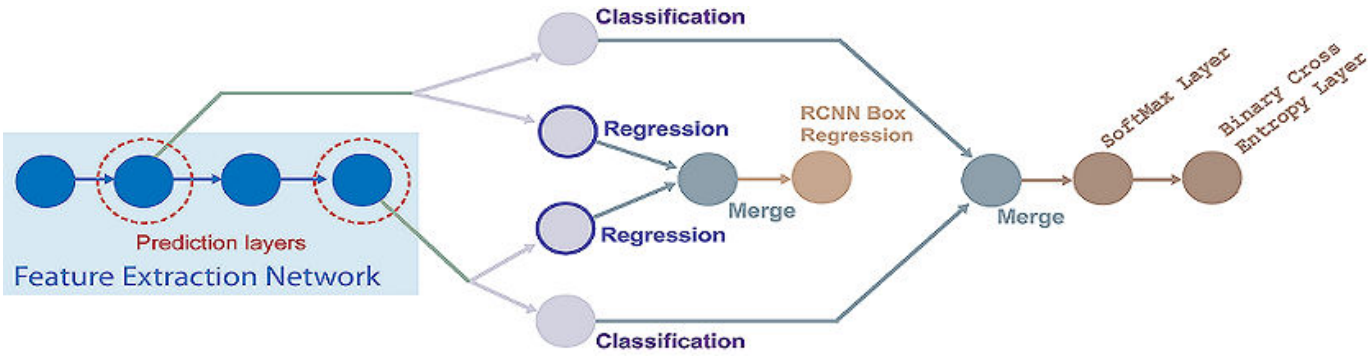


Anchor boxes at each predefined location in each feature map



Design an SSD Detection Network

You can design a custom SSD model programmatically by using the `ssdObjectDetector` function.



To design an SSD multibox detection network, follow these steps.

1. Start the model with a feature extractor network, which can be a pretrained or untrained CNN.
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 the names of the prediction layers as detection network source input to the `ssdObjectDetector` object. Also, specify the names of the classes and the anchor boxes as inputs to configure the detector for training.
4. The `ssdObjectDetector` object connects the outputs of the prediction layers to a classification branch and to a regression branch. The classification branch predicts the class for each tiled anchor box. The regression branch predicts anchor box offsets.

   The `ssdObjectDetector` object combines the outputs of the classification branches from all the prediction layers by using a merge layer. Then, the output from the merge layer of the classification branch is connected to a softmax layer followed by a binary cross-entropy layer. The classification branch computes classification loss by using softmax and binary cross-entropy loss function. The binary cross-entropy function uses the hard negative mining strategy.

   Similarly, the `ssdObjectDetector` object combines the outputs of the regression branches from all the prediction layers by using a merge layer. Then, the output from the merge layer of the regression branch is connected to a bounding box regression layer. The regression branch computes the bounding box loss by using smooth L1 function.
5. Train the SSD object detection network configured by the `ssdObjectDetector` object to detect objects in an image. You can use the trained SSD detector for multiclass object detection.

Train an Object Detector and Detect Objects with an SSD Model

To train a SSD object detection network, use the `trainSSDObjectDetector` function. For more information, see “Train SSD Object Detector”. To learn how to perform object detection by using the SSD deep learning technique, see the “Object Detection Using SSD Deep Learning” on page 3-201 example.

Transfer Learning

With transfer learning, you can use a pretrained CNN as the feature extractor in an SSD detection network. Use the `ssdObjectDetector` function to create an SSD detection network from a pretrained CNN, such as `MobileNetv2`. For a list of pretrained CNNs, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

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** 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. To create training data from any of the labelers exported ground truth object, you can use the `objectDetectorTrainingData` or `pixelLabelTrainingData` functions. For more details, see “Training Data for Object Detection and Semantic Segmentation” on page 9-84.

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
Image Labeler | Ground Truth Labeler | Video Labeler | Deep Network Designer

Objects
`ssdObjectDetector`

Functions
`trainSSDObjectDetector` | `analyzeNetwork`

Related Examples

- “Object Detection Using SSD Deep Learning” on page 3-201

More About

- “Anchor Boxes for Object Detection” on page 17-32
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

Getting Started with Object Detection Using Deep Learning

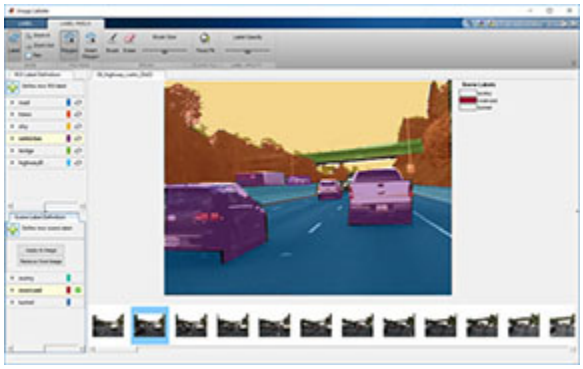
Object detection using deep learning provides a fast and accurate means to predict the location of an object in an image. Deep learning is a powerful machine learning technique in which the object detector automatically learns image features required for detection tasks. Several techniques for object detection using deep learning are available such as Faster R-CNN, you only look once (YOLO) v2, YOLO v3, YOLO v4, 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 9-40
- “Training Data for Object Detection and Semantic Segmentation” on page 9-84

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 17-38
- “Design an R-CNN, Fast R-CNN, and a Faster R-CNN Model” on page 17-52

Train Detector and Evaluate Results

Use the `trainFasterRCNNObjectDetector`, `trainYOLOv2ObjectDetector`, `trainYOLOv4ObjectDetector`, and `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 YOLO v4 Network for Vehicle Detection”
- “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`, `yolov3ObjectDetector`, `yolov4ObjectDetector`, 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)
```

- “Multiclass Object Detection Using YOLO v2 Deep Learning” on page 3-3
- “Object Detection Using YOLO v3 Deep Learning” on page 3-323
- “Object Detection Using YOLO v4 Deep Learning” on page 3-423
- “Object Detection Using SSD Deep Learning” on page 3-201

- “Object Detection Using Faster R-CNN Deep Learning” on page 3-364

Detect Objects Using Pretrained Object Detection Models

MathWorks® GitHub repository provides implementations of the latest pretrained object detection deep learning networks to download and use for performing out-of-the-box inference. The pretrained object detection networks are already trained on standard data sets such as the COCO and Pascal VOC data sets. You can use these pretrained models directly to detect different objects in a test image.

For a list of all the latest MathWorks pretrained object detectors, see MATLAB Deep Learning (GitHub).

See Also

Apps
Image Labeler | Video Labeler

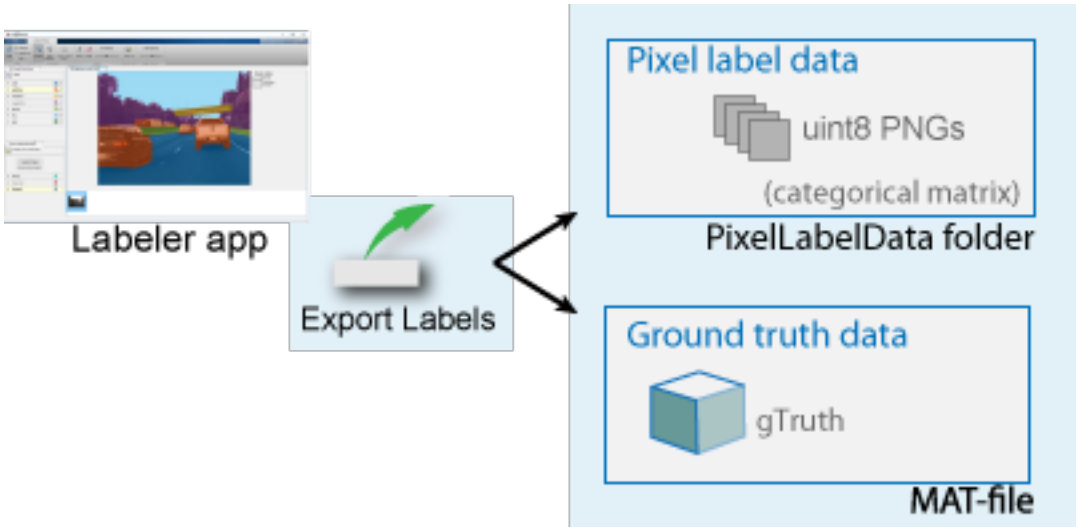
More About

- “Getting Started with YOLO v2” on page 17-37
- “Getting Started with YOLO v3” on page 17-41
- “Getting Started with YOLO v4” on page 17-44
- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 17-49
- “Getting Started with SSD Multibox Detection” on page 17-21
- “Getting Started with Mask R-CNN for Instance Segmentation” on page 17-55
- “Anchor Boxes for Object Detection” on page 17-32

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 17-28.

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 17-29.

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`.

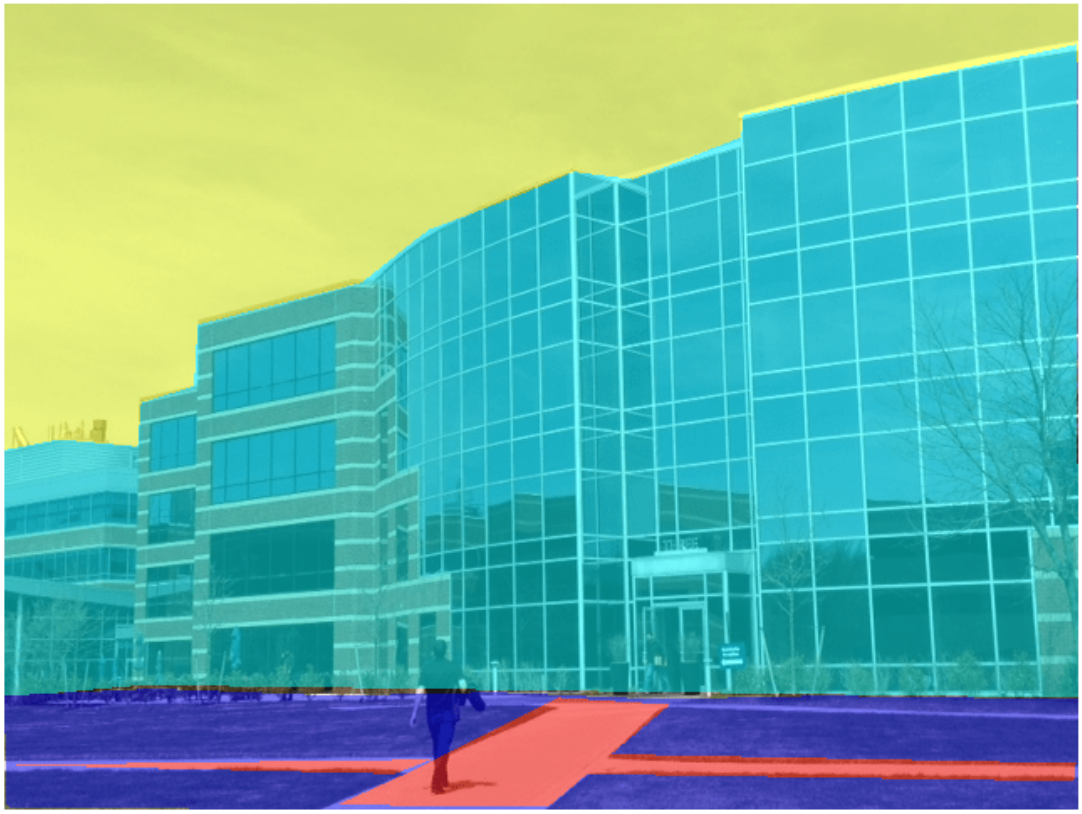
```
labelIDs = [1,2,3,4];
labelcats = ["sky" "grass" "building" "sidewalk"];
```

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 I(*i*,*j*).

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

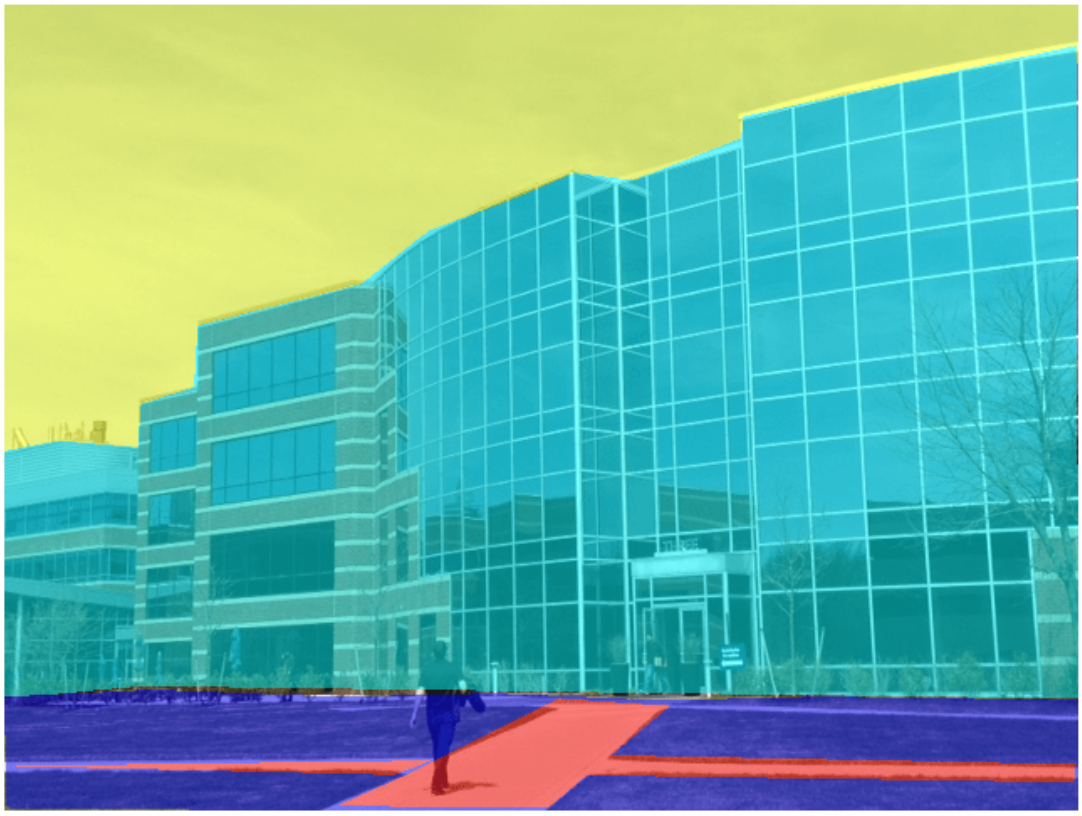
Display the label categories in `C`.

```
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
Image Labeler | **Video Labeler** | **Ground Truth Labeler**

Objects
`groundTruth` | `groundTruthMultisignal` | `pixelLabelImageDatastore`

Functions
`changeFilePaths (groundTruthMultisignal)` | `changeFilePaths (groundTruth)`

More About

- “Label Pixels for Semantic Segmentation” on page 9-14
- “Share and Store Labeled Ground Truth Data” on page 9-67

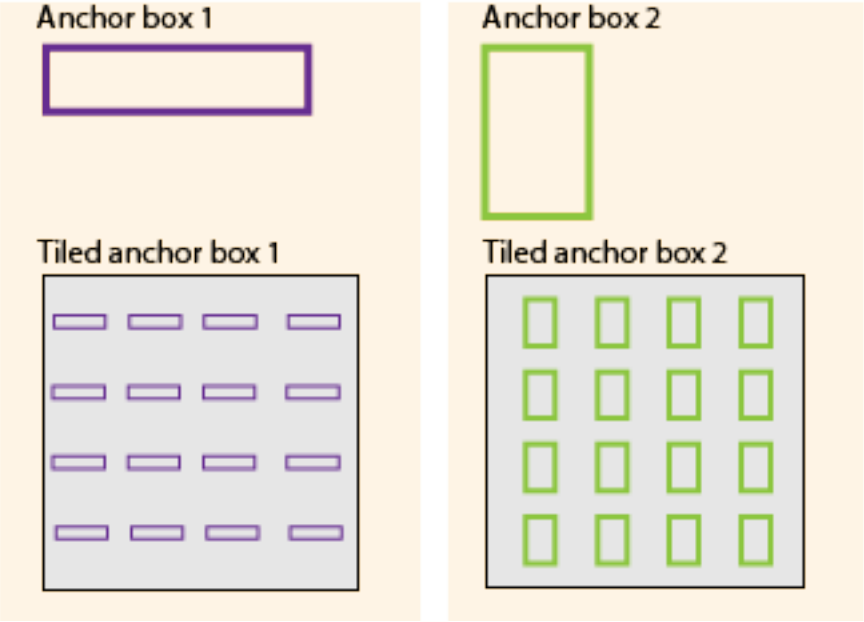
Anchor Boxes for Object Detection

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 manner, 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?

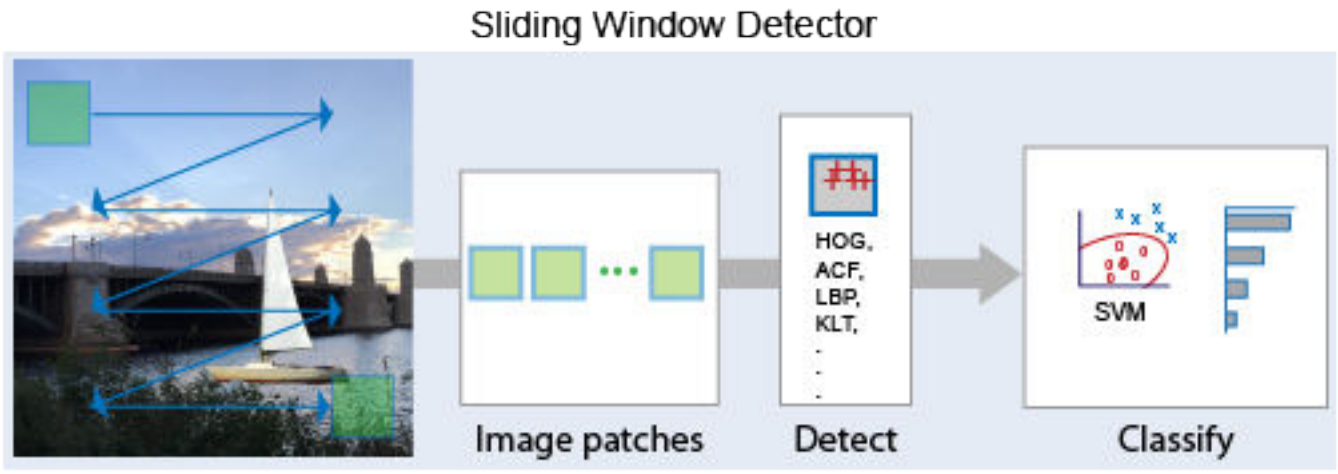
Anchor boxes are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect and are typically chosen based on object sizes in your training datasets. During detection, the predefined anchor 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

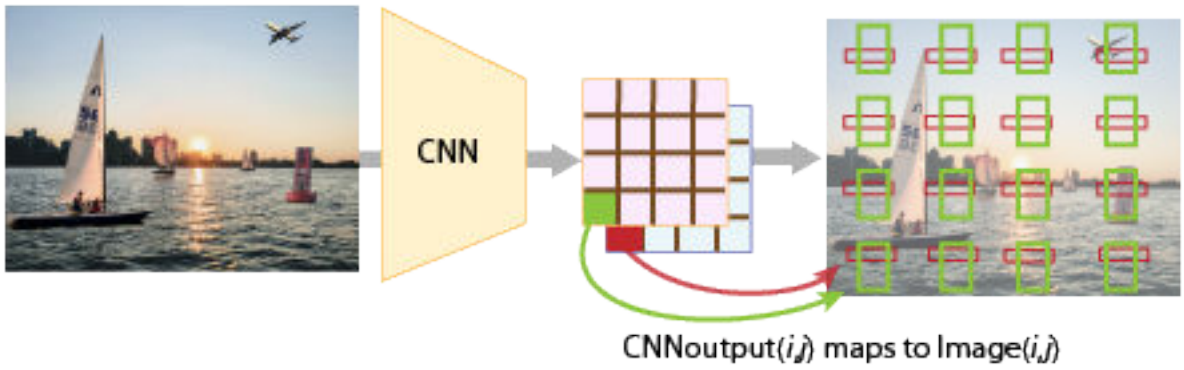
When using anchor boxes, you can evaluate all object predictions at once. Anchor boxes eliminate the need to scan an image with a sliding window that computes a separate prediction at every potential position. Examples of detectors that use a sliding window are those that are based on aggregate channel 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.



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?

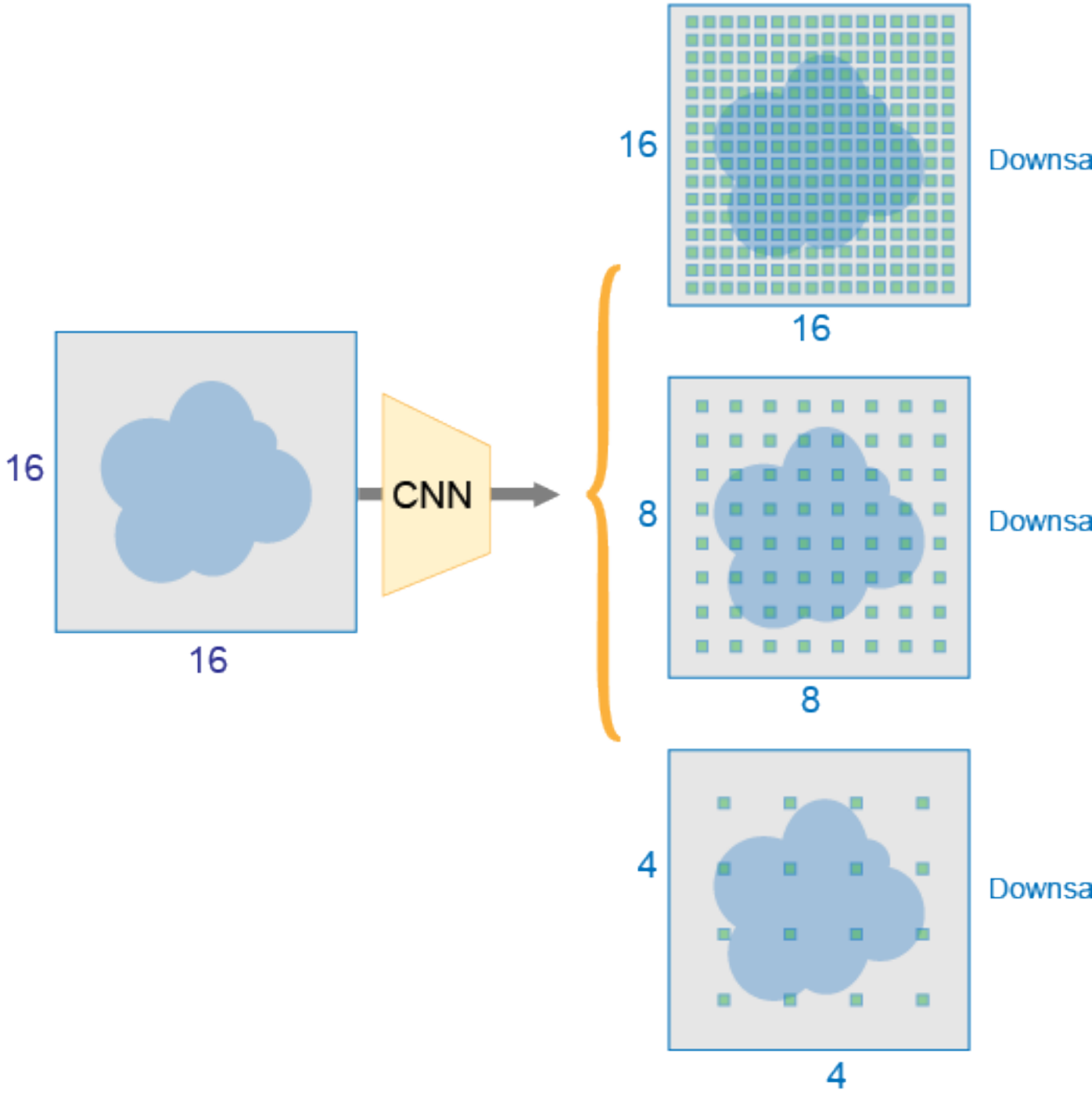
The position of an anchor box is determined by mapping the location of the network output back to the input image. The process is replicated for every network output. The result produces a set of tiled anchor boxes across the entire image. Each anchor box represents a specific prediction of a class. For example, there are two anchor boxes to make two predictions per location in the image below.



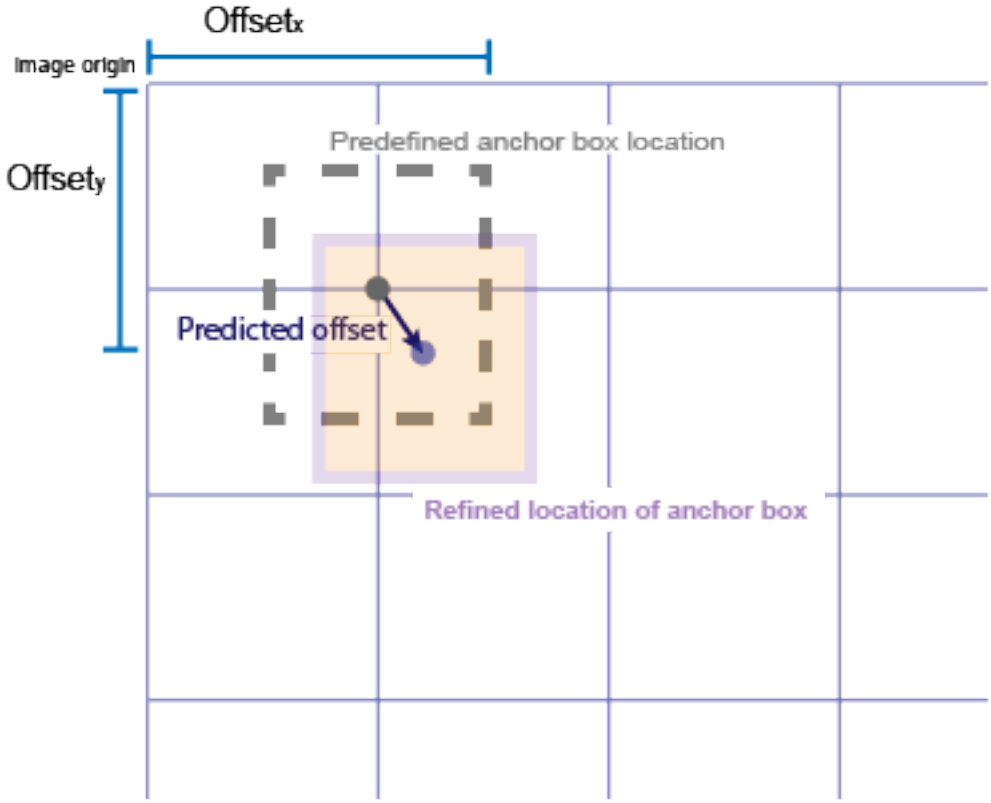
Each anchor box is tiled across the image. The number of network outputs equals the number of tiled anchor boxes. The network produces predictions for all outputs.

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.



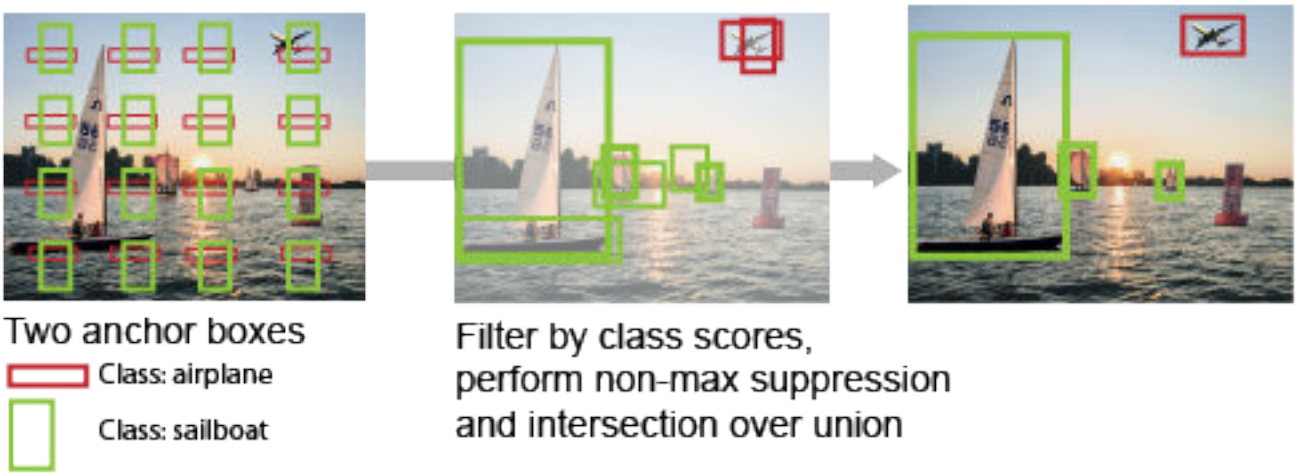
To fix localization errors, deep learning object detectors learn offsets to apply to each tiled anchor box refining the anchor box position and size.



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 3-319.

See Also

Related Examples

- “Create YOLO v2 Object Detection Network” on page 3-348
- “Train Object Detector Using R-CNN Deep Learning” on page 3-351
- “Object Detection Using Faster R-CNN Deep Learning” on page 3-364
- Estimate Anchor Boxes From Training Data on page 3-319

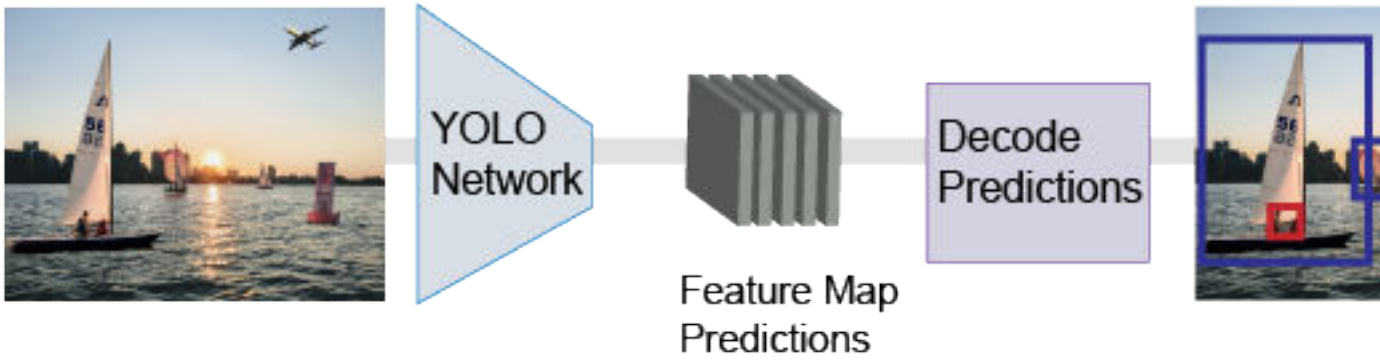
More About

- “Getting Started with YOLO v2” on page 17-37
- “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.

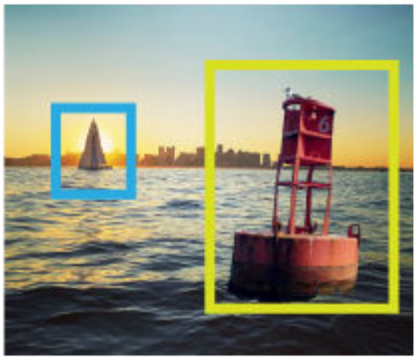


Predicting Objects in the Image

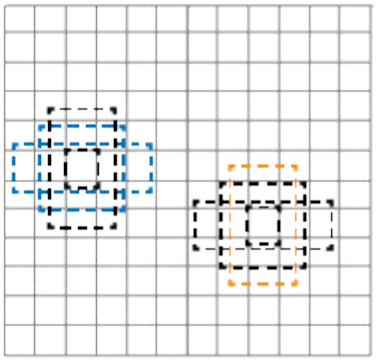
YOLO v2 uses anchor boxes to detect classes of objects in an image. For more details, see “Anchor Boxes for Object Detection” on page 17-32.The YOLO v2 predicts these three attributes for each anchor box:

- 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.

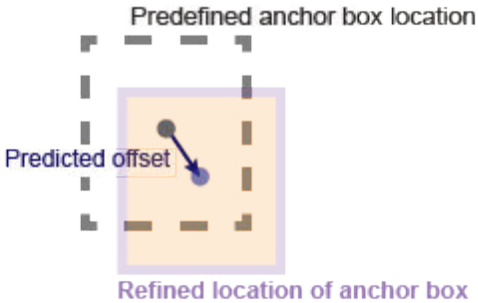
The figure shows predefined anchor boxes (the dotted lines) at each location in a feature map and the refined location after offsets are applied. Matched boxes with a class are in color.



Ground truth image and bounding boxes



Anchor boxes at each predefined location in each feature map



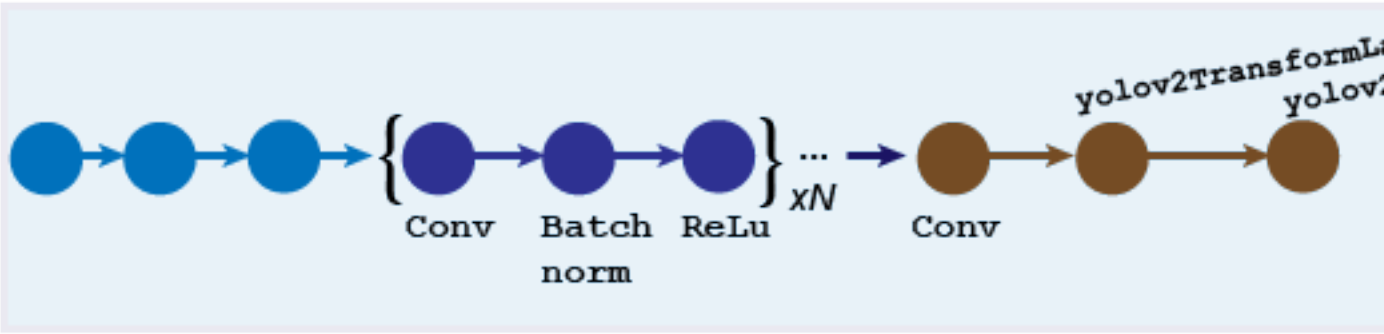
Transfer Learning

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 17-38.

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.



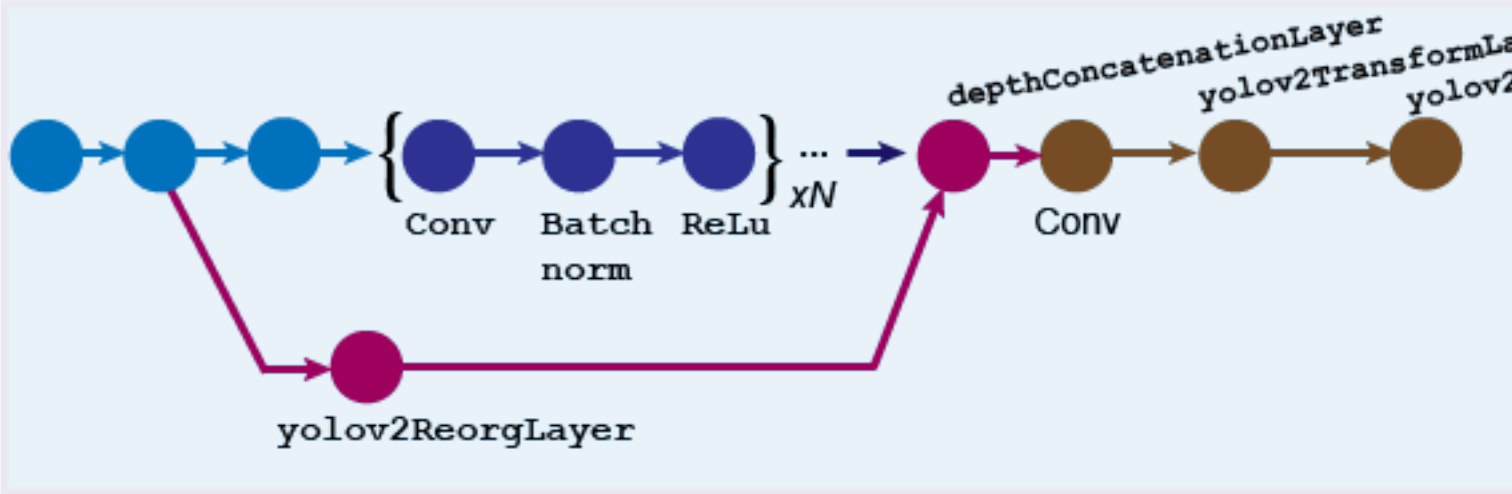
You can also use the **Deep Network Designer** app to manually create a network. The designer incorporates Computer Vision Toolbox YOLO v2 features.

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 3-348.

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 3-338 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** 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. To create training data from any of the labelers exported ground truth object, you can use the `objectDetectorTrainingData` or `pixelLabelTrainingData` functions. For more details, see “Training Data for Object Detection and Semantic Segmentation” on page 9-84.

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] Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali 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
Image Labeler | Ground Truth Labeler | Video Labeler | Deep Network Designer

Objects
`yolov2TransformLayer` | `yolov2OutputLayer` | `spaceToDepthLayer` | `yolov2ObjectDetector` | `depthConcatenationLayer`

Functions
`trainYOLOv2ObjectDetector` | `analyzeNetwork`

Related Examples

- “Train Object Detector Using R-CNN Deep Learning” on page 3-351
- “Object Detection Using YOLO v2 Deep Learning” on page 3-338
- “Code Generation for Object Detection by Using YOLO v2” on page 2-5

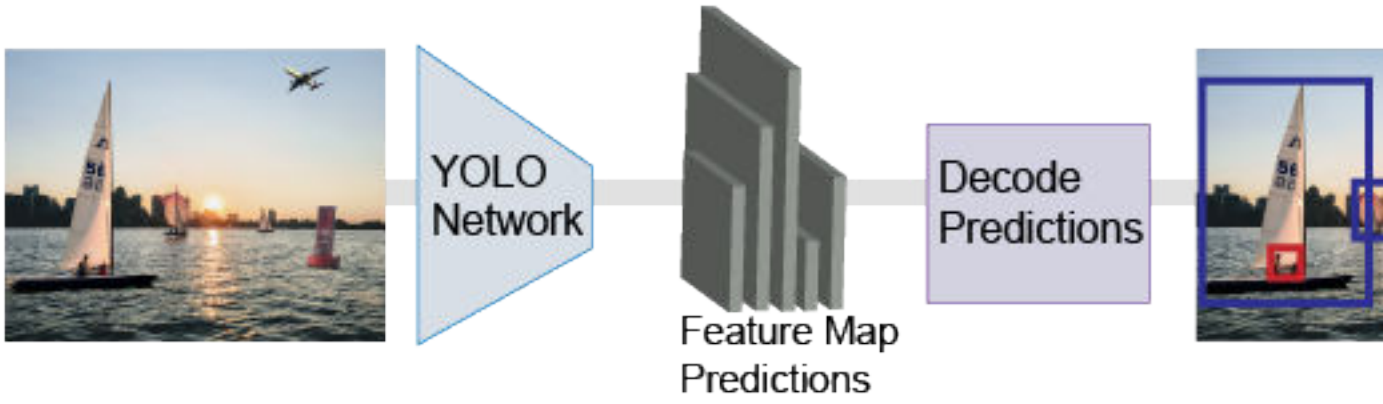
More About

- “Anchor Boxes for Object Detection” on page 17-32
- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” on page 17-49
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

Getting Started with YOLO v3

The you-only-look-once (YOLO) v3 object detector is a multi-scale object detection network that uses a feature extraction network and multiple detection heads to make predictions at multiple scales.

The YOLO v3 object detection model runs a deep learning convolutional neural network (CNN) on an input image to produce network predictions from multiple feature maps. The object detector gathers and decodes predictions to generate the bounding boxes.

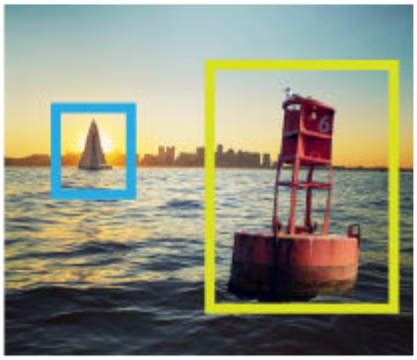


Predicting Objects in the Image

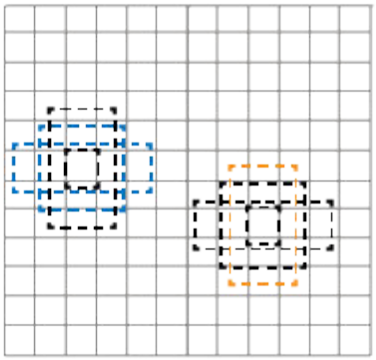
YOLO v3 uses anchor boxes to detect classes of objects in an image. For more details, see “Anchor Boxes for Object Detection” on page 17-32.The YOLO v3 predicts these three attributes for each anchor box:

- 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.

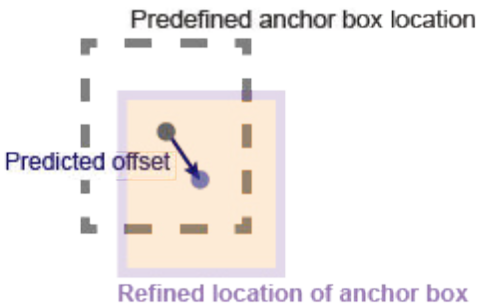
The figure shows predefined anchor boxes (the dotted lines) at each location in a feature map and the refined location after offsets are applied. Matched boxes with a class are in color.



Ground truth image and bounding boxes



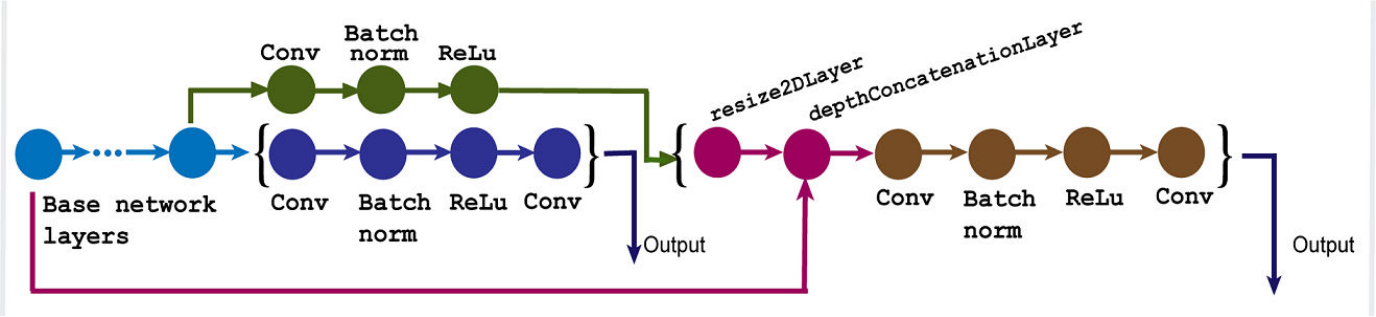
Anchor boxes at each predefined location in each feature map



Design a YOLO v3 Detection Network

To design a YOLO v3 object detection network, follow these steps.

1. Start the model with a feature extraction network. The feature extraction network serves as the base network for creating the YOLO v3 deep learning network. The base network can be a pretrained or untrained CNN. If the base network is a pretrained network, you can perform transfer learning.
2. Create detection subnetworks by using convolution, batch normalization, and ReLu layers. Add the detection subnetworks to any of the layers in the base network. The output layers that connect as inputs to the detection subnetworks are the detection network source. Any layer from the feature extraction network can be used as a detection network source. To use multiscale features for object detection, choose feature maps of different sizes.



To manually create a YOLO v3 deep learning network, use the **Deep Network Designer** app. To programmatically create a YOLO v3 deep learning network, use the `yolov3ObjectDetector` object.

Transfer Learning

To perform transfer learning, you can use a pretrained deep learning network as the base network for YOLO v3 deep learning network. Configure the YOLO v3 deep learning for training on a new dataset by specifying the anchor boxes and the new object classes. Use the `yolov3ObjectDetector` object to create a YOLO v3 detection network from any pretrained CNN, like `SqueezeNet` and perform transfer learning. For a list of pretrained CNNs, see “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

Train an Object Detector and Detect Objects with a YOLO v3 Model

To learn how to create a custom YOLO v3 object detector by using a deep learning network as base network and train for object detection, see the “Object Detection Using YOLO v3 Deep Learning” on page 3-323 example.

Label Training Data for Deep Learning

You can use the **Image Labeler**, **Video Labeler**, or **Ground Truth Labeler** 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. To create training data from any of the labelers exported ground truth object, you can use the `objectDetectorTrainingData` or `pixelLabelTrainingData` functions. For more details, see “Training Data for Object Detection and Semantic Segmentation” on page 9-84.

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] Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali 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
Image Labeler | Ground Truth Labeler | Video Labeler | Deep Network Designer

Objects
`yolov3ObjectDetector`

Functions
`detect` | `preprocess` | `forward` | `predict`

Related Examples

- “Object Detection Using YOLO v3 Deep Learning” on page 3-323

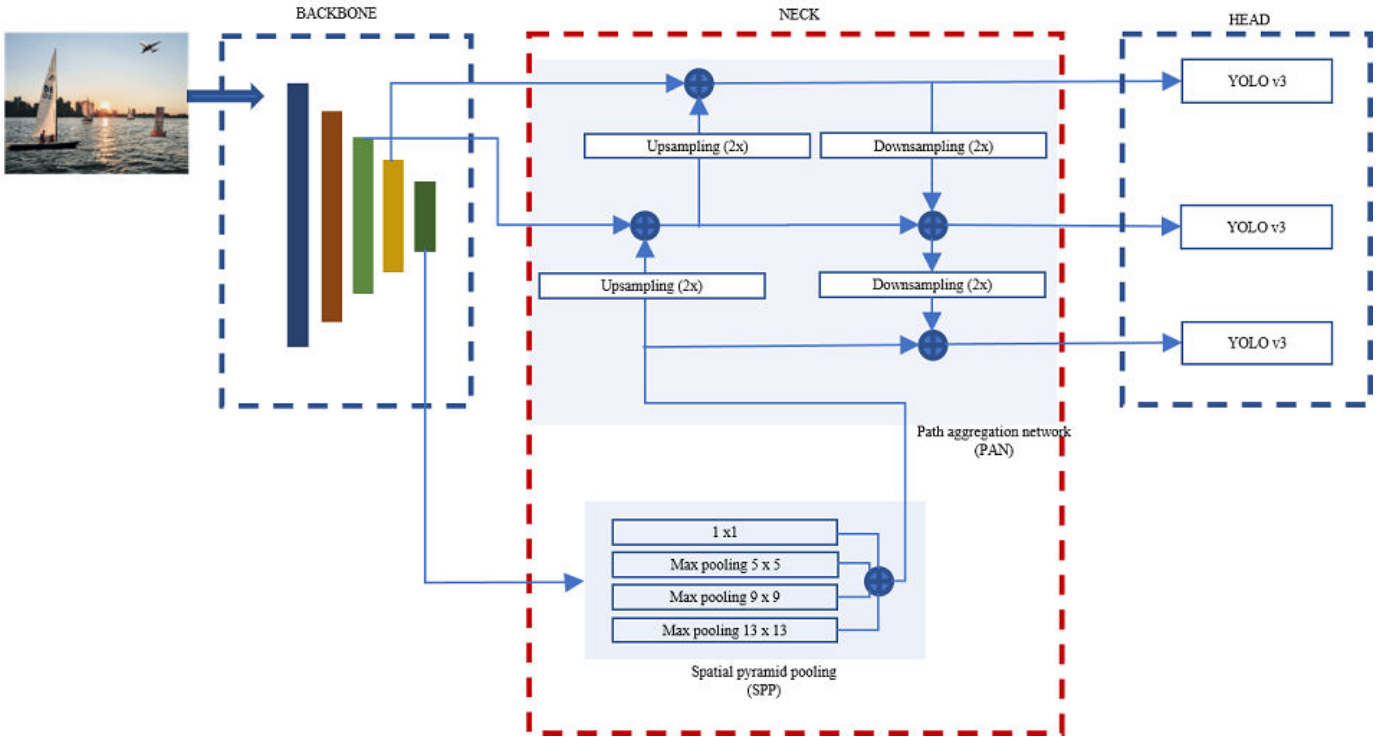
More About

- “Anchor Boxes for Object Detection” on page 17-32
- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)

Getting Started with YOLO v4

The you only look once version 4 (YOLO v4) object detection network is a one-stage object detection network and is composed of three parts: backbone, neck, and head.

- The backbone can be a pretrained convolutional neural network such as VGG16 or CSPDarkNet53 trained on COCO or ImageNet data sets. The backbone of the YOLO v4 network acts as the feature extraction network that computes feature maps from the input images.
- The neck connects the backbone and the head. It is composed of a spatial pyramid pooling (SPP) module and a path aggregation network (PAN). The neck concatenates the feature maps from different layers of the backbone network and sends them as inputs to the head.
- The head processes the aggregated features and predicts the bounding boxes, objectness scores, and classification scores. The YOLO v4 network uses one-stage object detectors, such as YOLO v3, as detection heads.



The YOLO v4 network uses CSPDarkNet-53 as the backbone for extracting features from the input images. The backbone has five residual block modules, and the feature map outputs from the residual block modules are fused at the neck of the YOLO v4 network.

The SPP module in the neck concatenates the max-pooling outputs of the low-resolution feature map to extract the most representative features. The SPP module uses kernels of size 1-by-1, 5-by-5, 9-by-9, and 13-by-13 for the max-pooling operation. The stride value is set to 1. Concatenating the feature maps increases the receptive field of backbone features and increases the accuracy of the network for detecting small objects. The concatenated feature maps from the SPP module are fused with the high-resolution feature maps by using a PAN. The PAN uses upsampling and downsampling operations to set bottom-up and top-down paths for combining the low-level and high-level features.

The PAN module outputs a set of aggregated feature maps to use for predictions. The YOLO v4 network has three detection heads. Each detection head is a YOLO v3 network that computes the final predictions. The YOLO v4 network outputs feature maps of sizes 19-by-19, 38-by-38, and 76-by-76 to predict the bounding boxes, classification scores, and objectness scores.

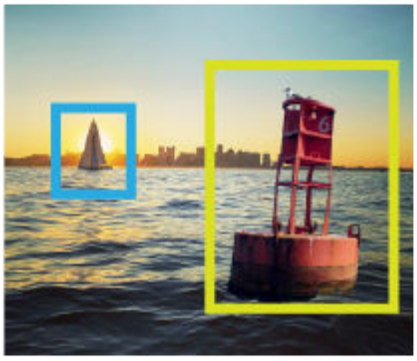
Tiny YOLO v4 network is a lightweight version of the YOLO v4 network with fewer network layers. The tiny YOLO v4 network uses a feature pyramid network as the neck and has two YOLO v3 detection heads. The network outputs feature maps of size 13-by-13 and 26-by-26 for computing predictions.

Predict Objects Using YOLO v4

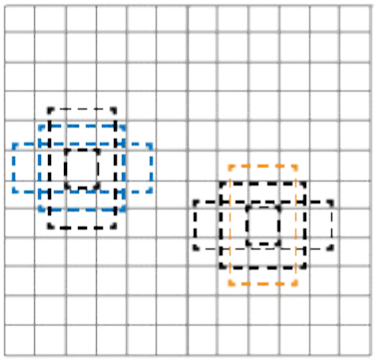
YOLO v4 uses anchor boxes to detect classes of objects in an image. For details about anchor boxes, see “Anchor Boxes for Object Detection” on page 17-32. Similar to YOLO v3, YOLO v4 predicts these three attributes for each anchor box:

- Intersection over union (IoU) — Predicts the objectness score of each anchor box.
- Anchor box offsets — Refines the anchor box position.
- Class probability — Predicts the class label assigned to each anchor box.

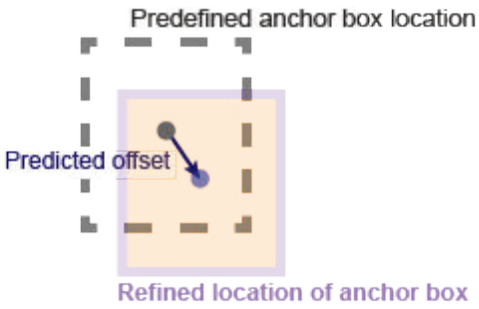
The figure shows predefined anchor boxes, represented by dotted lines, at each location in a feature map, and the refined location after applying the offsets. The anchor boxes that have been matched with a class are in color.



Ground truth image and bounding boxes



Anchor boxes at each predefined location in each feature map



You must specify the predefined anchor boxes, also known as *a priori* boxes, and the classes while training the network.

Create YOLO v4 Object Detection Network

To programmatically create a YOLO v4 deep learning network, use the `yolov4ObjectDetector` object. You can create a `yolov4ObjectDetector` object, to detect objects in an image, using the pretrained YOLO v4 deep learning networks `csp-darknet53-coco` and `tiny-yolov4-coco`. These networks are trained on the COCO data set. `csp-darknet53-coco` is a YOLO v4 network with three detection heads, and `tiny-yolov4-coco` is a tiny YOLO v4 network with two detection heads. To download these YOLO v4 pretrained networks, you must install the Computer Vision Toolbox Model for YOLO v4 Object Detection support package.

Train and Detect Objects Using YOLOv4 Network

To train a YOLO v4 object detection network on a labeled dataset, use the `trainYOLOv4ObjectDetector` function . You must specify the class names and the predefined anchor boxes for the data set you use to train the network.

The training function returns the trained network as a `yolov4ObjectDetector` object. You can then use the `detect` function to detect unknown objects in a test image with the trained YOLO v4 object detector. To learn how to create a YOLO v4 object detector and train for object detection, see the “Object Detection Using YOLO v4 Deep Learning” on page 3-423 example.

Specify Anchor Boxes

The shape, size, and number of anchor boxes used for training impact the efficiency and accuracy of the YOLO v4 object detection network. The anchor boxes must closely represent the sizes and aspect ratios of the objects in the training data. The training data must contain both the ground truth images and labels. The size of the training images must be the same as the network input size, and the bounding box labels must correspond to the size of the training images.

You must assign the same number of anchor boxes to each detection head in the YOLO v4 network. The size of the anchor boxes assigned to each detection head must correspond to the size of the feature map output from the detection head. You must assign large anchor boxes to detection heads with lower resolution feature maps and small anchor boxes to detection heads with higher resolution feature maps.

For example, these steps show you how to specify anchor boxes to train a YOLO v4 network that has three detection heads with feature map sizes of 19-by-19, 38-by-38, and 76-by-76, respectively.

1. Assume that you specify four anchor boxes for each detection head. Then, the total number of anchor boxes that you use for training the network must be twelve. You can use the `estimateAnchorBoxes` function to automatically estimate the anchor boxes for your specified training data.

   ```
   numAnchors = 12;
   [anchors] = estimateAnchorBoxes(trainingData,numAnchors);
   ```

2. Compute the area of each anchor box and sort them in descending order.

   ```
   area = anchors(:,1).*anchors(:,2);
   [~,idx] = sort(area,"descend");
   sortedAnchors = anchors(idx,:)
   ```

3. There are three detection heads in the YOLO v4 network, so make three sets of four anchor boxes each.

   ```
   anchorBoxes = {sortedAnchors(1:4,:) sortedAnchors(5:8,:) sortedAnchors(9:12,:)};
   ```

4. Create a YOLO v4 object detection network by using the `yolov4ObjectDetector` function. Specify the classes and the sorted anchor boxes. The function assigns the first set of anchor boxes to the first detection head, the second set to the second detection head, and so on. The first four anchor boxes have large areas and must be assigned to the first detection head, which outputs the lower resolution 19-by-19 feature map. The next four anchor boxes must be assigned to the second detection head, which outputs the feature map of size 38-by-38. The last four anchor boxes are assigned to the third detection head that outputs the highest resolution 76-by-76 feature map.

   ```
   detector = yolov4ObjectDetector("csp-darknet53-coco","car",anchorBoxes);
   ```

5 Train the detector by using the `trainYOLOv4ObjectDetector` function.

```
detector = trainYOLOv4ObjectDetector(trainingData,detector,trainingOptions);
```

Transfer Learning

To perform transfer learning, use a pretrained convolutional neural network (CNN) as the base network for a YOLO v4 deep learning network. Configure the YOLO v4 deep learning network for training on a new data set by specifying the anchor boxes and the new object classes. Use the `yolov4ObjectDetector` object to create a custom YOLO v4 detection network from any pretrained CNN, such as `ResNet-50`. Then, train the network by using the `trainYOLOv4ObjectDetector` function.

For information about how to create a custom YOLO v4 object detector, see “Create Custom YOLO v4 Object Detector”.

Label Training Data for Deep Learning

You can use the **Image Labeler**, **Video Labeler**, or **Ground Truth Labeler** app to interactively label pixels and export label data for training. You can also use the apps to label rectangular regions of interest (ROIs) for object detection, scene labels for image classification, and pixels for semantic segmentation. To create training data from a ground truth object exported by any of the labelers, use the `objectDetectorTrainingData` or `pixelLabelTrainingData` functions. For more details, see “Training Data for Object Detection and Semantic Segmentation” on page 9-84.

References

[1] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. “YOLOv4: Optimal Speed and Accuracy of Object Detection.” *ArXiv:2004.10934 [Cs, Eess]*, April 22, 2020. https://arxiv.org/abs/2004.10934.

[2] Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You only look once: Unified, real-time object detection." *In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* 779–788. Las Vegas, NV: USA: IEEE, 2016. https://doi.org/10.1109/CVPR.2016.91.

See Also

Apps
Image Labeler | Ground Truth Labeler | Video Labeler | Deep Network Designer

Objects
`yolov4ObjectDetector`

Functions
`trainYOLOv4ObjectDetector` | `detect`

Related Examples

- “Object Detection Using YOLO v4 Deep Learning” on page 3-423

More About

- “Anchor Boxes for Object Detection” on page 17-32
- “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 17-55.

Object Detection Using R-CNN Algorithms

Models for object detection using regions with CNNs are based on the following three processes:

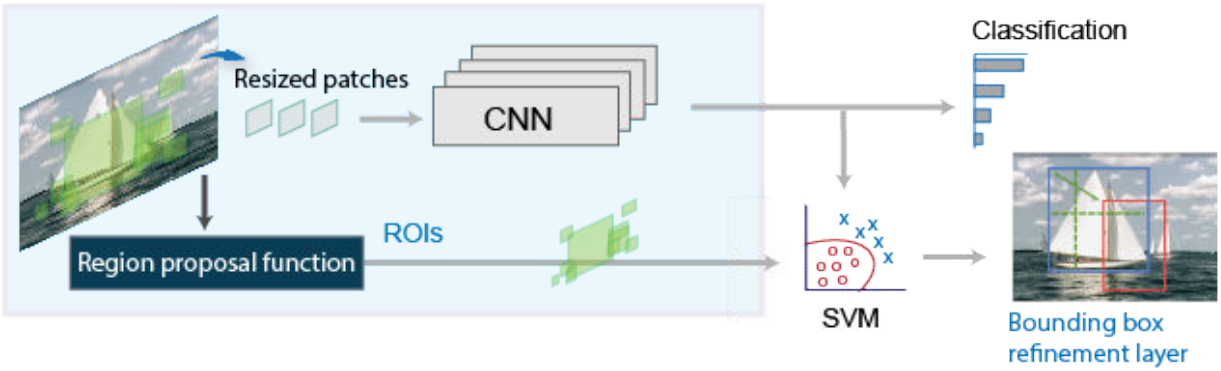
- Find regions in the image that might contain an object. These regions are called region proposals.
- Extract CNN features from the region proposals.
- Classify the objects using the extracted features.

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.

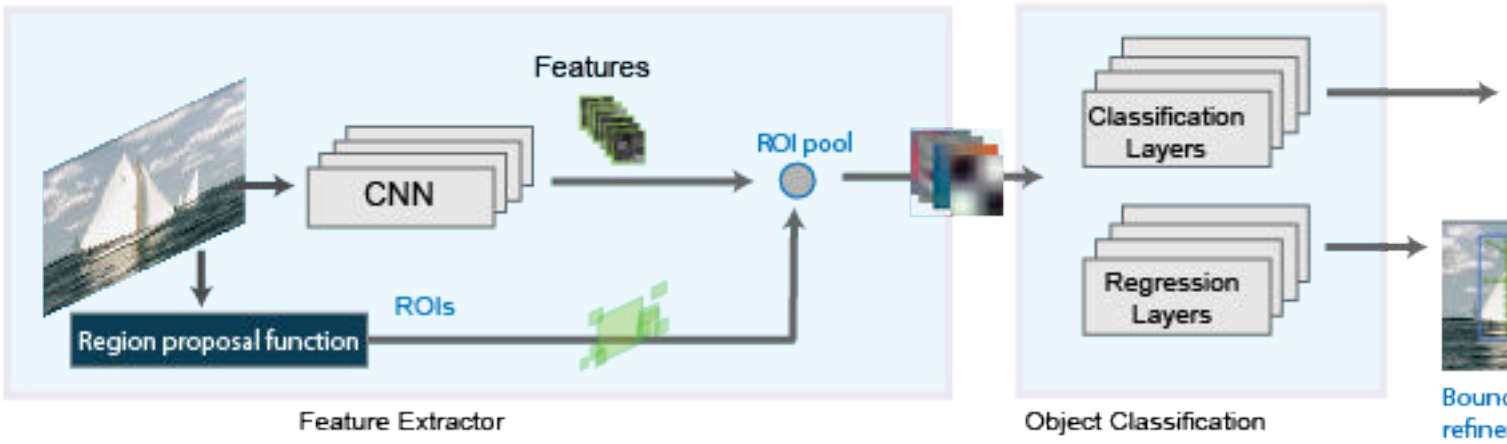
Use the `trainRCNNObjectDetector` function to train an R-CNN object detector. The function returns an `rcnnObjectDetector` object that detects objects in an image.



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.

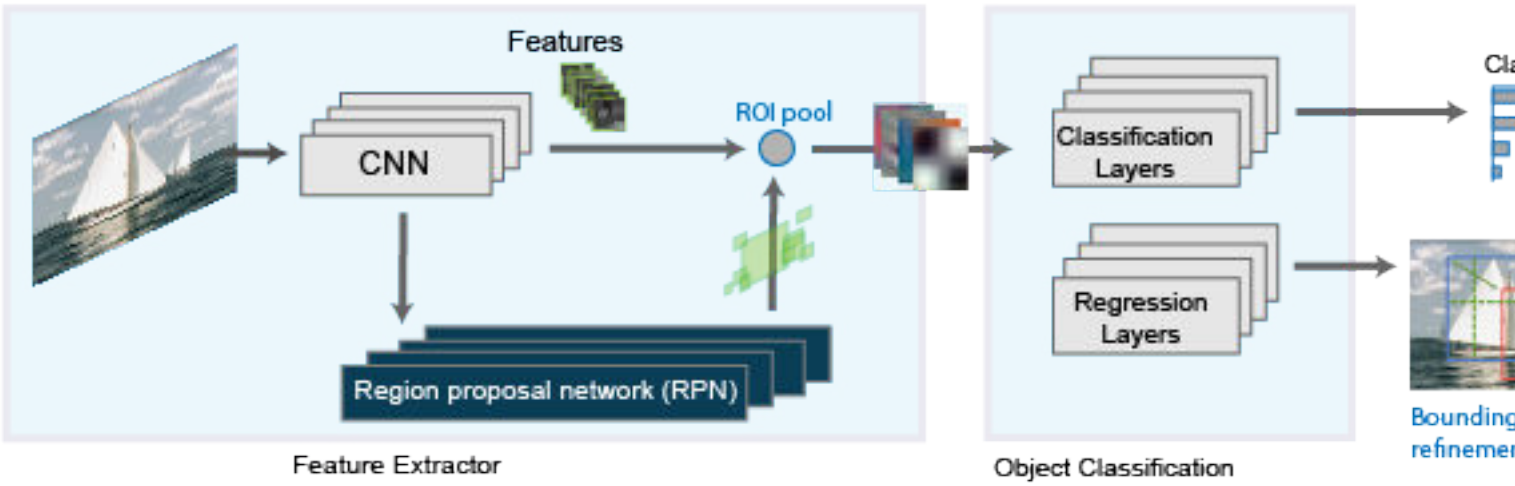
Use the `trainFastRCNNObjectDetector` function to train a Fast R-CNN object detector. The function returns a `fastRCNNObjectDetector` that detects objects from an image.



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 17-32. 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
`trainRCNNObjectDetector`	• Slow training and detection • Allows custom region proposal
`trainFastRCNNObjectDetector`	• Allows custom region proposal
`trainFasterRCNNObjectDetector`	• 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'`

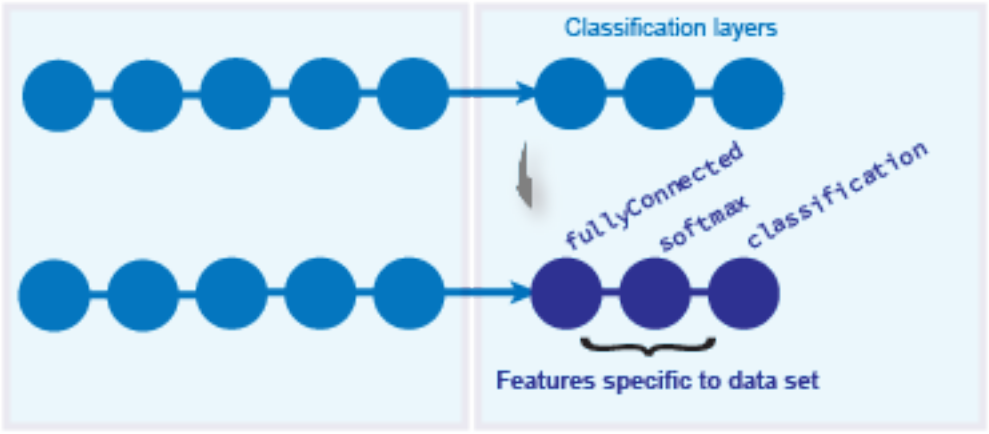
You can also design a custom model based on a pretrained image classification CNN. See the “Design an R-CNN, Fast R-CNN, and a Faster R-CNN Model” on page 17-52 section and the **Deep Network Designer** app.

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.

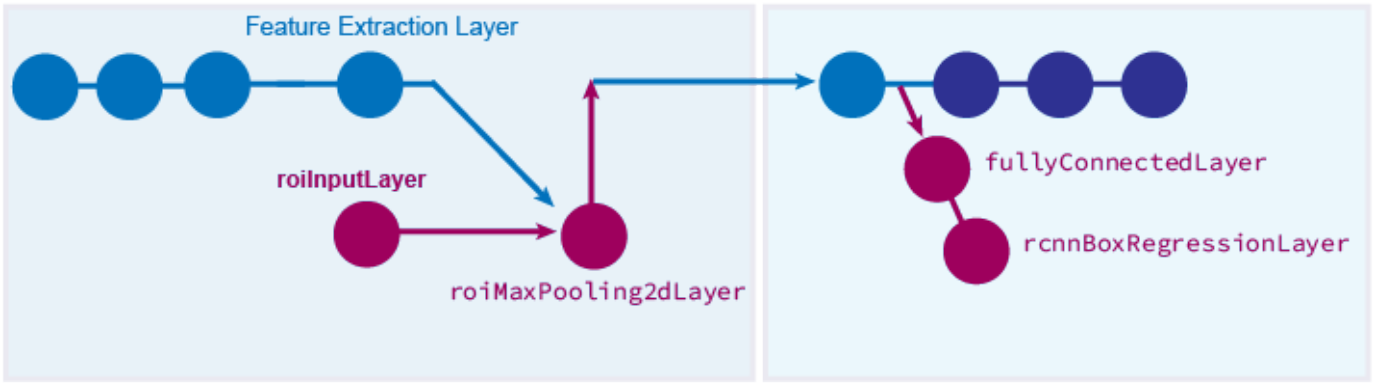
1. The basic R-CNN model starts with a pretrained network. The last three classification layers are replaced with new layers that are specific to the object classes you want to detect.

   For an example of how to create an R-CNN object detection network, see “Create R-CNN Object Detection Network”



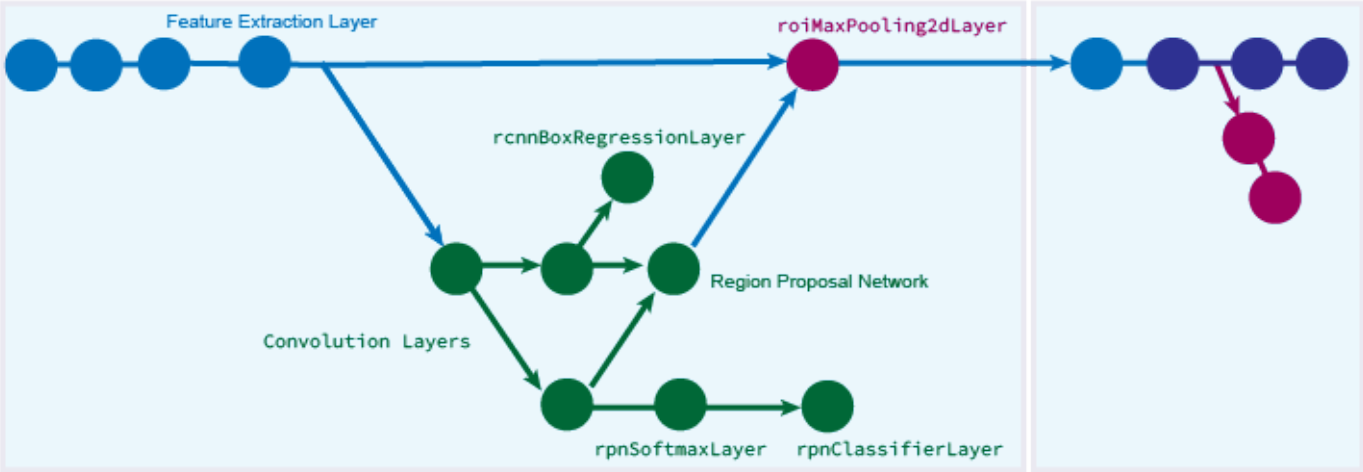
2. The Fast R-CNN model builds on the basic R-CNN model. A box regression layer is added to improve on the position of the object in the image by learning a set of box offsets. An ROI pooling layer is inserted into the network to pool CNN features for each region proposal.

   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** 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. To create training data from any of the labelers exported ground truth object, you can use the `objectDetectorTrainingData` or `pixelLabelTrainingData` functions. For more details, see “Training Data for Object Detection and Semantic Segmentation” on page 9-84.

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
Image Labeler | Ground Truth Labeler | Video Labeler | Deep Network Designer

Functions
`trainRCNNObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `fasterRCNNObjectDetector` | `fastRCNNObjectDetector` | `rcnnObjectDetector`

Related Examples

- “Train Object Detector Using R-CNN Deep Learning” on page 3-351
- “Object Detection Using Faster R-CNN Deep Learning” on page 3-364

More About

- “Anchor Boxes for Object Detection” on page 17-32
- “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.

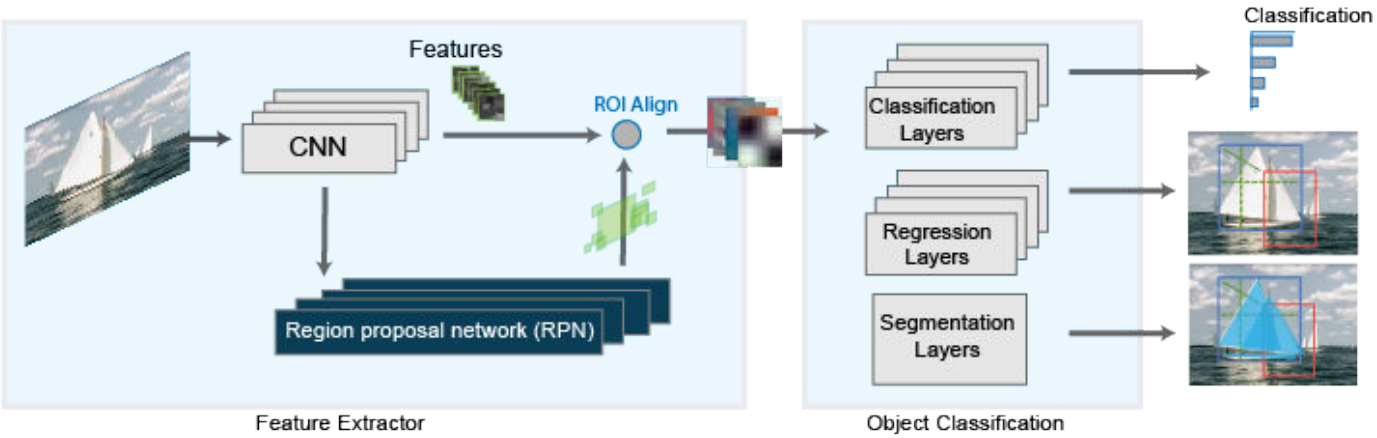
Mask R-CNN is a popular deep learning instance segmentation technique that performs pixel-level segmentation on detected objects [1]. The Mask R-CNN algorithm can accommodate multiple classes and overlapping objects.

You can create a pretrained Mask R-CNN network using the `maskrcnn` object. The network is trained on the MS-COCO data set and can detect objects of 80 different classes. To perform instance segmentation, pass the pretrained network to the `segmentObjects` function.

If you want to modify the network to detect additional classes, or to adjust other parameters of the network, then you can perform transfer learning. For an example that shows how to train a Mask R-CNN, see “Perform Instance Segmentation Using Mask R-CNN” on page 3-418.

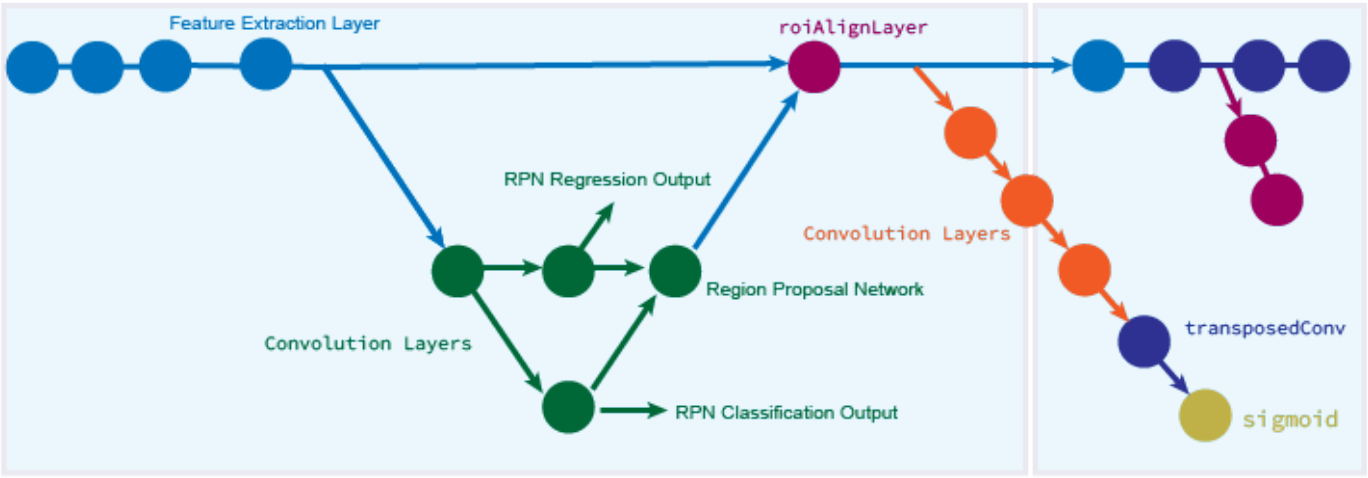
Mask R-CNN Network Architecture

The Mask R-CNN network consists of two stages. The first stage 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. Mask R-CNN replaces the ROI max pooling layer in Faster R-CNN 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 17-49.

This diagram shows a modified Faster R-CNN network on the left and a mask branch on the right.



To configure a Mask R-CNN network for transfer learning, specify the class names and anchor boxes when you create a `maskrcnn` object. You can optionally specify additional network properties including the network input size and the ROI pooling sizes.

Prepare Mask R-CNN Training Data

Load Data

To train a Mask R-CNN, you need the following data.

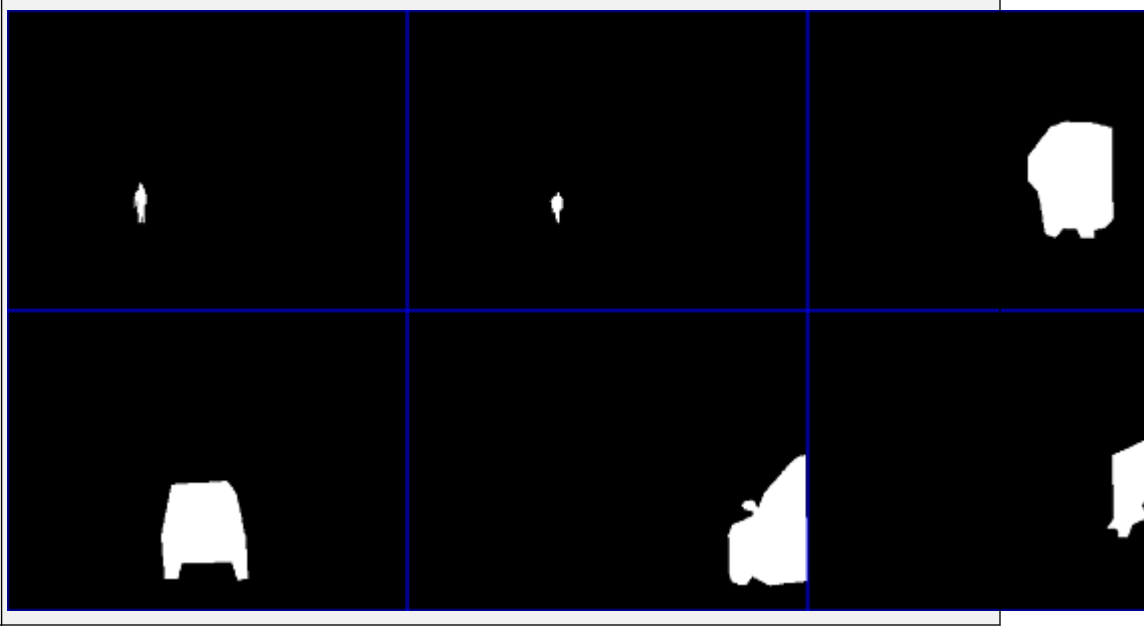
Data	Description
RGB image	RGB images that serve as network inputs, specified as *H*-by-*W*-by-3 numeric arrays. For example, this sample RGB image is a modified image from the CamVid data set [2] that has been edited to remove personally identifiable information. 

Ground-truth bounding boxes	Bounding boxes for objects in the RGB images, specified as a *NumObjects*-by-4 matrix, with rows in the format [*x y w h*]).
	
	For example, the `bboxes` variable shows the bounding boxes of six objects in the sample RGB image.
	
	`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`

Data	Description

Instance labels	Label of each instance, specified as a *NumObjects*-by-1 string vector or a *NumObjects*-by-1 cell array of character vectors.)
	
	For example, the `labels` variable shows the labels of six objects in the sample RGB image.
	
	`labels =`
	
	`6×1 cell array`
	
	`{'Person' }`
	`{'Person' }`
	`{'Vehicle'}`
	`{'Vehicle'}`
	`{'Vehicle'}`
	`{'Vehicle'}`

Data	Description

Instance masks	Masks for instances of objects. Mask data comes in two formats:
	• 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.
	The Mask R-CNN network requires binary masks, not polygon coordinates. To convert polygon coordinates to binary masks, use the `poly2mask` function. The `poly2mask` 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 `masks_polygon` variable to binary masks of size *h*-by-*w*-by-*numObjects*.
	`denseMasks = false([h,w,numObjects]);`
	`for i = 1:numObjects`
	`denseMasks(:,:,i) = poly2mask(masks_polygon{i}(:,1),masks_polygon{i}(:,2`
	`end`
	For example, this montage shows the binary masks of six objects in the sample RGB image.
	 

Create Datastore that Reads 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}. You can create a datastore in this format using these steps:

1. Create an `imageDatastore` that returns RGB image data
2. Create a `boxLabelDatastore` that returns bounding box data and instance labels as a two-column cell array

3. Create an `imageDatastore` and specify a custom read function that returns mask data as a binary matrix
4. Combine the three datastores using the `combine` function

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 `bboxresize` function to resize the bounding boxes.

For more information, see “Datastores for Deep Learning” (Deep Learning Toolbox).

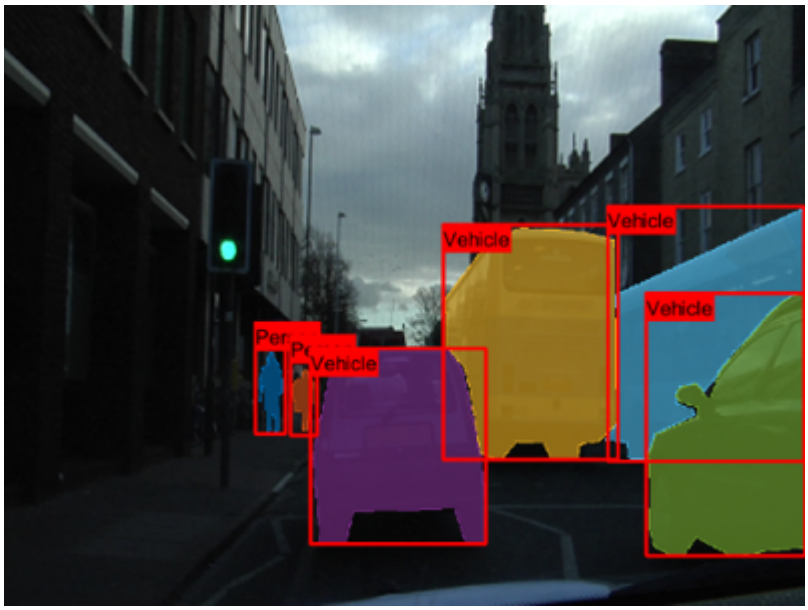
Visualize Training Data

To display the instance masks over the image, use the `insertObjectMask`. You can specify a colormap 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` colormap.

```
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");
```



Train Mask R-CNN Model

Train a Mask R-CNN network using the `trainMaskRCNN` function. For an example, see “Perform Instance Segmentation Using Mask R-CNN” on page 3-418.

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
`maskrcnn` | `trainMaskRCNN` | `segmentObjects`

Related Examples

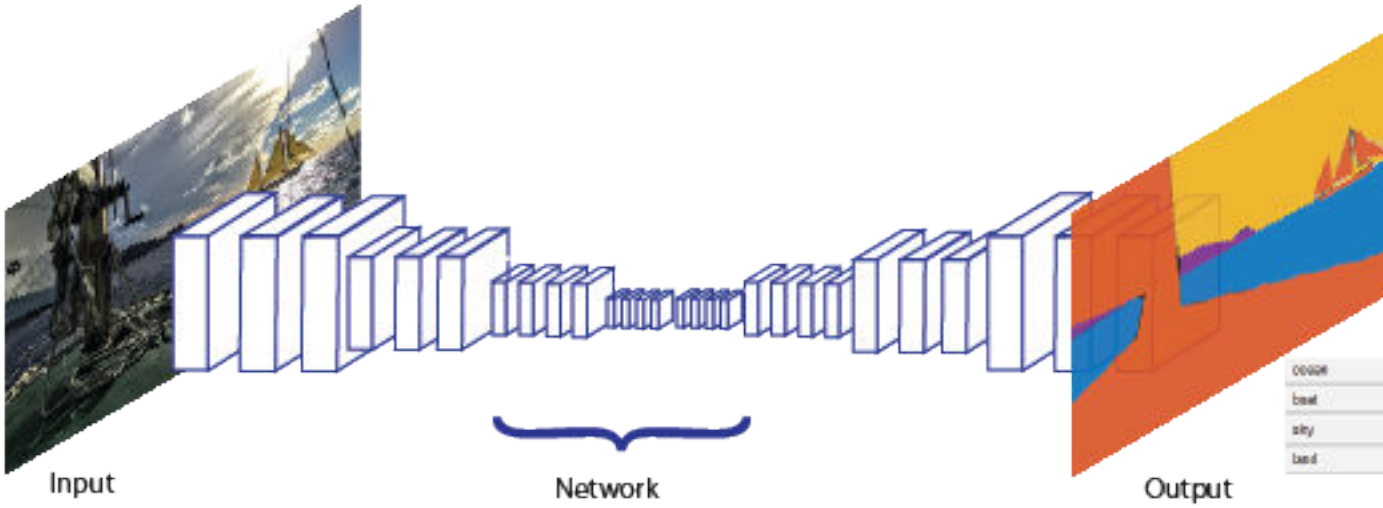
- “Perform Instance Segmentation Using Mask R-CNN” on page 3-418

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Datastores for Deep Learning” (Deep Learning Toolbox)

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*).



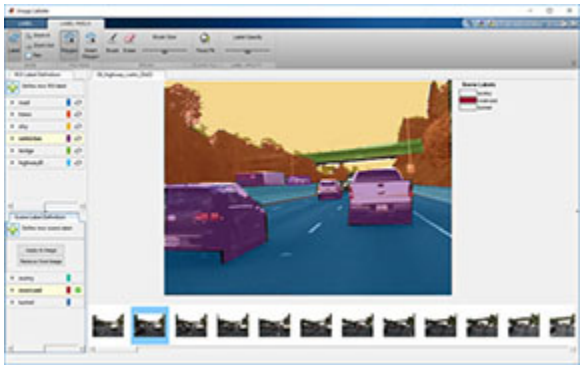
Applications for semantic segmentation include:

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

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**, **Video Labeler**, or **Ground Truth Labeler** (available in Automated Driving Toolbox) apps to interactively label pixels and export label data for training. The app can also be used to label rectangular regions of interest (ROIs) and scene labels for image classification.



Train and Test 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”

Segment Objects Using Pretrained DeepLabv3+ Network

MathWorks GitHub repository provides implementations of the latest pretrained deep learning networks to download and use for performing out-of-the-box inference.

For a list of all the latest MathWorks pretrained semantic segmentation models and examples, see MATLAB Deep Learning (GitHub).

See Also

Apps
Image Labeler

Functions
`semanticseg` | `semanticSegmentationMetrics` | `evaluateSemanticSegmentation` | `pixelLabelDatastore` | `segnetLayers` | `fcnLayers` | `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 3-222
- “Label Pixels for Semantic Segmentation” on page 9-14
- “Define Custom Pixel Classification Layer with Tversky Loss” on page 10-59
- “Semantic Segmentation Using Dilated Convolutions” on page 10-54
- “Calculate Segmentation Metrics in Block-Based Workflow” on page 3-239
- “Explore Semantic Segmentation Network Using Grad-CAM” on page 3-168

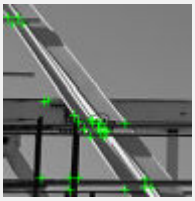
More About

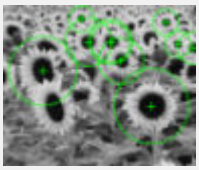
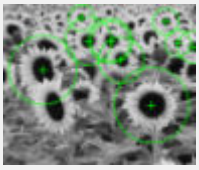
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

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]	 Corners Single-scale detection Point tracking, image registration with little or no scale change, corner detection in scenes of human origin, such as streets and indoor scenes.
	`detectMinEigenFeatures` Minimum eigenvalue algorithm Uses minimum eigenvalue metric to determine corner locations. [4]	
	`detectHarrisFeatures` Harris-Stephens algorithm More efficient than the minimum eigenvalue algorithm. [3]	
`BRISKPoints`	`detectBRISKFeatures` Binary Robust Invariant Scalable Keypoints (BRISK) algorithm [6]	 Corners Multiscale detection Point tracking, image registration, handles changes in scale and rotation, corner detection in scenes of human origin, such as streets and indoor scenes

Points Object	Returned By	Type of Feature
`SIFTPoints`	`detectSIFTFeatures` Scale-invariant feature transform	 Blobs Multiscale detection Object detection and image registration with scale and rotation changes
`SURFPoints`	`detectSURFFeatures` Speeded-up robust features (SURF) algorithm [11]	 Blobs Multiscale detection Object detection and image registration with scale and rotation changes
`ORBPoints`	`detectORBFeatures` Oriented FAST and Rotated BRIEF (ORB) method [13]	 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
`KAZEPoints`	`detectKAZEFeatures` KAZE is not an acronym, but a name derived from the Japanese word *kaze*, which means wind. The reference is to the flow of air ruled by nonlinear processes on a large scale. [12]	 Multi-scale blob features Reduced blurring of object boundaries

Points Object	Returned By	Type of Feature
`MSERRegions`	`detectMSERFeatures` Maximally stable extremal regions (MSER) algorithm [7] [8] [9] [10]	 Regions of uniform intensity Multi-scale detection Registration, wide baseline stereo calibration, text detection, object detection. Handles changes to scale and rotation. More robust to affine transforms in contrast to other detectors.

Functions That Accept Points Objects

Function	Description	
`relativeCameraPose`	Compute relative rotation and translation between camera poses	
`estimateFundamentalMatrix`	Estimate fundamental matrix from corresponding points in stereo images	
`estimateGeometricTransform2D`	Estimate geometric transform from matching point pairs	
`estimateUncalibratedRectification`	Uncalibrated stereo rectification	
`extractFeatures`	Extract interest point descriptors	
	Method	**Feature Vector**
	`BRISK`	The function sets the `Orientation` property of the `validPoints` output object to the orientation of the extracted features, in radians.
	`FREAK`	The function sets the `Orientation` property of the `validPoints` output object to the orientation of the extracted features, in radians.

Function	Description	
	SURF	The function sets the `Orientation` property of the `validPoints` output object to the orientation of the extracted features, in radians. When you use an `MSERRegions` object with the `SURF` method, the `Centroid` property of the object extracts SURF descriptors. The `Axes` property of the object selects the scale of the SURF descriptors such that the circle representing the feature has an area proportional to the MSER ellipse area. The scale is calculated as `1/4*sqrt((majorAxes/2).*(minorAxes/2))` and saturated to `1.6`, as required by the `SURFPoints` object.
	KAZE	Non-linear pyramid-based features. The function sets the `Orientation` property of the `validPoints` output object to the orientation of the extracted features, in radians. When you use an `MSERRegions` object with the `KAZE` method, the `Location` property of the object is used to extract KAZE descriptors. The `Axes` property of the object selects the scale of the KAZE descriptors such that the circle representing the feature has an area proportional to the MSER ellipse area.
	ORB	The function does not set the `Orientation` property of the `validPoints` output object to the orientation of the extracted features. By default, the `Orientation` property of `validPoints` is set to the `Orientation` property of the input `ORBPoints` object.
	Block	Simple square neighbhorhood. The `Block` method extracts only the neighborhoods fully contained within the image boundary. Therefore, the output, `validPoints`, can contain fewer points than the input `POINTS`.

Function	Description	
	`Auto`	The function selects the `Method` based on the class of the input points and implements: The `FREAK` method for a `cornerPoints` input object. The `SURF` method for a `SURFPoints` or `MSERRegions` input object. The `FREAK` method for a `BRISKPoints` input object. The `ORB` method for a `ORBPoints` input object. For an *M*-by-2 input matrix of [*x y*] coordinates, the function implements the `Block` method.
`extractHOGFeatures`	Extract histogram of oriented gradients (HOG) features	
`insertMarker`	Insert markers in image or video	
`showMatchedFeatures`	Display corresponding feature points	
`triangulate`	3-D locations of undistorted matching points in stereo images	
`undistortPoints`	Correct point coordinates for lens distortion	

References

[1] Rosten, E., and T. Drummond, “Machine Learning for High-Speed Corner Detection.” *9th European Conference on Computer Vision*. Vol. 1, 2006, pp. 430–443.

[2] Mikolajczyk, K., and C. Schmid. “A performance evaluation of local descriptors.” *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Vol. 27, Issue 10, 2005, pp. 1615–1630.

[3] Harris, C., and M. J. Stephens. “A Combined Corner and Edge Detector.” *Proceedings of the 4th Alvey Vision Conference*. August 1988, pp. 147–152.

[4] Shi, J., and C. Tomasi. “Good Features to Track.” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. June 1994, pp. 593–600.

[5] Tuytelaars, T., and K. Mikolajczyk. “Local Invariant Feature Detectors: A Survey.” *Foundations and Trends in Computer Graphics and Vision*. Vol. 3, Issue 3, 2007, pp. 177–280.

[6] Leutenegger, S., M. Chli, and R. Siegwart. “BRISK: Binary Robust Invariant Scalable Keypoints.” *Proceedings of the IEEE International Conference*. ICCV, 2011.

[7] Nister, D., and H. Stewenius. "Linear Time Maximally Stable Extremal Regions." *Lecture Notes in Computer Science. 10th European Conference on Computer Vision*. Marseille, France: 2008, no. 5303, pp. 183–196.

[8] Matas, J., O. Chum, M. Urba, and T. Pajdla. "Robust wide-baseline stereo from maximally stable extremal regions." *Proceedings of British Machine Vision Conference*. 2002, pp. 384–396.

[9] Obdrzalek D., S. Basovnik, L. Mach, and A. Mikulik. "Detecting Scene Elements Using Maximally Stable Colour Regions." *Communications in Computer and Information Science*. La Ferte-Bernard, France: 2009, Vol. 82 CCIS (2010 12 01), pp 107–115.

[10] Mikolajczyk, K., T. Tuytelaars, C. Schmid, A. Zisserman, T. Kadir, and L. Van Gool. "A Comparison of Affine Region Detectors." *International Journal of Computer Vision*. Vol. 65, No. 1–2, November, 2005, pp. 43–72 .

[11] Bay, H., A. Ess, T. Tuytelaars, and L. Van Gool. “SURF:Speeded Up Robust Features.” *Computer Vision and Image Understanding (CVIU)*.Vol. 110, No. 3, 2008, pp. 346–359.

[12] Alcantarilla, P.F., A. Bartoli, and A.J. Davison. "KAZE Features", *ECCV 2012, Part VI, LNCS 7577* pp. 214, 2012

[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.

[15] Lowe, David G.. "Distinctive Image Features from Scale-Invariant Keypoints." *Int. J. Comput. Vision* 60 , no. 2 (2004): 91--110.

See Also

More About

- Local Feature Detection and Extraction on page 17-71

See Also

Related Examples

- “Object Detection in a Cluttered Scene Using Point Feature Matching” on page 3-211

Local Feature Detection and Extraction

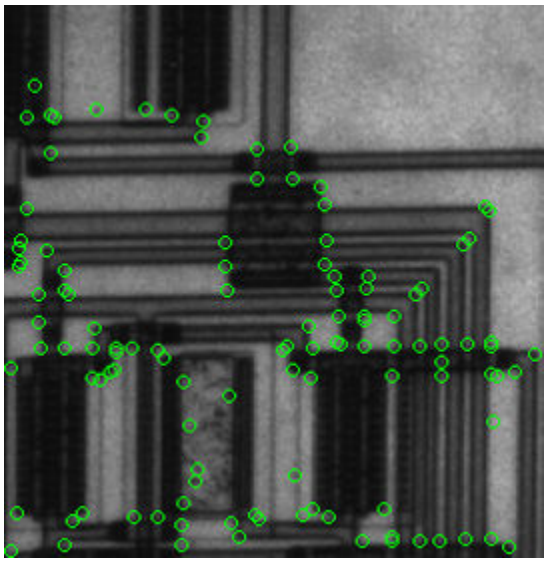
Local features and their descriptors, which are a compact vector representations of a local neighborhood, are the building blocks of many computer vision algorithms. Their applications include image registration, object detection and classification, tracking, and motion estimation. Using local features 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 SIFT, SURF, KAZE, and MSER methods for detecting blob features. The toolbox includes the SIFT, 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 17.1. 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 17-113.

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 4-34
Object detection	“Object Detection in a Cluttered Scene Using Point Feature Matching” on page 3-211
Object recognition	“Digit Classification Using HOG Features” on page 4-21
Object tracking	“Face Detection and Tracking Using the KLT Algorithm” on page 8-20
Image category recognition	“Image Category Classification Using Bag of Features” on page 3-275
Finding geometry of a stereo system	“Uncalibrated Stereo Image Rectification” on page 1-149
3-D reconstruction	“Structure From Motion From Two Views” on page 1-92, “Structure From Motion From Multiple Views” on page 1-141
Image retrieval	“Image Retrieval Using Customized Bag of Features” on page 3-291

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: • 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 SIFT, SURF, ORB, or KAZE feature detector and descriptor. The SURF and the KAZE methods are computationally intensive.
Performance requirements: • Real-time performance required • Accuracy versus speed	Binary descriptors are generally faster but less accurate than gradient-based descriptors. For greater accuracy, use several detectors and descriptors at the same time.

Choose a Detection Function Based on Feature Type

Detector	Feature Type	Function	Scale Independent
FAST [1]	Corner	`detectFASTFeatures`	No
Minimum eigenvalue algorithm [4]	Corner	`detectMinEigenFeatures`	No
Corner detector [3]	Corner	`detectHarrisFeatures`	No
SIFT [14] on page 17-88	Blob	`detectSIFTFeatures`	Yes
SURF [11]	Blob	`detectSURFFeatures`	Yes
KAZE [12]	Blob	`detectKAZEFeatures`	Yes
BRISK [6]	Corner	`detectBRISKFeatures`	Yes
MSER [8]	Region with uniform intensity	`detectMSERFeatures`	Yes
ORB [13]	Corner	`detectORBFeatures`	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 Correspondences	Classification
HOG	No	`extractHOGFeatures(I, ...)`	No	No	No	Yes
LBP	No	`extractLBPFeatures(I, ...)`	No	Yes	No	Yes
SIFT	No	`extractFeatures(I,points,'Method','SIFT')`	Yes	Yes	Yes	Yes
SURF	No	`extractFeatures(I,points,'Method','SURF')`	Yes	Yes	Yes	Yes
KAZE	No	`extractFeatures(I,points,'Method','KAZE')`	Yes	Yes	Yes	Yes
FREAK	Yes	`extractFeatures(I,points,'Method','FREAK')`	Yes	Yes	Yes	No
BRISK	Yes	`extractFeatures(I,points,'Method','BRISK')`	Yes	Yes	Yes	No
ORB	Yes	`extractFeatures(I,points,'Method','ORB')`	No	Yes	Yes	No
• Block • Simple pixel neighborhood around a keypoint	No	`extractFeatures(I,points,'Method','Block')`	No	No	Yes	Yes

Note

- The `extractFeatures` function provides different extraction methods to best match the requirements of your application. When you do not specify the `'Method'` input for the `extractFeatures` function, the function automatically selects the method based on the type of input point class.
- Binary descriptors are fast but less precise in terms of localization. They are not suitable for classification tasks. The `extractFeatures` function returns a `binaryFeatures` object. This object enables the Hamming-distance-based matching metric used in the `matchFeatures` function.

Use Local Features

Registering two images is a simple way to understand local features. This example finds a geometric transformation between two images. It uses local features to find well-localized anchor points.

Display two images

The first image is the original image.

```
original = imread('cameraman.tif');
figure;
imshow(original);
```

The second image is the original image rotated and scaled.

```
scale = 1.3;
J = imresize(original,scale);
theta = 31;
distorted = imrotate(J,theta);
figure
imshow(distorted)
```

Detect matching features between the original and distorted image

Detecting the matching SURF features is the first step in determining the transform needed to correct the distorted image.

```
ptsOriginal  = detectSURFFeatures(original);
ptsDistorted = detectSURFFeatures(distorted);
```

Extract features and compare the detected blobs between the two images

The detection step found several roughly corresponding blob structures in both images. Compare the detected blob features. This process is facilitated by feature extraction, which determines a local patch descriptor.

```
[featuresOriginal,validPtsOriginal] = ...
            extractFeatures(original,ptsOriginal);
[featuresDistorted,validPtsDistorted] = ...
            extractFeatures(distorted,ptsDistorted);
```

It is possible that not all of the original points were used to extract descriptors. Points might have been rejected if they were too close to the image border. Therefore, the valid points are returned in addition to the feature descriptors.

The patch size used to compute the descriptors is determined during the feature extraction step. The patch size corresponds to the scale at which the feature is detected. Regardless of the patch size, the two feature vectors, `featuresOriginal` and `featuresDistorted`, are computed in such a way that they are of equal length. The descriptors enable you to compare detected features, regardless of their size and rotation.

Find candidate matches

Obtain candidate matches between the features by inputting the descriptors to the `matchFeatures` function. Candidate matches imply that the results can contain some invalid matches. Two patches that match can indicate like features but might not be a correct match. A table corner can look like a chair corner, but the two features are obviously not a match.

```
indexPairs = matchFeatures(featuresOriginal,featuresDistorted);
```

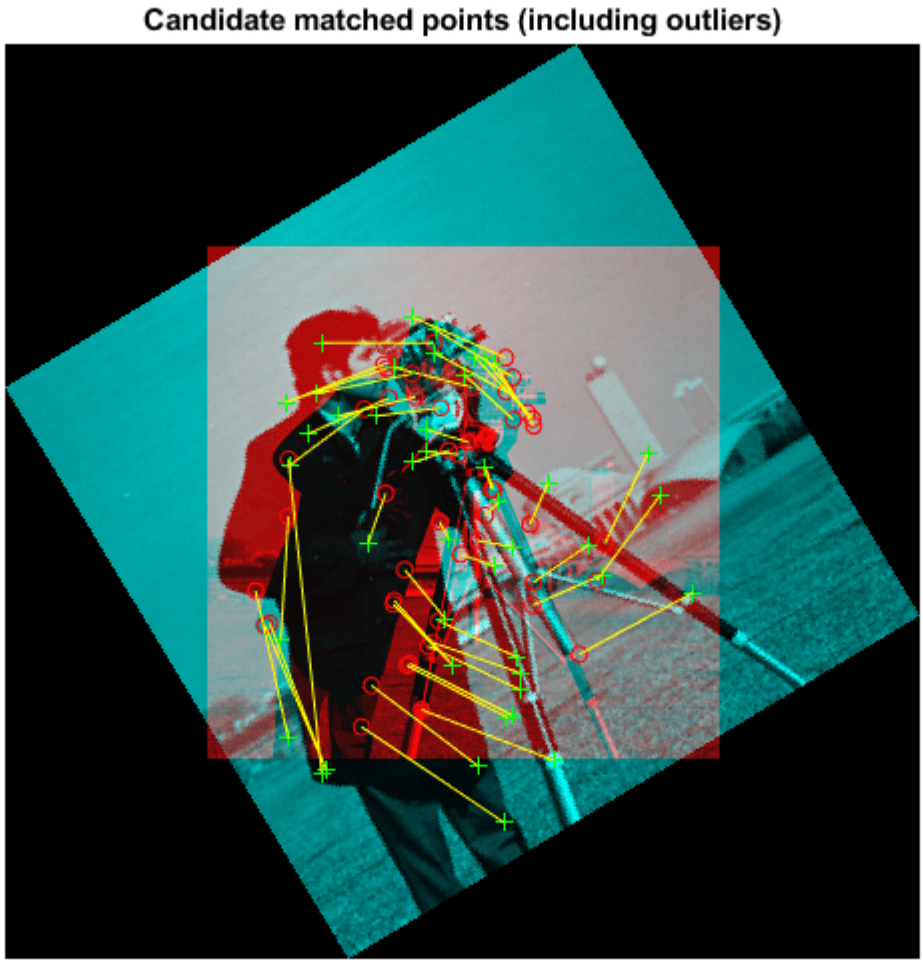
Find point locations from both images

Each row of the returned `indexPairs` contains two indices of candidate feature matches between the images. Use the indices to collect the actual point locations from both images.

```
matchedOriginal  = validPtsOriginal(indexPairs(:,1));
matchedDistorted = validPtsDistorted(indexPairs(:,2));
```

Display the candidate matches

```
figure
showMatchedFeatures(original,distorted,matchedOriginal,matchedDistorted)
title('Candidate matched points (including outliers)')
```



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.

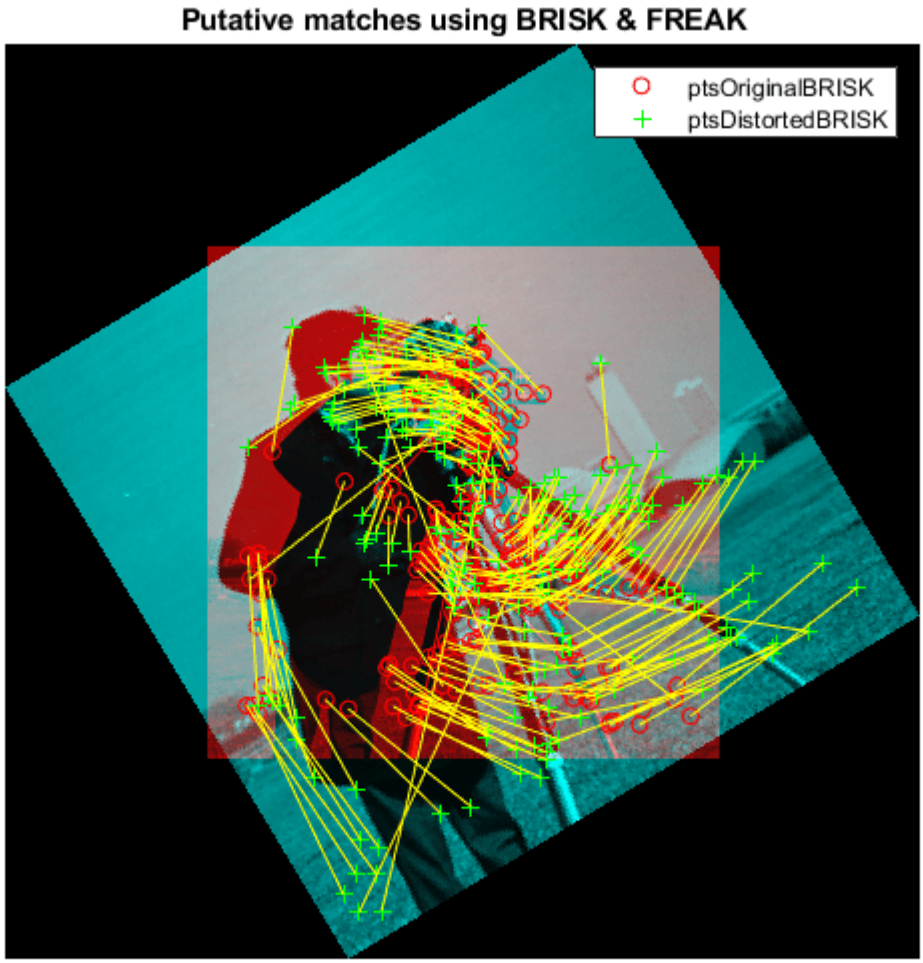
```
indexPairsBRISK = matchFeatures(featuresOriginalFREAK,...
            featuresDistortedFREAK,'MatchThreshold',40,'MaxRatio',0.8);

indexPairsSURF = matchFeatures(featuresOriginalSURF,featuresDistortedSURF);
```

Obtain candidate matched points for BRISK and SURF.

```
matchedOriginalBRISK  = validPtsOriginalBRISK(indexPairsBRISK(:,1));
matchedDistortedBRISK = validPtsDistortedBRISK(indexPairsBRISK(:,2));

matchedOriginalSURF  = validPtsOriginalSURF(indexPairsSURF(:,1));
matchedDistortedSURF = validPtsDistortedSURF(indexPairsSURF(:,2));
```

Visualize the BRISK putative matches.

```
figure
showMatchedFeatures(original,distorted,matchedOriginalBRISK,...
            matchedDistortedBRISK)
title('Putative matches using BRISK & FREAK')
legend('ptsOriginalBRISK','ptsDistortedBRISK')
```



Combine the candidate matched BRISK and SURF local features. Use the `Location` property to combine the point locations from BRISK and SURF features.

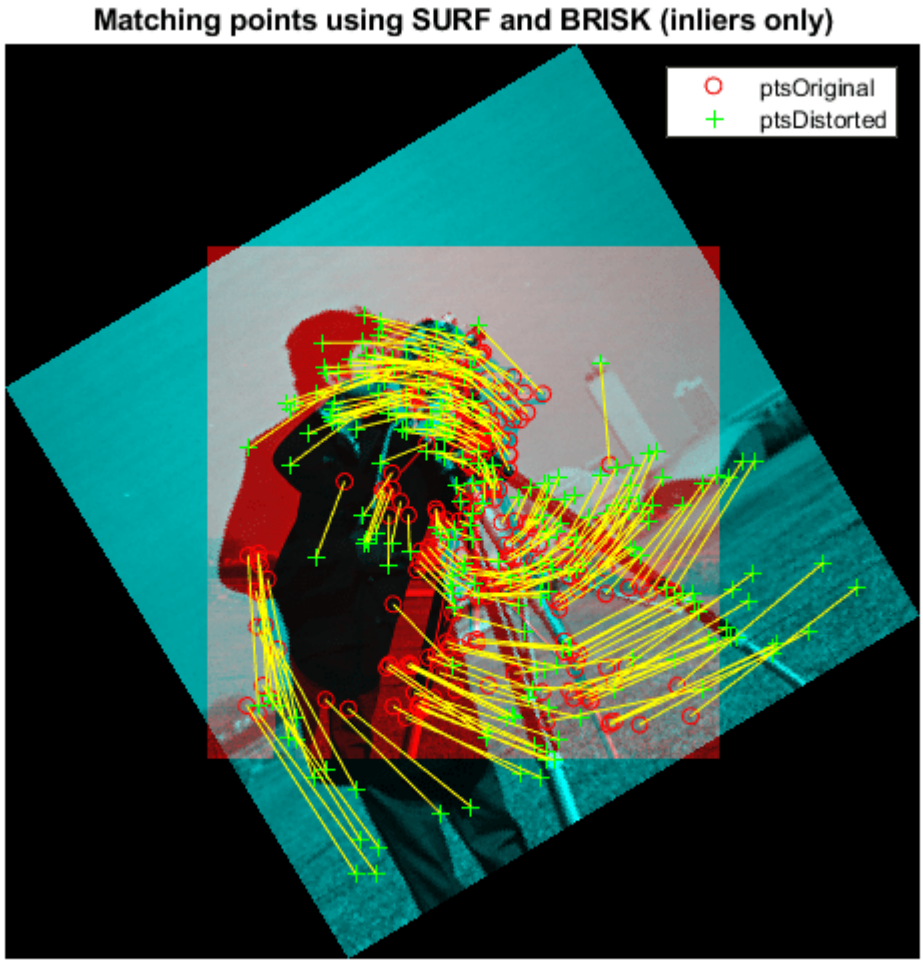
```
matchedOriginalXY  = ...
    [matchedOriginalSURF.Location; matchedOriginalBRISK.Location];
matchedDistortedXY = ...
    [matchedDistortedSURF.Location; matchedDistortedBRISK.Location];
```

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.

[14] Lowe, David G.. "Distinctive Image Features from Scale-Invariant Keypoints." *Int. J. Comput. Vision* 60 , no. 2 (2004): 91--110.

See Also

Related Examples

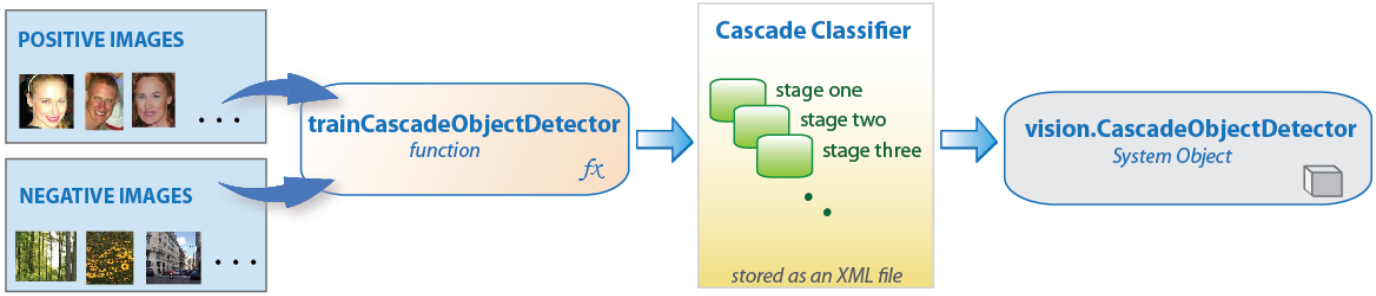
- “Detect BRISK Points in an Image and Mark Their Locations”
- “Find Corner Points in an Image Using the FAST Algorithm”
- “Find Corner Points Using the Harris-Stephens Algorithm”
- “Find Corner Points Using the Eigenvalue Algorithm”
- “Find MSER Regions in an Image”
- “Detect SURF Interest Points in a Grayscale Image”
- “Automatically Detect and Recognize Text Using MSER and OCR” on page 4-2
- “Object Detection in a Cluttered Scene Using Point Feature Matching” on page 3-211

Get Started with Cascade Object Detector

In this section...
“Why Train a Detector?” on page 17-89
“What Kinds of Objects Can You Detect?” on page 17-89
“How Does the Cascade Classifier Work?” on page 17-89
“Create a Cascade Classifier Using the trainCascadeObjectDetector” on page 17-90
“Troubleshooting” on page 17-93
“Examples” on page 17-95
“Train Stop Sign Detector” on page 17-99

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.

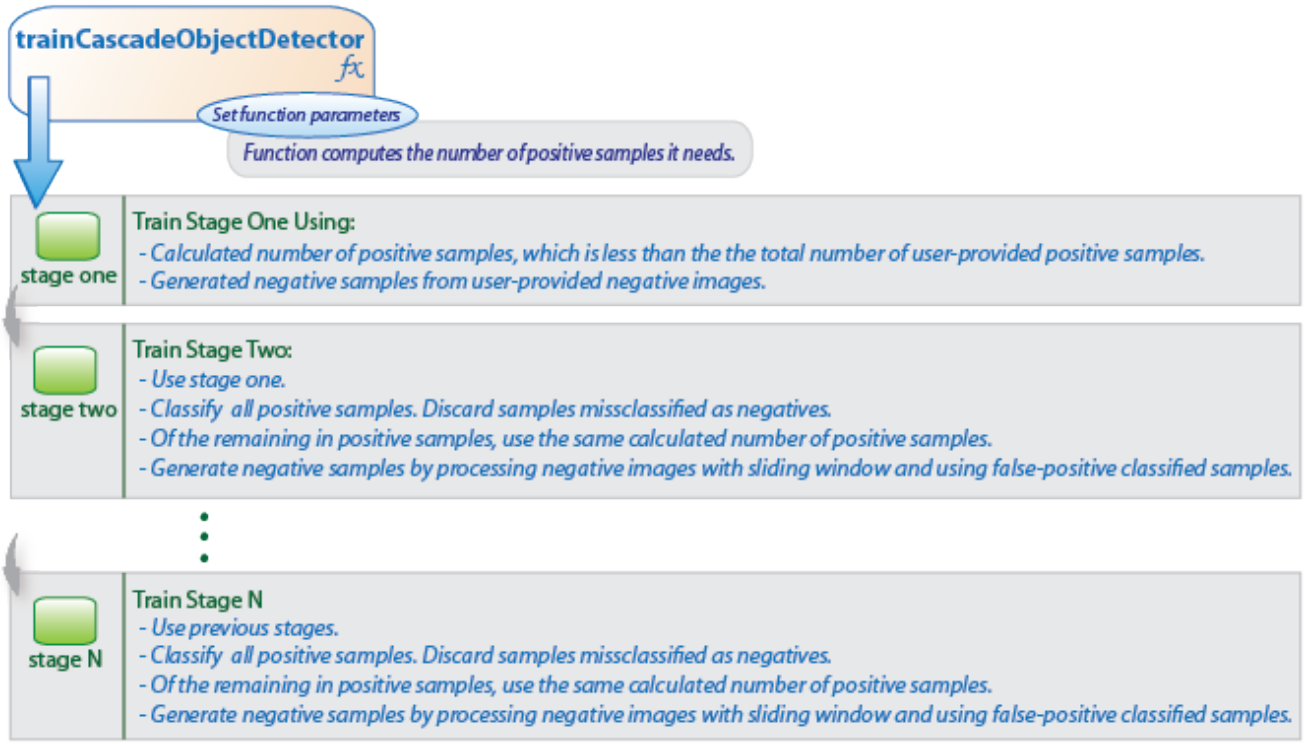
- A true positive occurs when a positive sample is correctly classified.
- A false positive occurs when a negative sample is mistakenly classified as positive.
- A false negative occurs when a positive sample is mistakenly classified as negative.

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.



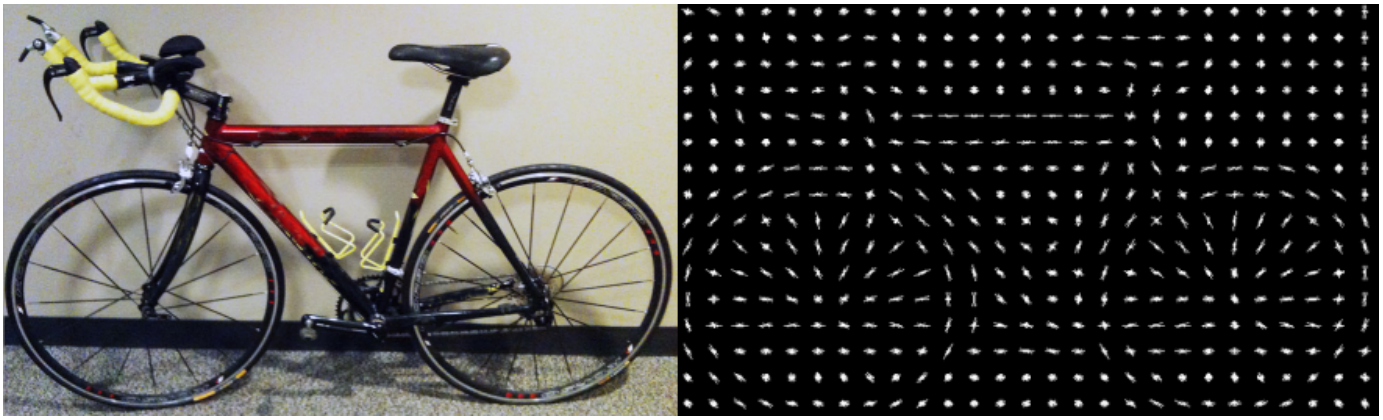
Considerations when Setting Parameters

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

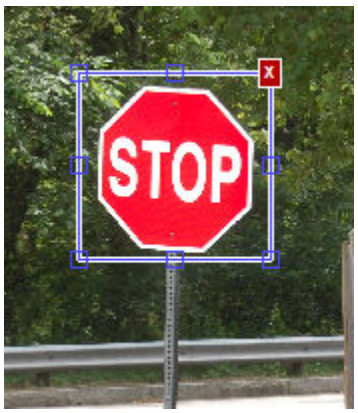
Choose the feature that suits the type of object detection you need. The `trainCascadeObjectDetector` supports three types of features: Haar, local binary patterns (LBP), and histograms of oriented gradients (HOG). Haar and LBP features are often used to detect faces because they work well for representing 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

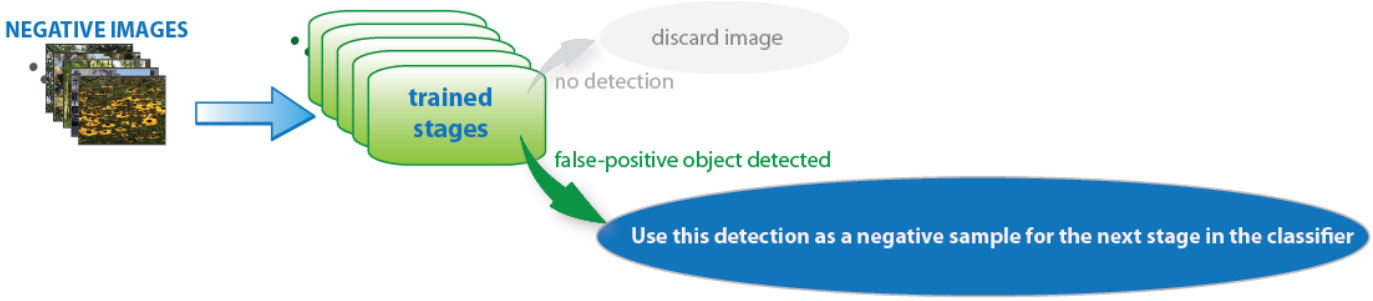
To create positive samples easily, you can use the **Image Labeler** app. The Image Labeler provides an easy way to label positive samples by interactively specifying rectangular regions of interest (ROIs).



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:

number of positive samples = `floor`(`totalPositiveSamples` / (1 + (`NumCascadeStages` - 1) * (1 - `TruePositiveRate`)))

This calculation does not guarantee that the same number of positive samples are available for each stage. The reason is that it is impossible to predict with certainty how many positive samples will be rejected as negatives. The training continues as long as the number of positive samples available to train a stage is greater than 10% of the number of samples the function determined automatically using the preceding formula. If there are not enough positive samples the training stops and the function issues a warning. The function also outputs a classifier consisting of the stages that it had trained up to that point. If the training stops, you can add more positive samples. Alternatively, you can increase `TruePositiveRate`. Reducing the number of stages can also work, but such reduction can also result in a higher overall false alarm rate.

What to do if you run out of negative samples?

The function calculates the number of negative samples used at each stage. This calculation is done by multiplying the number of positive samples used at each stage by the value of `NegativeSamplesFactor`.

Just as with positive samples, there is no guarantee that the calculated number of negative samples are always available for a particular stage. The `trainCascadeObjectDetector` function generates negative samples from the negative images. However, with each new stage, the overall false alarm rate of the cascade classifier decreases, making it less likely to find the negative samples.

The training continues as long as the number of negative samples available to train a stage is greater than 10% of the calculated number of negative samples. If there are not enough negative samples, the training stops and the function issues a warning. It outputs a classifier consisting of the stages that it had trained up to that point. When the training stops, the best approach is to add more negative images. Alternatively, you can reduce the number of stages or increase the false positive rate.

Examples

Train a Five-Stage Stop-Sign Detector

This example shows you how to set up and train a five-stage, stop-sign detector, using 86 positive samples. The default value for `TruePositiveRate` is `0.995`.

Step 1: Load the positive samples data from a MAT-file. In this example, file names and bounding boxes are contained in the array of structures labeled `'data'`.

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

Step 2: Add the image directory to the MATLAB path.

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

Step 3: Specify the folder with negative images.

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

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector.xml',data,negativeFolder,'FalseAlarmRate',0.2,'NumCascadeStages',5);
```

Computer Vision Toolbox software returns the following message:

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 86 of 86 positive samples per stage
Using at most 172 negative samples per stage

Training stage 1 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 2 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 3 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 4 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 5 of 5
[..................................................................]
Used 86 positive and 172 negative samples

Training complete
```

All 86 positive samples were used to train each stage. This high rate occurs because the true positive rate is very high relative to the number of positive samples.

Train a Five-Stage Stop-Sign Detector with a Decreased True Positive Rate

This example shows you how to train a stop-sign detector on the same data set as the first example, (steps 1–3), but with the `TruePositiveRate` decreased to 0.98.

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector_tpr0_98.xml',data,negativeFolder,...
'FalseAlarmRate',0.2,'NumCascadeStages', 5,...
'TruePositiveRate', 0.98);
```

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 79 of 86 positive samples per stage
Using at most 158 negative samples per stage

Training stage 1 of 5
[..................................................................]
Used 79 positive and 158 negative samples

Training stage 2 of 5
[..................................................................]
Used 79 positive and 158 negative samples

Training stage 3 of 5
[..................................................................]
Used 79 positive and 158 negative samples

Training stage 4 of 5
[..................................................................]
Used 79 positive and 158 negative samples

Training stage 5 of 5
[..................................................................]
Used 79 positive and 85 negative samples

Training complete
```

Only 79 of the total 86 positive samples were used to train each stage. This lowered rate occurs because the true positive rate was low enough for the function to start rejecting some of the positive samples as false negatives.

Train a Ten-Stage Stop-Sign Detector

This example shows you how to train a stop-sign detector on the same data set as the first example, (steps 1–3), but with the number of stages increased to 10.

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector_10stages.xml',data,negativeFolder,...
'FalseAlarmRate',0.2,'NumCascadeStages',10);
```

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 86 of 86 positive samples per stage
Using at most 172 negative samples per stage

Training stage 1 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 2 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 3 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 4 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 5 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 6 of 10
[..................................................................]
Used 86 positive and 33 negative samples

Training stage 7 of 10
[.....................................................Warning:
Unable to generate a sufficient number of negative samples for this stage.
Consider reducing the number of stages, reducing the false alarm rate
or adding more negative images.

Cannot find enough samples for training.
Training will halt and return cascade detector with 6 stages
Training complete
```

In this case, `NegativeSamplesFactor` was set to 2, therefore the number of negative samples used to train each stage was 172. Notice that the function generated only 33 negative samples for stage 6 and was not able to train stage 7 at all. This condition occurs because the number of negatives in stage 7 was less than 17, (roughly half of the previous number of negative samples). The function produced a stop-sign detector with 6 stages, instead of the 10 previously specified. The resulting overall false alarm rate is 0.2^7=1.28e-05, while the expected false alarm rate is 1.024e-07.

At this point, you can add more negative images, reduce the number of stages, or increase the false positive rate. For example, you can increase the false positive rate, `FalseAlarmRate`, to 0.5. The expected overall false-positive rate in this case is 0.0039.

Step 4: Train the detector.

```
trainCascadeObjectDetector('stopSignDetector_10stages_far0_5.xml',data,negativeFolder,...
'FalseAlarmRate',0.5,'NumCascadeStages',10);
```

```
Automatically setting ObjectTrainingSize to [ 33, 32 ]
Using at most 86 of 86 positive samples per stage
Using at most 172 negative samples per stage

Training stage 1 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 2 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 3 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 4 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 5 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 6 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 7 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 8 of 10
[..................................................................]
Used 86 positive and 172 negative samples

Training stage 9 of 10
[..................................................
Very low false alarm rate 0.000587108 reached in stage.
 Training will halt and return cascade detector with 8 stages
Training complete
```

This time the function trains eight stages before the threshold reaches the overall false alarm rate of 0.000587108 and training stops.

Train Stop Sign Detector

Load the positive samples data from a MAT file. The file contains the ground truth, specified as table of bounding boxes for several object categories. The grount truth was labeled and exported from the `Image Labeler` app.

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

Prefix the fullpath to the stop sign images.

```
stopSigns = fullfile(toolboxdir('vision'),'visiondata',stopSignsAndCars{:,1});
```

Create datastores to load the ground truth data for stop signs.

```
imds = imageDatastore(stopSigns);
blds = boxLabelDatastore(stopSignsAndCars(:,2));
```

Combine the image and box label datastores.

```
positiveInstances = combine(imds,blds);
```

Add the image folder path to the MATLAB path.

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

Specify a 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
```

```
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: 0 seconds

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

Training stage 3 of 5
[..................................................................]
Used 42 positive and 84 negative samples
Time to train stage 3: 2 seconds
```

```
Training stage 4 of 5
[..................................................................]
Used 42 positive and 84 negative samples
Time to train stage 4: 6 seconds

Training stage 5 of 5
[..................................................................]
Used 42 positive and 17 negative samples
Time to train stage 5: 9 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 in the test image.

```
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 folder from the path.

```
rmpath(imDir);
```

See Also

More About

- “Get Started with the Image Labeler” on page 9-30

External Websites

- Cascade Trainer

Train Optical Character Recognition for Custom Fonts

In this section...
“Open the OCR Trainer App” on page 17-102 “Train OCR” on page 17-102 “App Controls” on page 17-104

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.

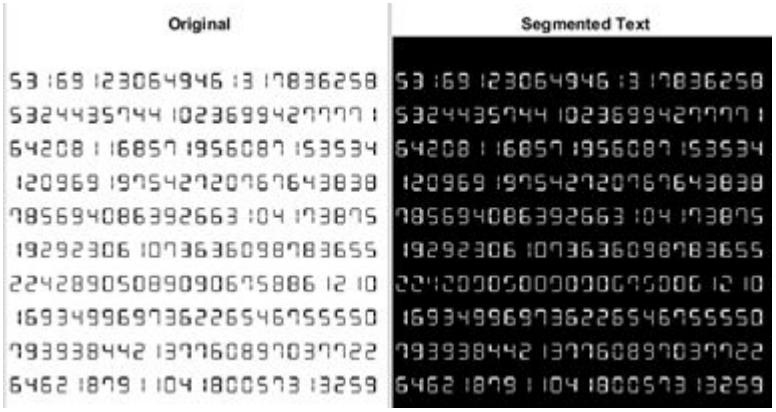
Open the OCR Trainer App

- 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.

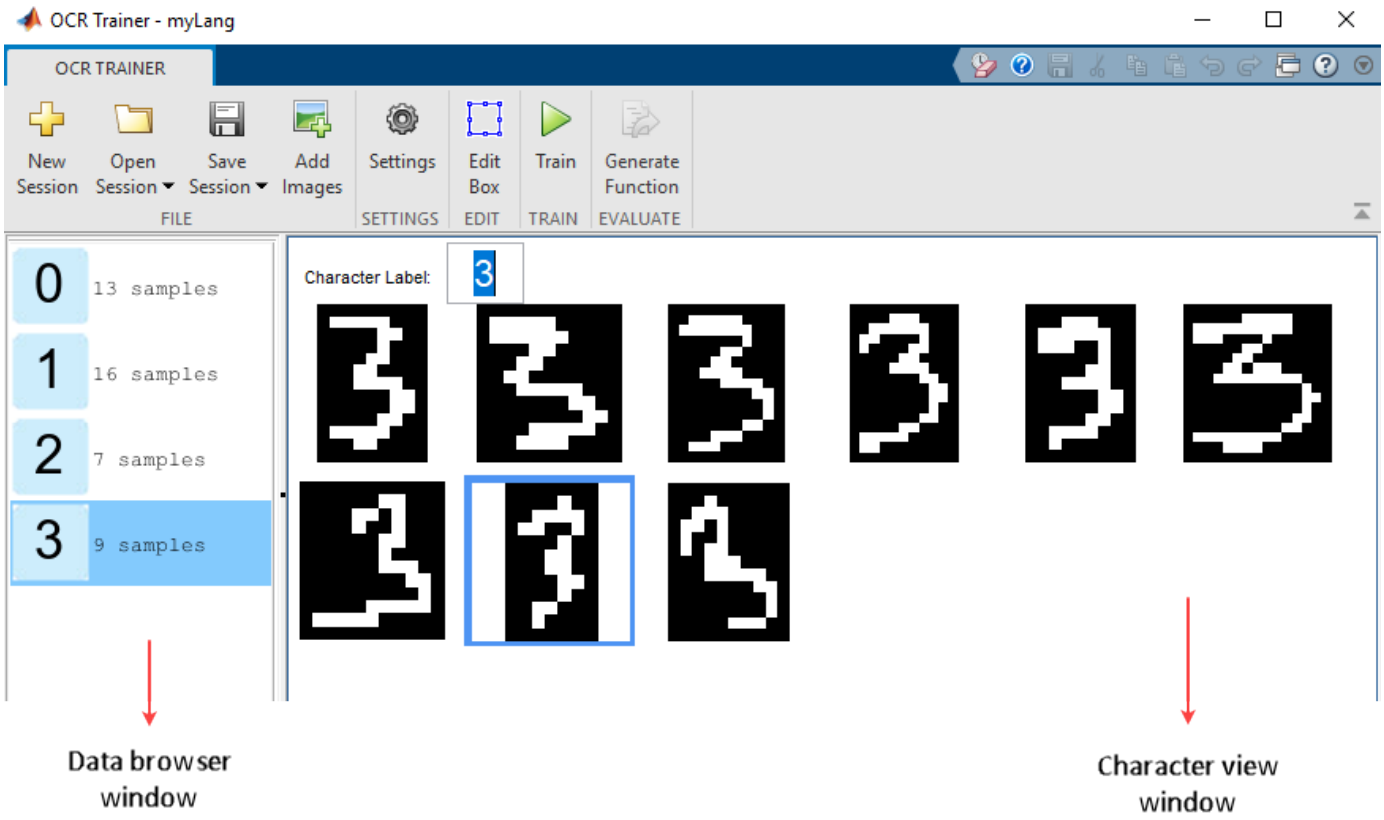
```
I = zeros(500,500,3,'uint8');

textLines = [
    "some training text"
    "even more stuff to learn"
    ]
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

`ocr` | `visionSupportPackages` | `ocrText` | `graythresh` | `imbinarize` | `imtophat`

More About

- “Install Computer Vision Toolbox Add-on Support Files” on page 13-2

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
`img`	Input	• Binary, grayscale, or truecolor image. • The input image is from the image set that was originally passed into `bagOfFeatures`.

Argument	Input/Output	Description
`features`	Output	• A `binaryFeatures` object. • An *M*-by-*N* numeric matrix of image features, where *M* is the number of features and *N* is the length of each feature vector. • The feature length, *N*, must be greater than zero and be the same for all images processed during the `bagOfFeatures` creation process. • If you cannot extract features from an image, supply an empty feature matrix and an empty feature metrics vector. Use the empty matrix and vector if, for example, you did not find any keypoints for feature extraction. • Numeric, real, and nonsparse.
`featureMetrics`	Output	• An *M*-by-1 vector of feature metrics indicating the strength of each feature vector. • Used to apply the `'SelectStrongest'` criteria in `bagOfFeatures` framework. • Numeric, real, and nonsparse.
`location`	Output	• An *M*-by-2 matrix of 1-based [*x* *y*] values. • The [*x* *y*] values can be fractional. • Numeric, real, and nonsparse.

Preprocess the image

Input images can require preprocessing before feature extraction. To extract SURF features and to use the `detectSURFFeatures` or `detectMSERFeatures` functions, the images must be grayscale. If the images are not grayscale, you can convert them using the `im2gray` function.

```
grayImage = im2gray(I);
```

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;
```

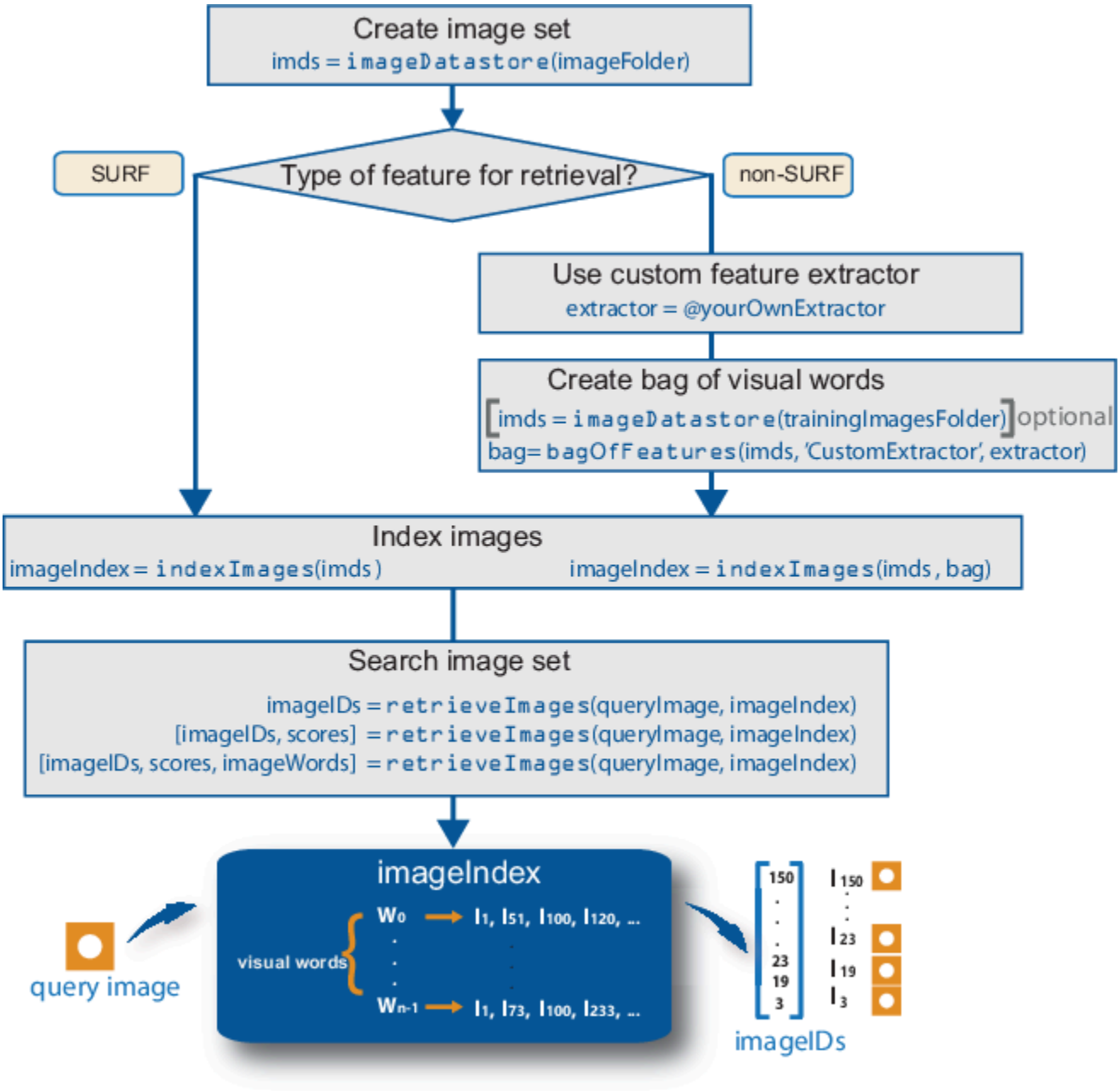
You can optionally return the feature location information. The feature location can be used for spatial or geometric verification image search applications. See the “Geometric Verification Using estimateGeometricTransform2D Function” example. The `retrieveImages` and `indexImages` functions are used for content-based image retrieval systems.

```
if nargout > 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 3-291

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');
```

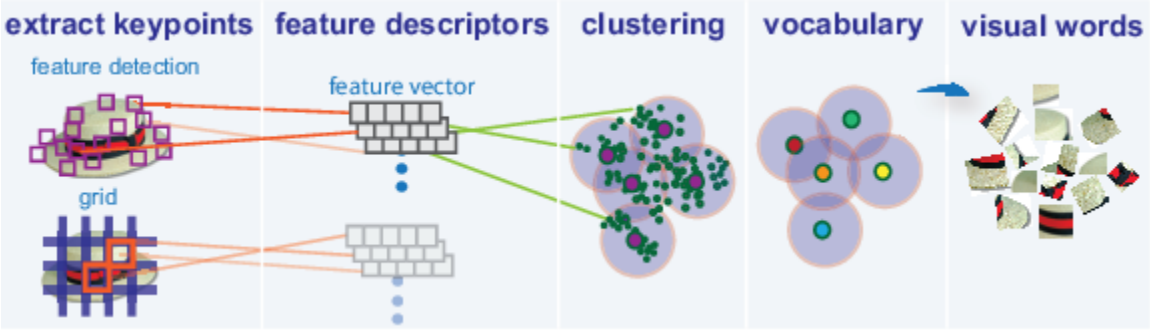
Step 2: Create Bag of Features

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.



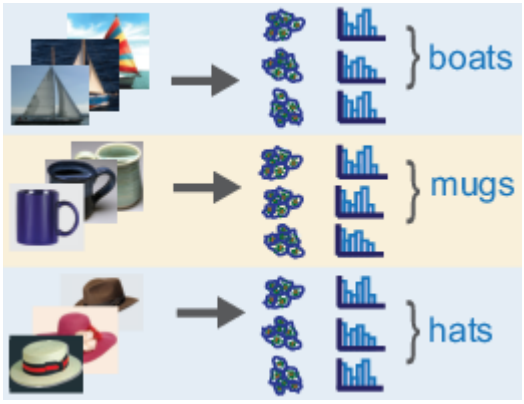
This algorithm workflow analyzes images in their entirety. Images must have appropriate labels describing the class that they represent. For example, a set of car images could be labeled cars. The workflow does not rely on spatial information nor on marking the particular objects in an image. The bag-of-visual-words technique relies on detection without localization.

Step 3: Train an Image Classifier With Bag of Visual Words

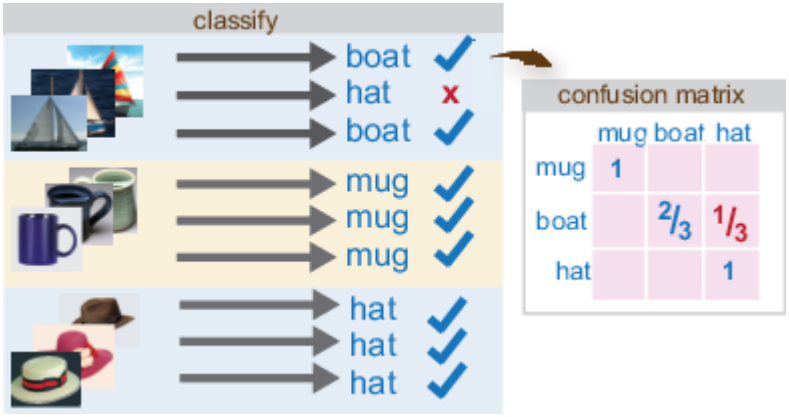
The `trainImageCategoryClassifier` function returns an image classifier. The function trains a multiclass classifier using the error-correcting output codes (ECOC) framework with binary support vector machine (SVM) classifiers. The `trainImageCategoryClassfier` function uses the bag of visual words returned by the `bagOfFeatures` object to encode images in the image set into the histogram of visual words. The histogram of visual words are then used as the positive and negative samples to train the classifier.

1. Use the `bagOfFeatures` `encode` method to encode each image from the training set. This function detects and extracts features from the image and then uses the approximate nearest neighbor algorithm to construct a feature histogram for each image. The function then increments histogram bins based on the proximity of the descriptor to a particular cluster center. The histogram length corresponds to the number of visual words that the `bagOfFeatures` object constructed. The histogram becomes a feature vector for the image.

2. Repeat step 1 for each image in the training set to create the training data.



3 Evaluate the quality of the classifier. Use the `imageCategoryClassifier` `evaluate` method to test the classifier against the validation image set. The output confusion matrix represents the analysis of the prediction. A perfect classification results in a normalized matrix containing 1s on the diagonal. An incorrect classification results fractional values.



Step 4: Classify an Image or Image Set

Use the `imageCategoryClassifier` `predict` method on a new image to determine its category.

References

[1] Csurka, G., C. R. Dance, L. Fan, J. Willamowski, and C. Bray. *Visual Categorization with Bags of Keypoints*. Workshop on Statistical Learning in Computer Vision. ECCV 1 (1–22), 1–2.

See Also

Related Examples

- “Image Category Classification Using Bag of Features” on page 3-275
- “Image Retrieval Using Customized Bag of Features” on page 3-291

Motion Estimation and Tracking

- “Multiple Object Tracking” on page 18-2
- “Video Mosaicking” on page 18-5

Multiple Object Tracking

Tracking is the process of locating a moving object or multiple objects over time in a video stream. Tracking an object is not the same as object detection. Object detection is the process of locating an object of interest in a single frame. Tracking associates detections of an object across multiple frames.

Tracking multiple objects requires detection, prediction, and data association.

- **Detection**: Detect objects of interest in a video frame.
- **Prediction**: Predict the object locations in the next frame.
- **Data association**: Use the predicted locations to associate detections across frames to form *tracks*.

Detection

Selecting the right approach for detecting objects of interest depends on what you want to track and whether the camera is stationary.

Detect Objects Using a Stationary Camera

To detect objects in motion with a stationary camera, you can perform background subtraction using the `vision.ForegroundDetector` System object. The background subtraction approach works efficiently but requires the camera to be stationary.

Detect Objects Using a Moving Camera

To detect objects in motion with a moving camera, you can use a sliding-window detection approach. This approach typically works more slowly than the background subtraction approach. To detect and track a specific category of object, use the System objects or functions described in the table.

Select A Detection Algorithm

Type of Object to Track	Camera	Functionality
Anything that moves	Stationary	`vision.ForegroundDetector` System object™
Faces, eyes, nose, mouth, upper body	Stationary, Moving	`vision.CascadeObjectDetector` System object
Pedestrians	Stationary, Moving	`vision.PeopleDetector` System object
Custom object category	Stationary, Moving	`trainCascadeObjectDetector` function or custom sliding window detector using `extractHOGFeatures` and `selectStrongestBbox`

Prediction

To track an object over time means that you must predict its location in the next frame. The simplest method of prediction is to assume that the object will be near its last known location. In other words, the previous detection serves as the next prediction. This method is especially effective for high

frame rates. However, using this prediction method can fail when objects move at varying speeds, or when the frame rate is low relative to the speed of the object in motion.

A more sophisticated method of prediction is to use the previously observed motion of the object. The Kalman filter (`vision.KalmanFilter`) predicts the next location of an object, assuming that it moves according to a motion model, such as constant velocity or constant acceleration. The Kalman filter also takes into account process noise and measurement noise. Process noise is the deviation of the actual motion of the object from the motion model. Measurement noise is the detection error.

To make configuring a Kalman filter easier, use `configureKalmanFilter`. This function sets up the filter for tracking a physical object moving with constant velocity or constant acceleration within a Cartesian coordinate system. The statistics are the same along all dimensions. If you need to configure a Kalman filter with different assumptions, you need to construct the `vision.KalmanFilter` object directly.

Data Association

Data association is the process of associating detections corresponding to the same physical object across frames. The temporal history of a particular object consists of multiple detections, and is called a *track*. A track representation can include the entire history of the previous locations of the object. Alternatively, it can consist only of the object's last known location and its current velocity.

Detection to Track Cost Functions

To match a detection to a track, you must establish criteria for evaluating the matches. Typically, you establish this criteria by defining a cost function. The higher the cost of matching a detection to a track, the less likely that the detection belongs to the track. A simple cost function can be defined as the degree of overlap between the bounding boxes of the predicted and detected objects. The “Tracking Pedestrians from a Moving Car” on page 8-40 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 than can incorporate information about the object size and appearance.

Elimination of Unlikely Matches

Gating is a method of eliminating highly unlikely matches from consideration, such as by imposing a threshold on the cost function. An observation cannot be matched to a track if the cost exceeds a certain threshold value. Using this threshold method effectively results in a circular *gating region* around each prediction, where a matching detection can be found. An alternative gating technique is to make the gating region large enough to include the *k*-nearest neighbors of the prediction.

Assign Detections to Track

Data association reduces to a minimum weight bipartite matching problem, which is a well-studied area of graph theory. A bipartite graph represents tracks and detections as vertices. It also represents the cost of matching a detection and a track as a weighted edge between the corresponding vertices.

The `assignDetectionsToTracks` function implements the Munkres' variant of the Hungarian bipartite matching algorithm. Its input is the *cost matrix*, where the rows correspond to tracks and the columns correspond to detections. Each entry contains the cost of assigning a particular detection to a particular track. You can implement gating by setting the cost of impossible matches to infinity.

Track Management

Data association must take into account the fact that new objects can appear in the field of view, or that an object being tracked can leave the field of view. In other words, in any given frame, some number of new tracks might need to be created, and some number of existing tracks might need to be discarded. The `assignDetectionsToTracks` function returns the indices of unassigned tracks and unassigned detections in addition to the matched pairs.

One way of handling unmatched detections is to create a new track from each of them. Alternatively, you can create new tracks from unmatched detections greater than a certain size, or from detections that have certain locations or appearance. For example, if the scene has a single entry point, such as a doorway, then you can specify that only unmatched detections located near the entry point can begin new tracks, and that all other detections are considered noise.

Another way of handling unmatched tracks is to delete any track that remain unmatched for a certain number of frames. Alternatively, you can specify to delete an unmatched track when its last known location is near an exit point.

See Also

`assignDetectionsToTracks` | `vision.KalmanFilter` | `bboxOverlapRatio` | `configureKalmanFilter` | `vision.ForegroundDetector` | `vision.PeopleDetector` | `vision.CascadeObjectDetector` | `trainCascadeObjectDetector` | `extractHOGFeatures` | `vision.PointTracker` | `selectStrongestBbox`

Related Examples

- “Tracking Pedestrians from a Moving Car” on page 8-40
- “Use Kalman Filter for Object Tracking” on page 8-50
- “Motion-Based Multiple Object Tracking” on page 8-31

More About

- “Get Started with Cascade Object Detector” on page 17-89

External Websites

- Detect and Track Multiple Faces

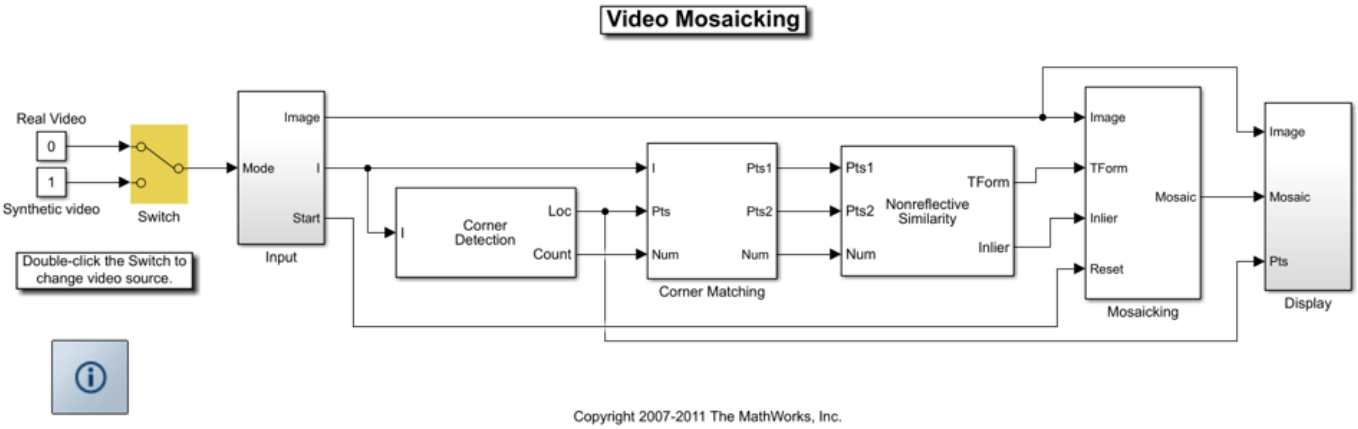
Video Mosaicking

This example shows how to create a mosaic from a video sequence. Video mosaicking is the process of stitching video frames together to form a comprehensive view of the scene. The resulting mosaic image is a compact representation of the video data. The Video Mosaicking block is often used in video compression and surveillance applications.

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

Example Model

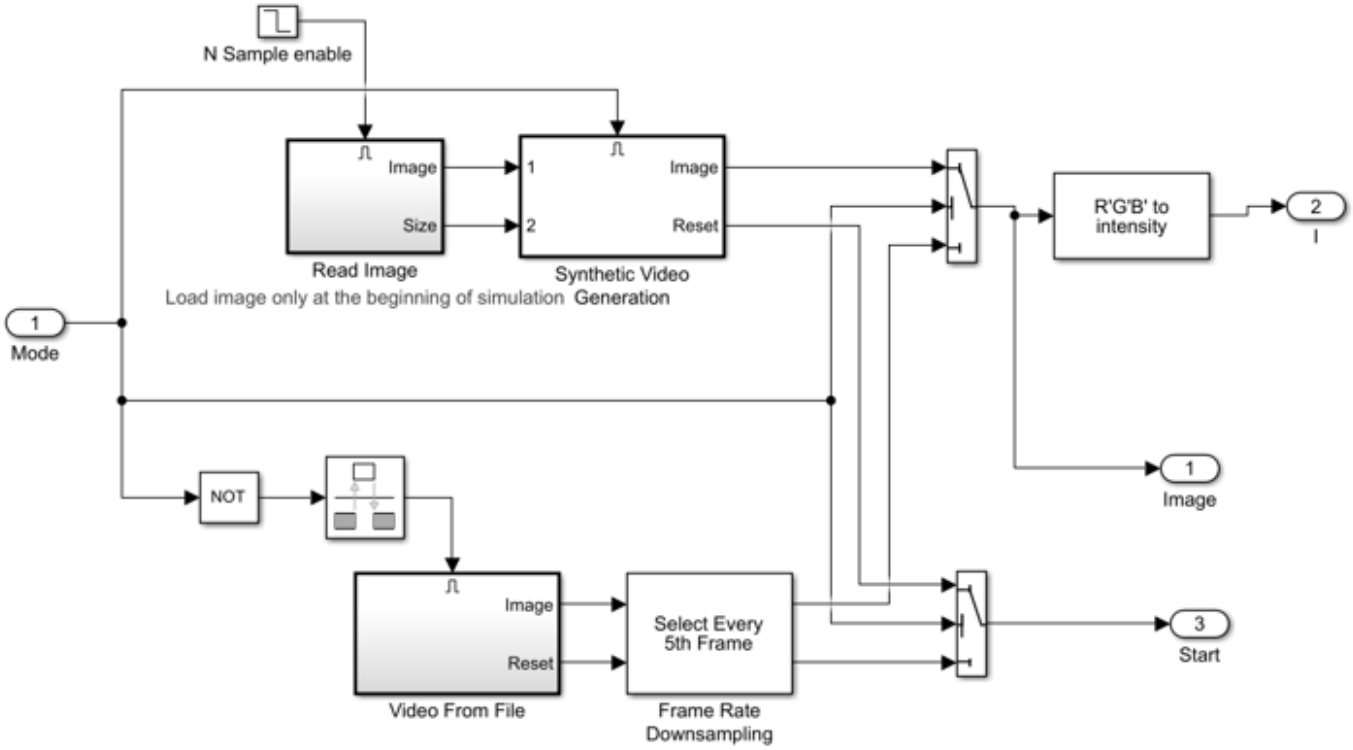
The following figure shows the Video Mosaicking model:



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

Input Subsystem

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

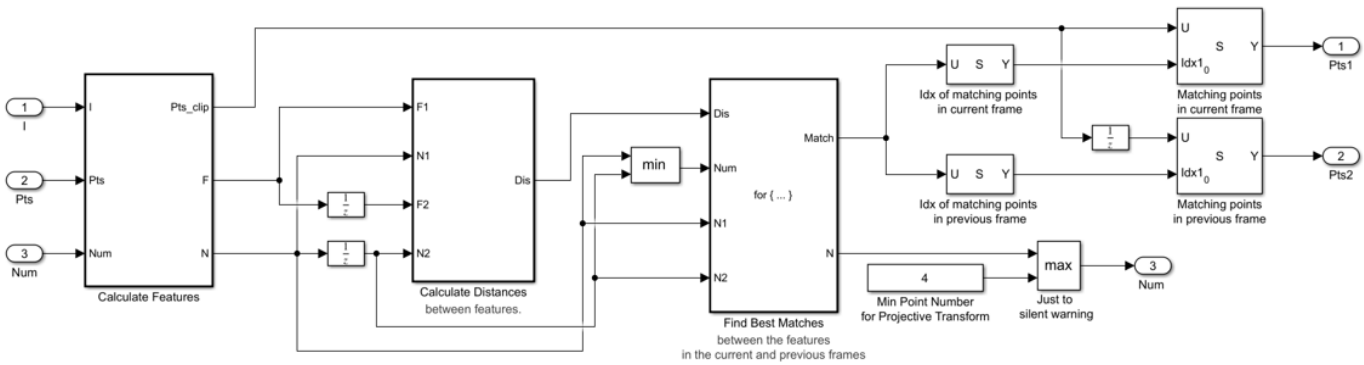


If you choose to use a video sequence from a file, you can reduce computation time by processing only some of the video frames. This is done by setting the downsampling rate in the Frame Rate Downsampling subsystem.

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

Corner Matching Subsystem

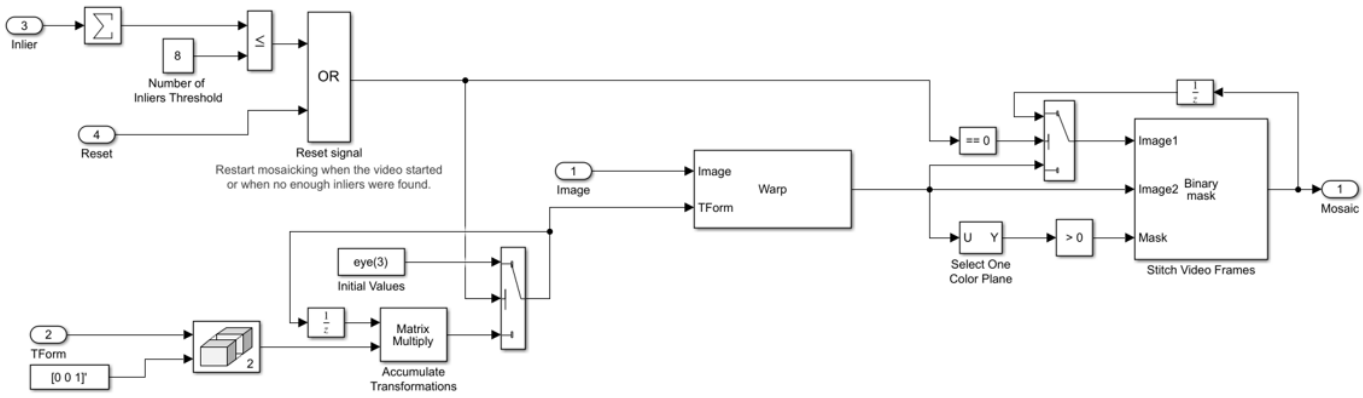
The subsystem finds corner features in the current video frame in one of three methods. The example uses Local intensity comparison (Rosen & Drummond), which is the fastest method. The other methods available are the Harris corner detection (Harris & Stephens) and the Minimum Eigenvalue (Shi & Tomasi).



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

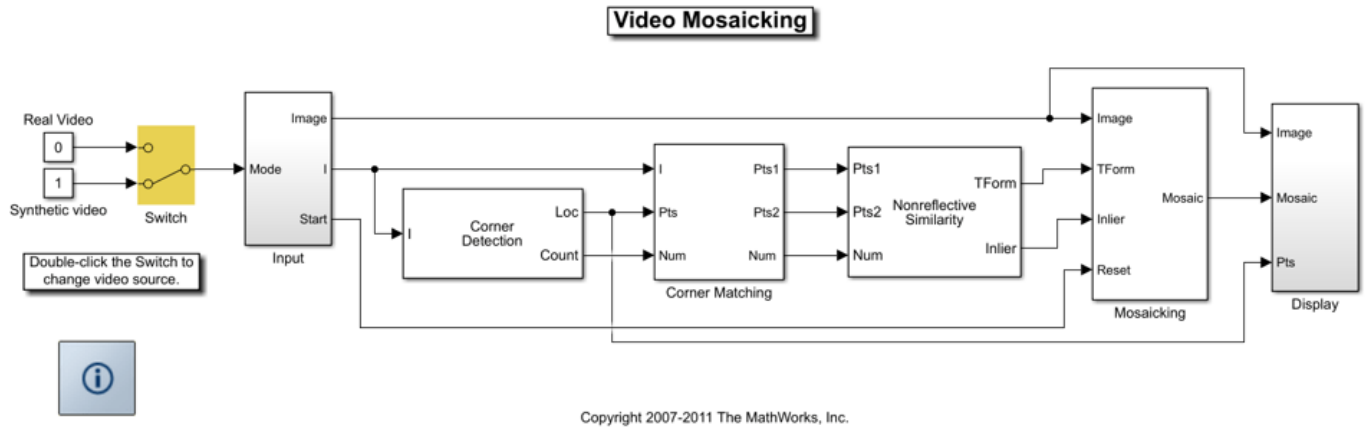
Mosaicking Subsystem

By accumulating transformation matrices between consecutive video frames, the subsystem calculates the transformation matrix between the current and the first video frame. The subsystem then overlays the current video frame on to the output image. By repeating this process, the subsystem generates a mosaic image.



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

Video Mosaicking Using Synthetic Video

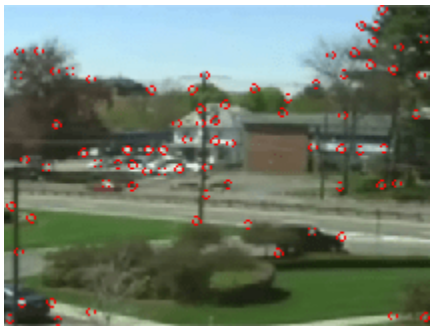


The Corners window shows the corner locations in the current video frame.

The Mosaic window shows the resulting mosaic image.

Video Mosaicking Using Captured Video

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



The Mosaic window shows the resulting mosaic image.

Filters, Transforms, and Enhancements

- “Adjust the Contrast of Intensity Images” on page 19-2
- “Adjust the Contrast of Color Images” on page 19-6
- “Remove Salt and Pepper Noise from Images” on page 19-10
- “Sharpen an Image” on page 19-14

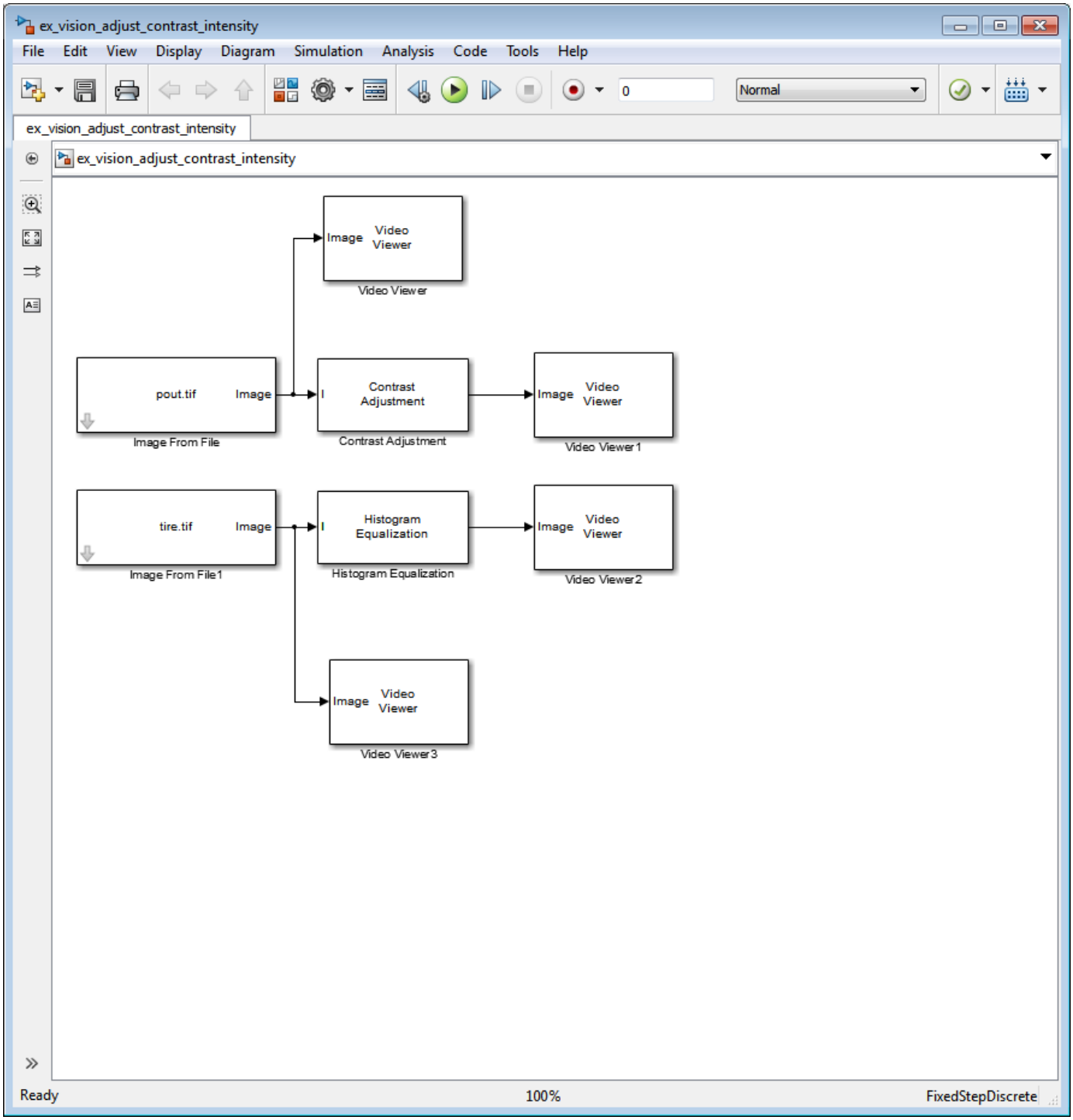
Adjust the Contrast of Intensity Images

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.

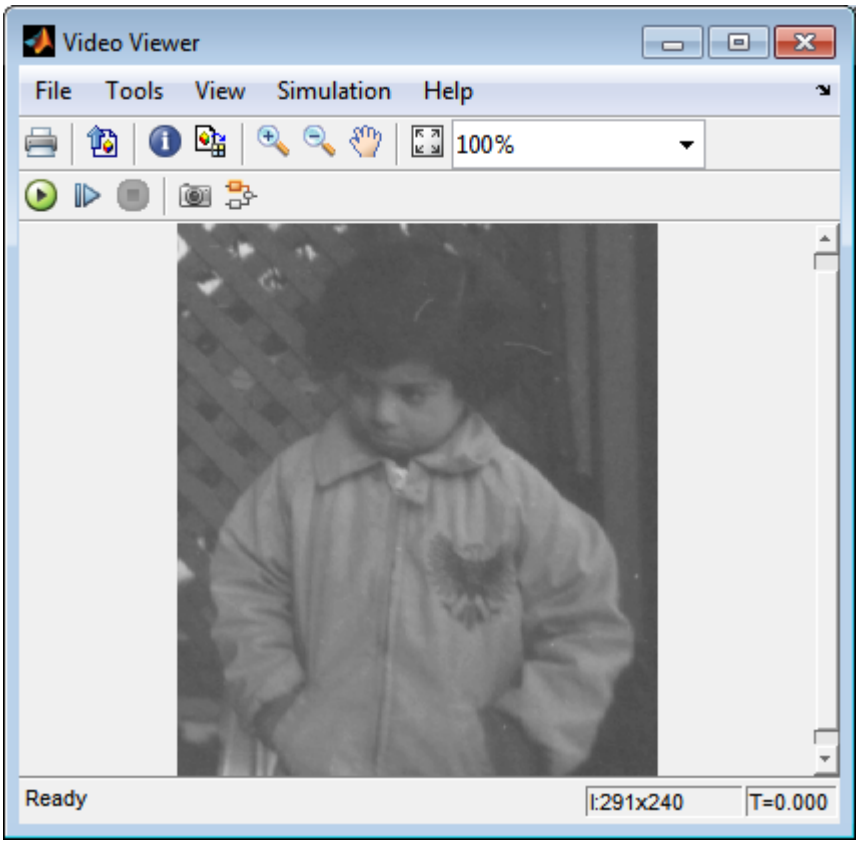


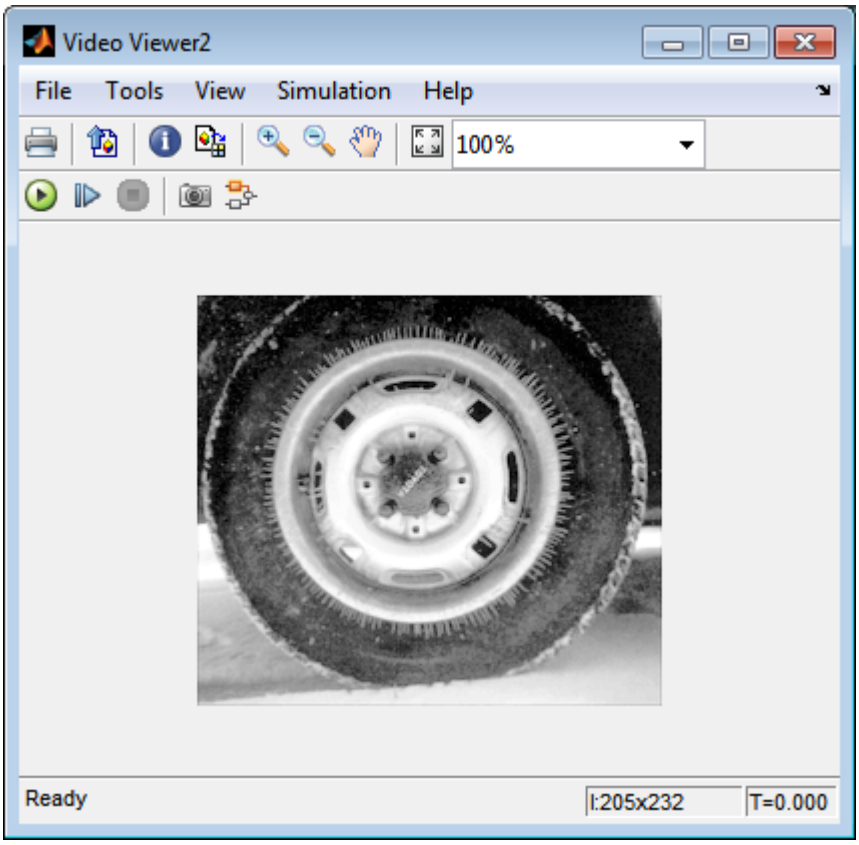
9 Set the configuration parameters. Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings** > **Model Settings**. Set the parameters as follows:

- **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`

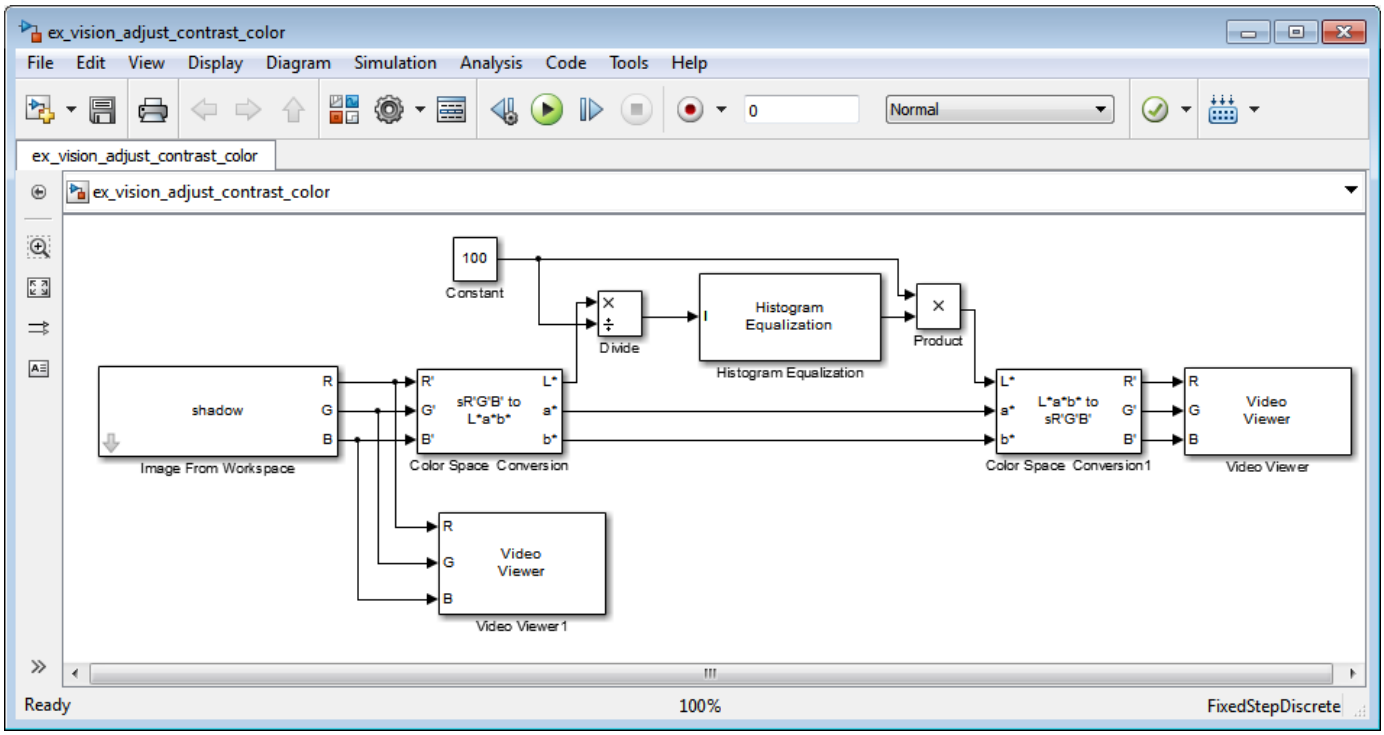
   Because the range of the L* values is between `0` and `100`, you must normalize them to be between zero and one before you pass them to the Histogram Equalization block, which expects floating point input in this range.
6. Use the Constant block to define a normalization factor. Set the **Constant value** 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.

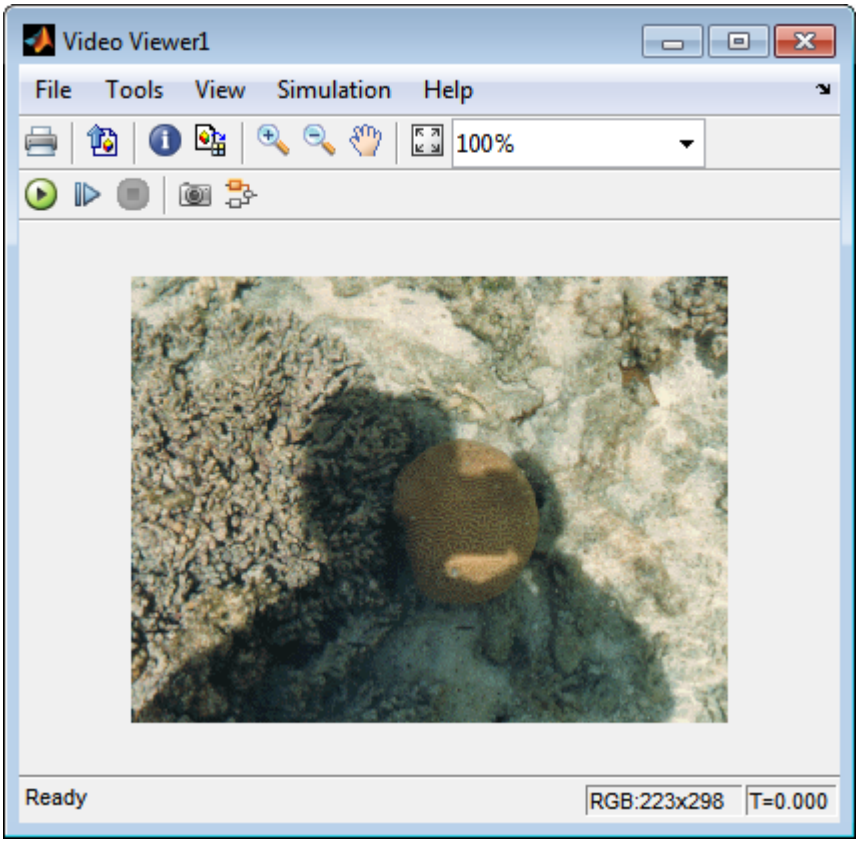


13 Set the configuration parameters. Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings** > **Model Settings**. Set the parameters as follows:

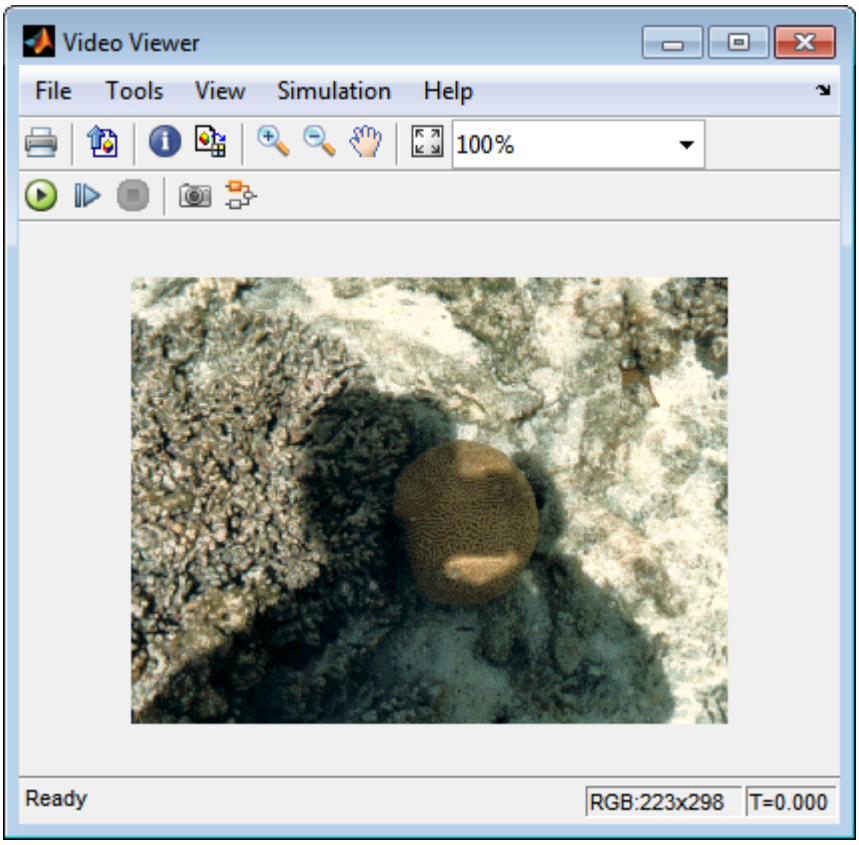
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `Discrete (no continuous states)`

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:

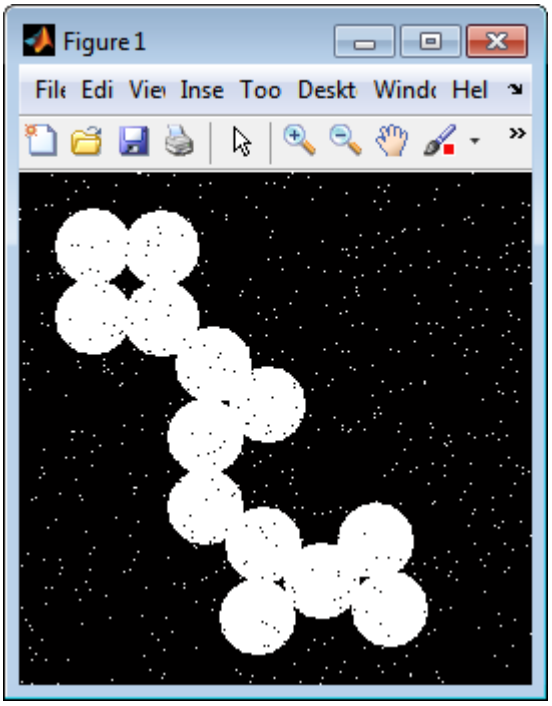
   ```
   I= double(imread('circles.png'));
   I= imnoise(I,'salt & pepper',0.02);
   ```

   `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.

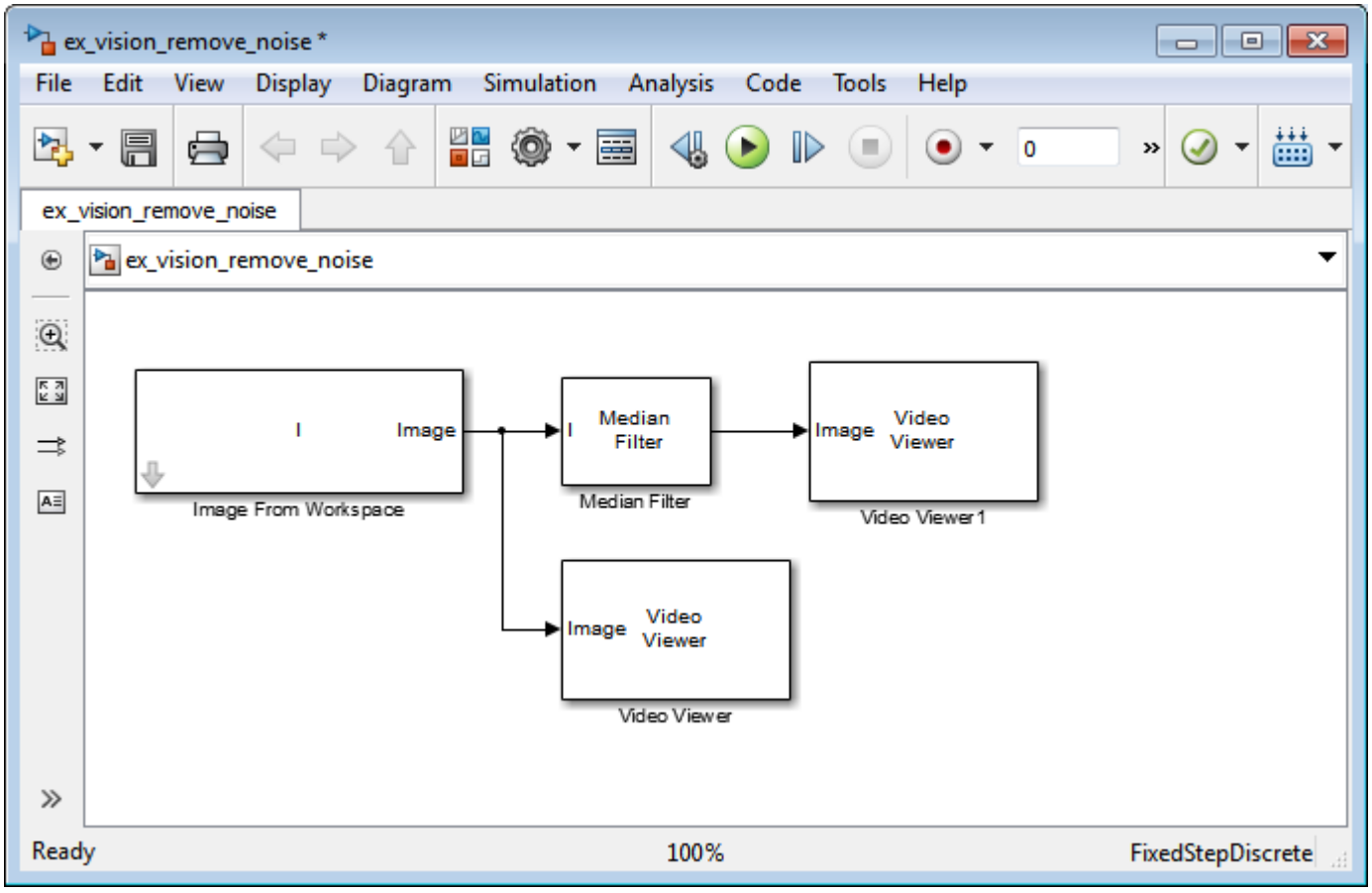
3. Create a Simulink model, and add the blocks shown in the following table.

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

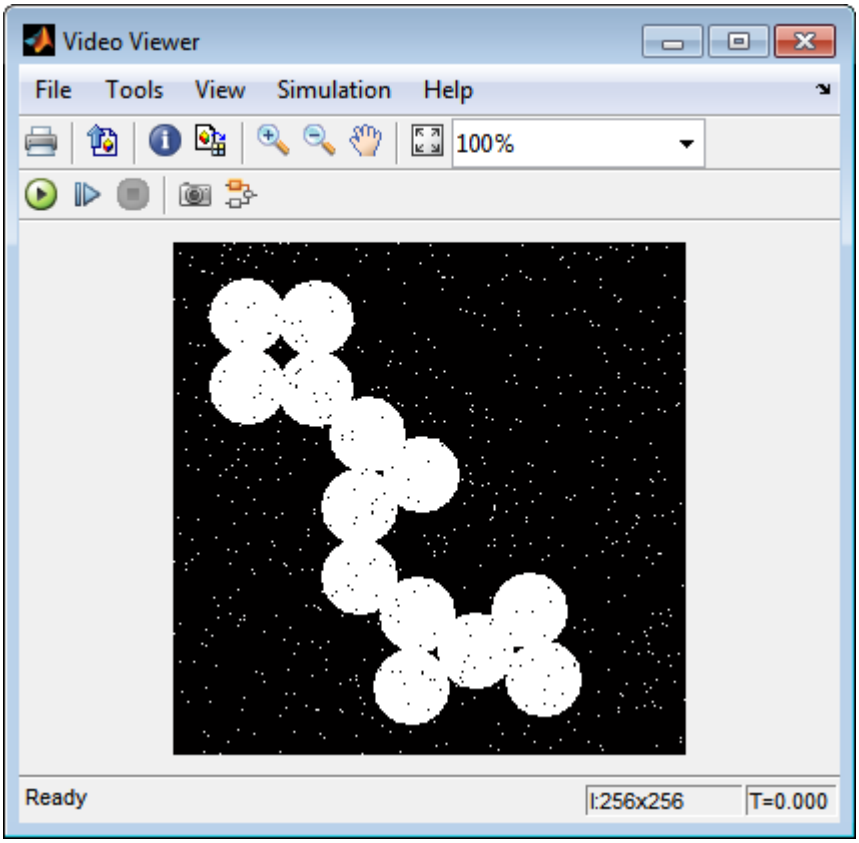
4. Use the Image From Workspace block to import the noisy image into your model. Set the **Value** parameter to `I`.
5. Use the Median Filter block to eliminate the black and white speckles in the image. Use the default parameters.

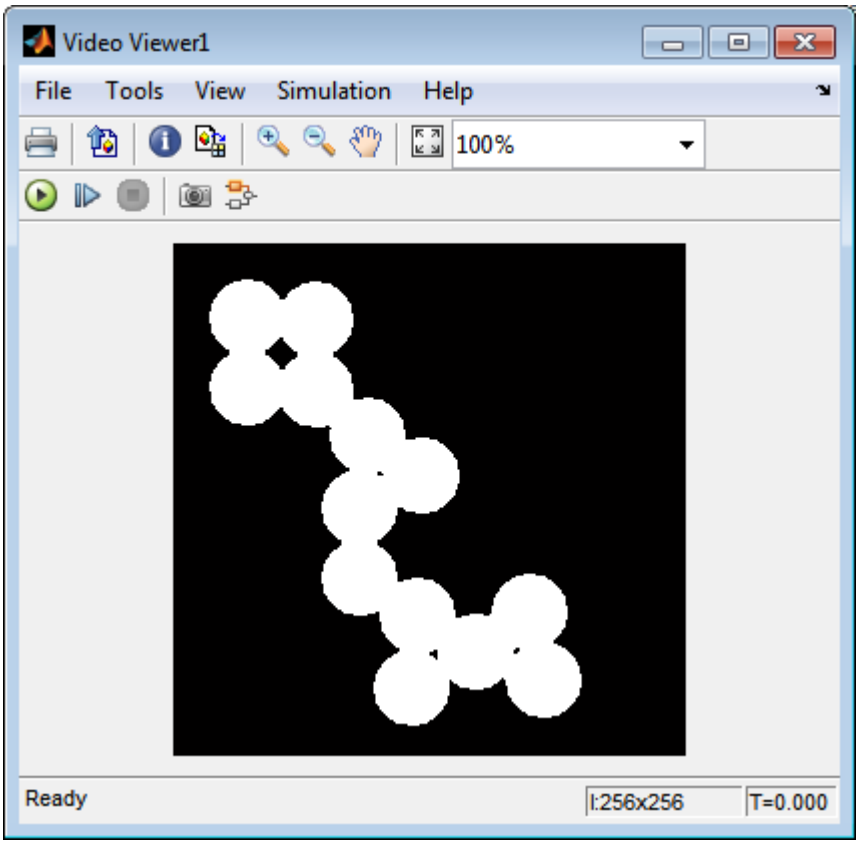
   The Median Filter block replaces the central value of the 3-by-3 neighborhood with the median value of the neighborhood. This process removes the noise in the image.
6. Use the Video Viewer blocks to display the original noisy image, and the modified image. Images are represented by 8-bit unsigned integers. 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

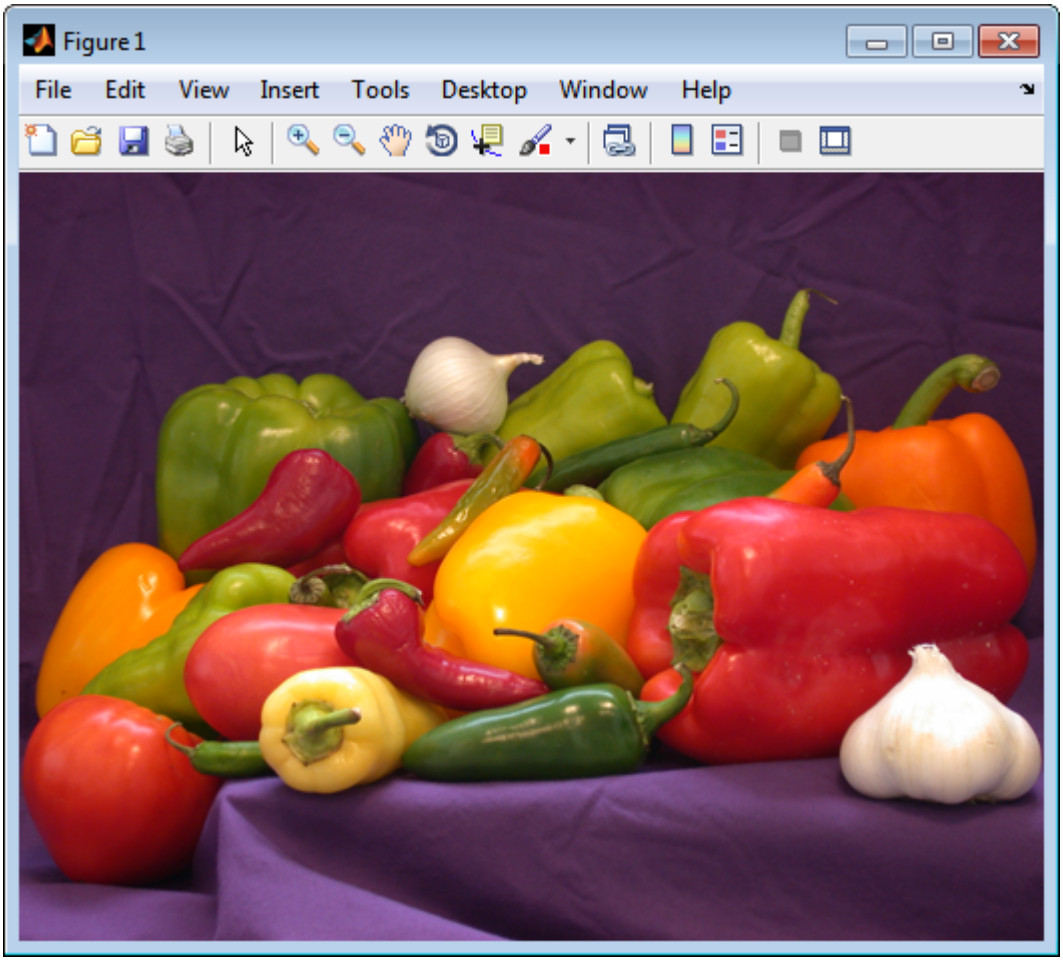
   ```
   I= im2double(imread('peppers.png'));
   ```

   `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`

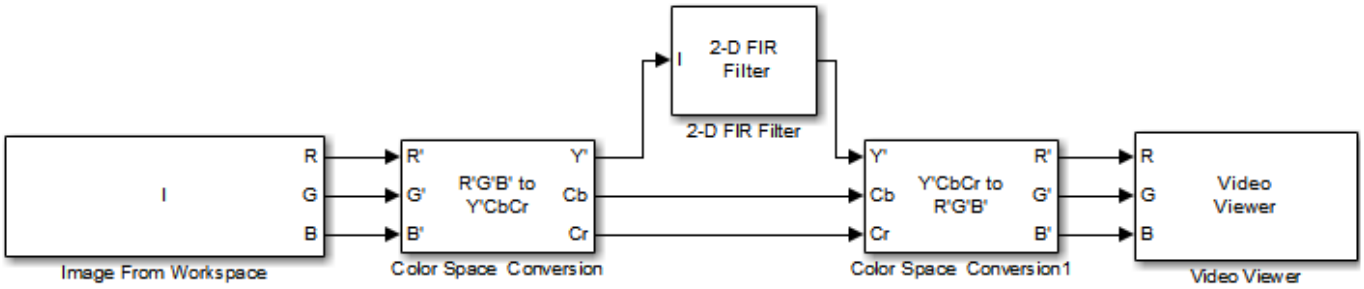
The `fspecial('unsharp')` command creates two-dimensional highpass filter coefficients suitable for correlation. This highpass filter sharpens the image by removing the low frequency noise in it.

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

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

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

9 Connect the blocks as shown in the following figure.

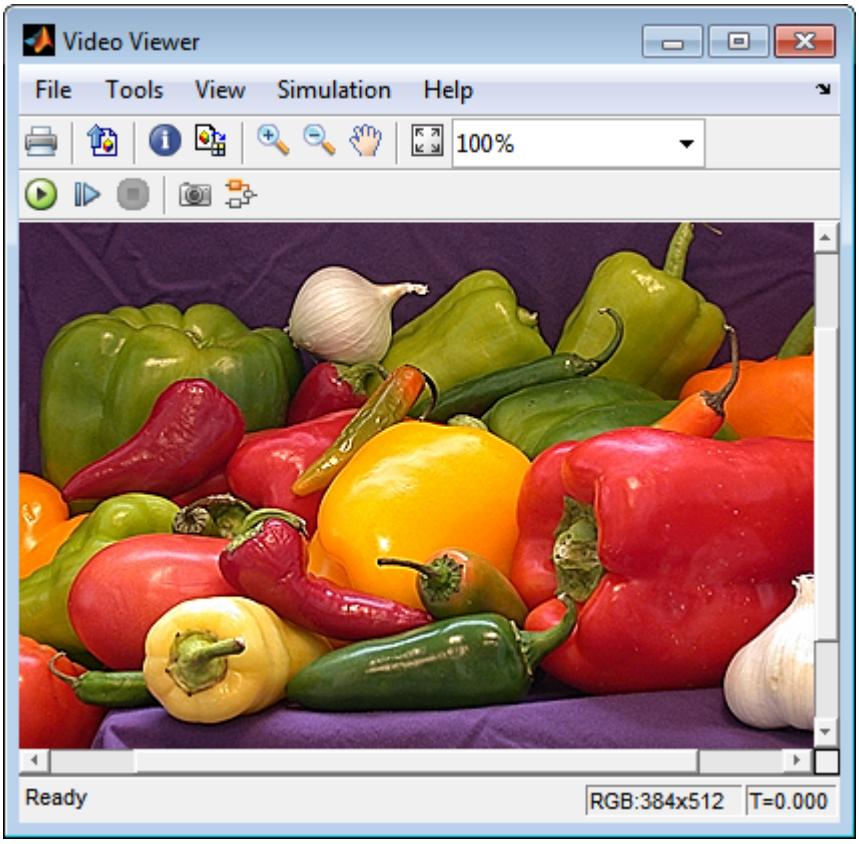


10 Set the configuration parameters. Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings > Model Settings**. Set the parameters as follows:

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

11 Run the model.

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



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

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

Statistics and Morphological Operations

- “Correct Nonuniform Illumination” on page 20-2
- “Count Objects in an Image” on page 20-8

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)`.

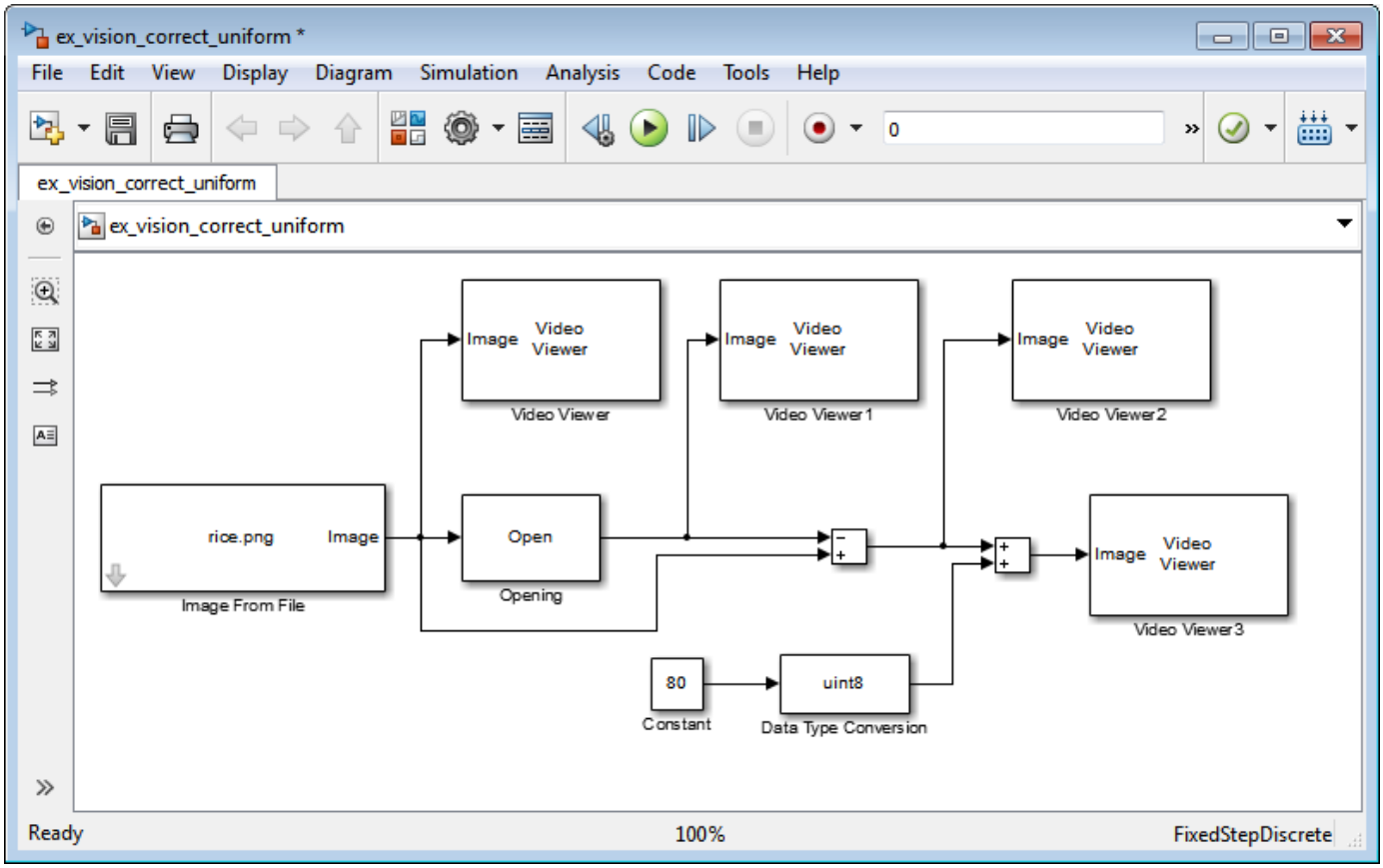
   The `strel` object creates a circular STREL object with a radius of 15 pixels. When working with the Opening block, pick a STREL object that fits within the objects you want to keep. It often takes experimentation to find the neighborhood or STREL object that best suits your application.
5. Use the Video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

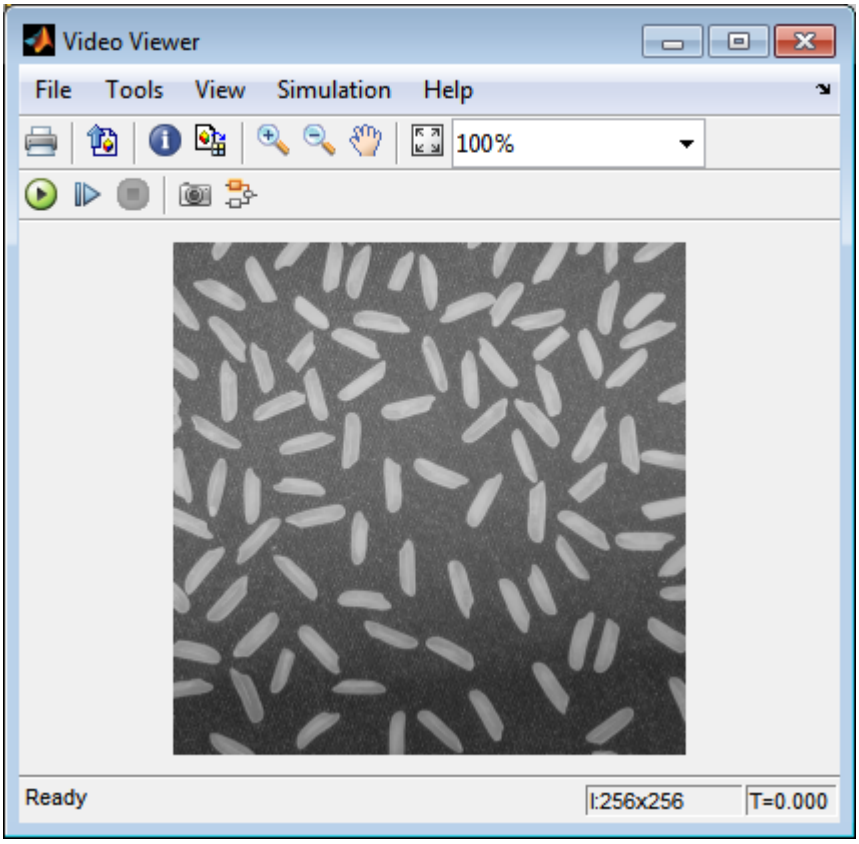


13 Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings** > **Model Settings**. Set the **Solver** parameters as follows:

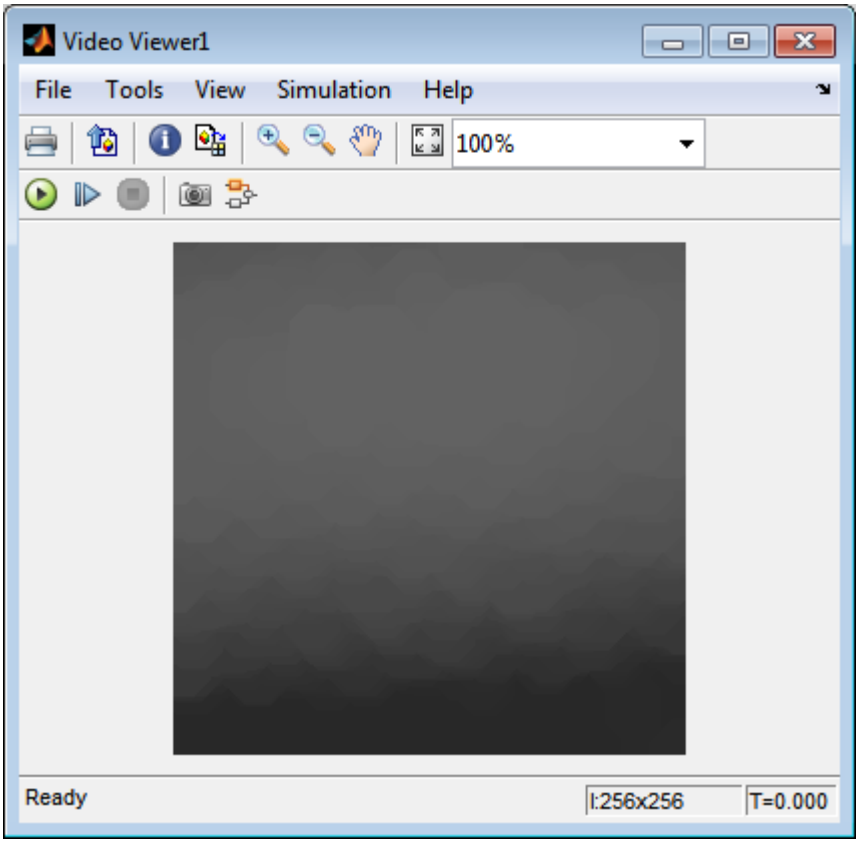
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `discrete (no continuous states)`

14 Run the model.

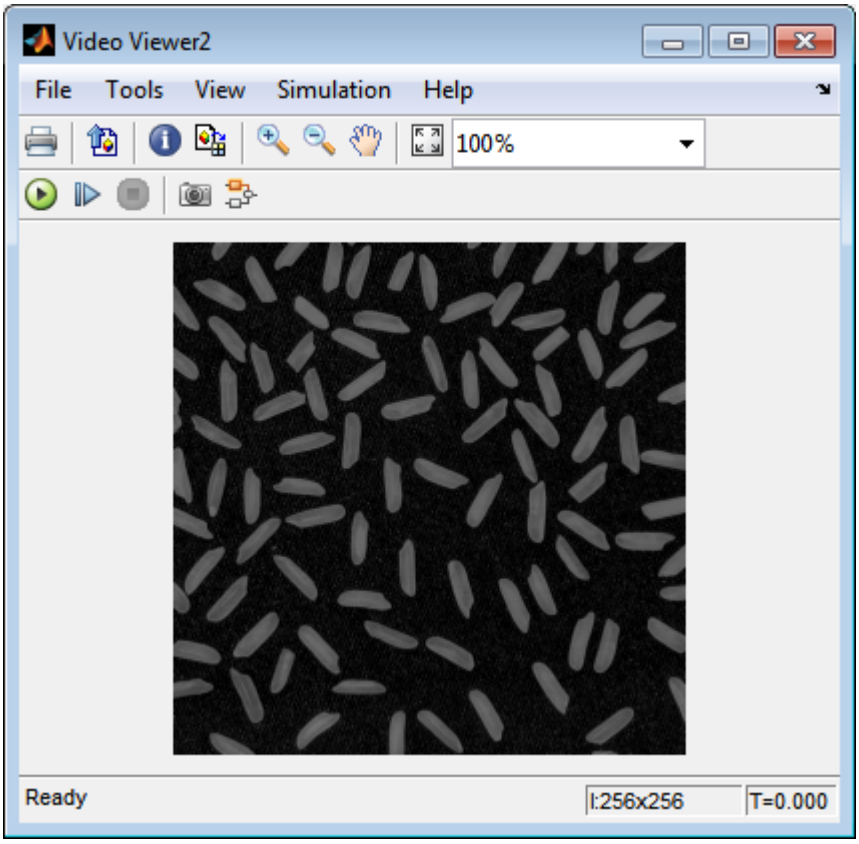
The original image appears in the Video Viewer window.



The estimated background appears in the Video Viewer1 window.

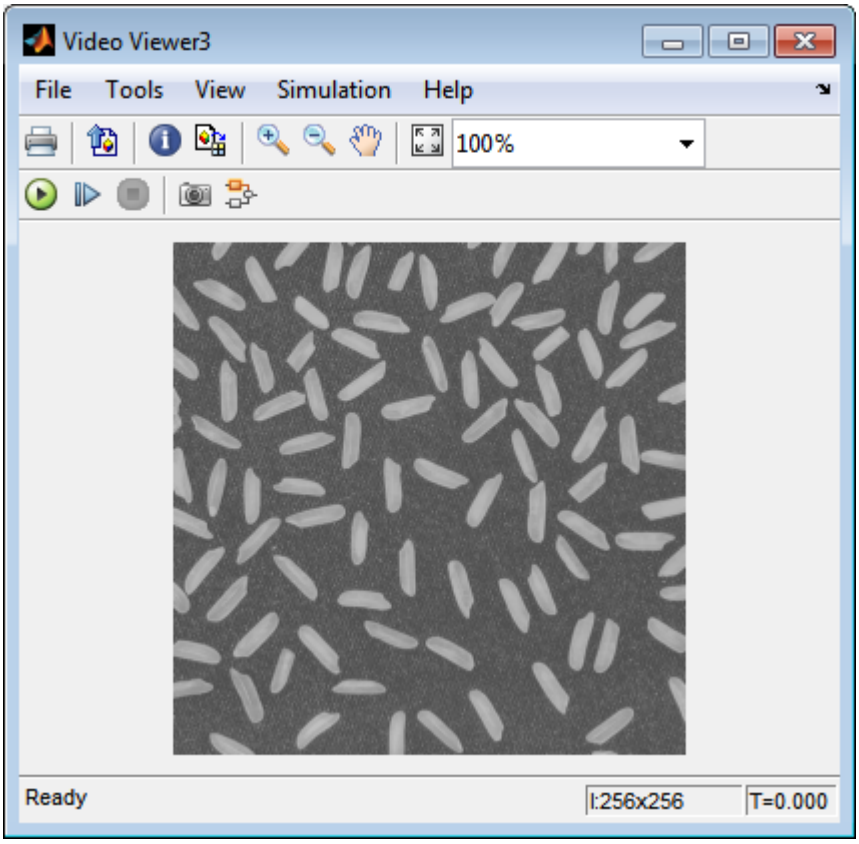


The image without the estimated background appears in the Video Viewer2 window.



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

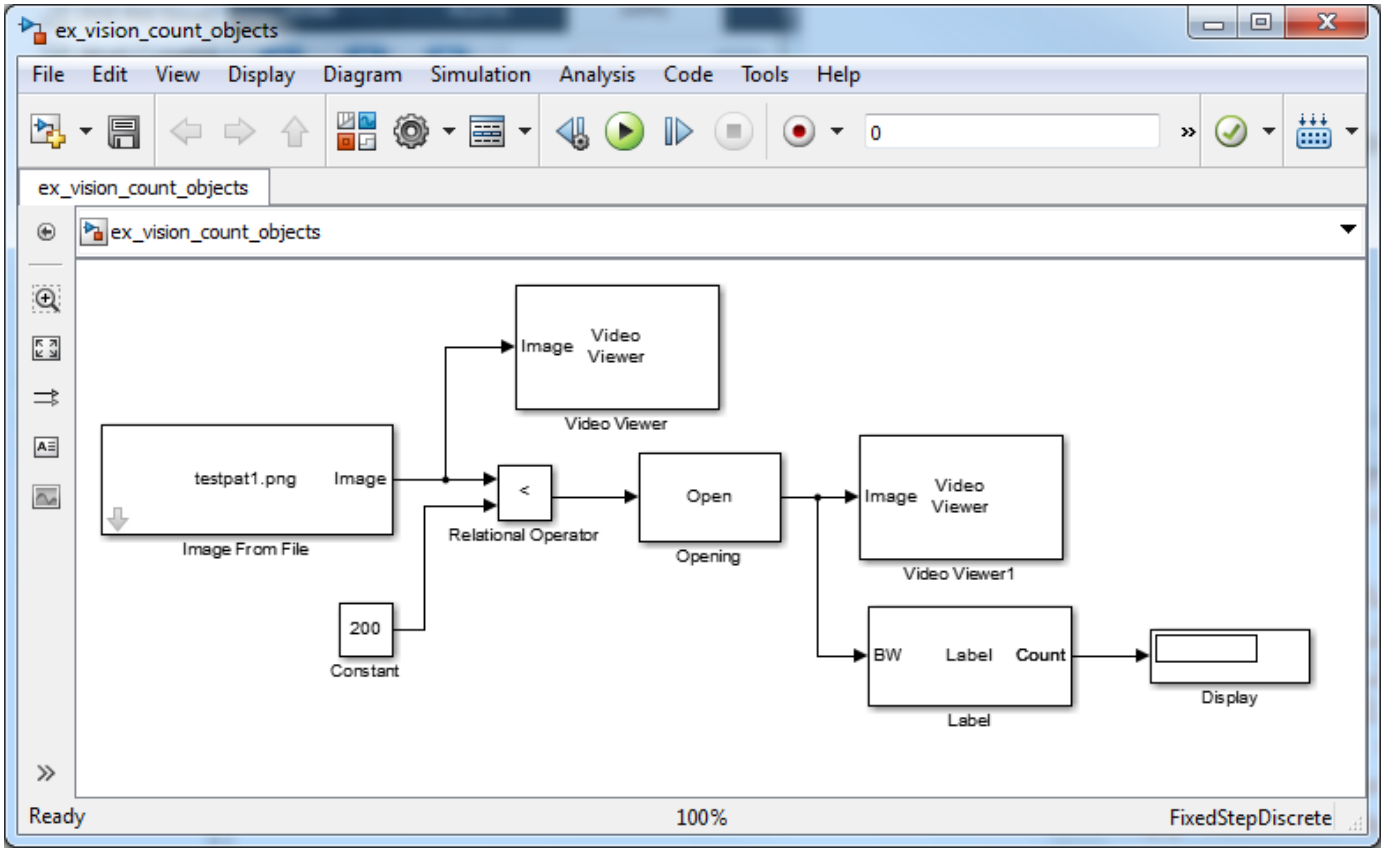
2. Use the Image From File block to import your image. Set the **File name** parameter to `testpat1.png`. This is a 256-by-256 matrix image of 8-bit unsigned integers.
3. Use the Constant block to define a threshold value for the Relational Operator block. Set the **Constant value** parameter to `200`.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

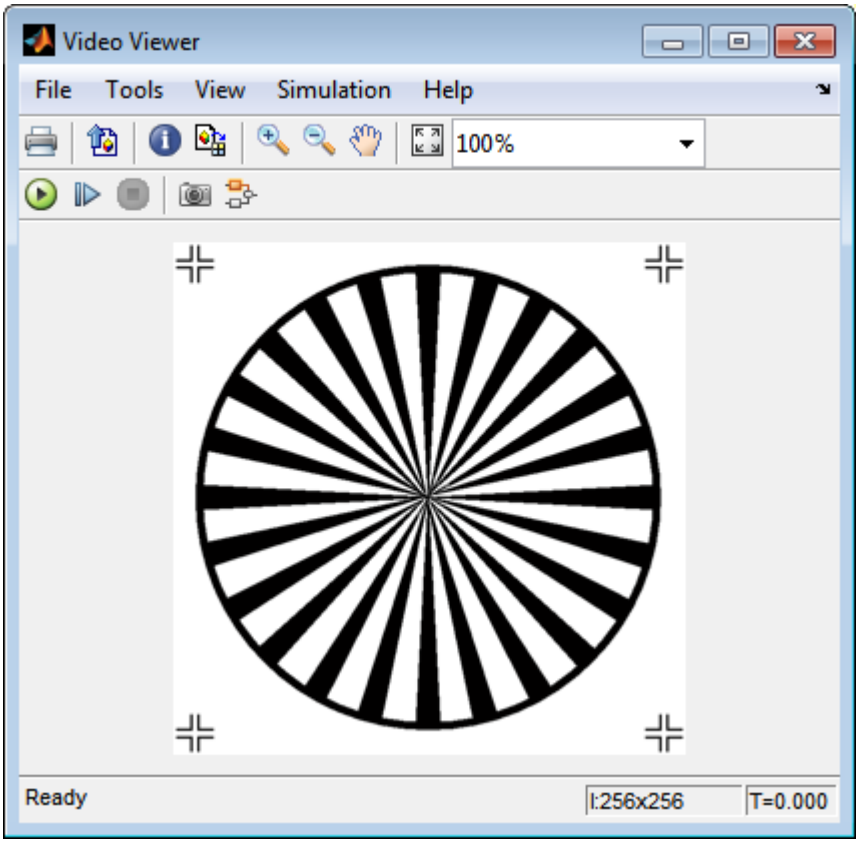


11 Open the Configuration Parameters dialog box from the **Modeling** tab by selecting **Model Settings** > **Model Settings**. Set the **Solver** parameters as follows:

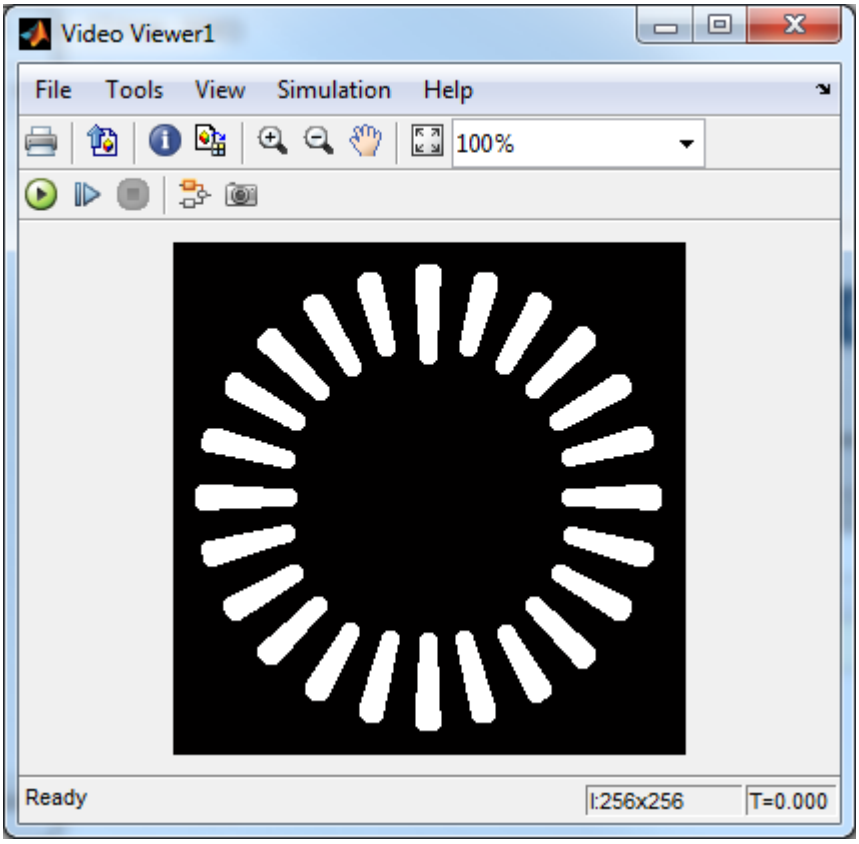
- **Solver** pane, **Stop time** = `0`
- **Solver** pane, **Type** = `Fixed-step`
- **Solver** pane, **Solver** = `discrete (no continuous states)`

12 Run the model.

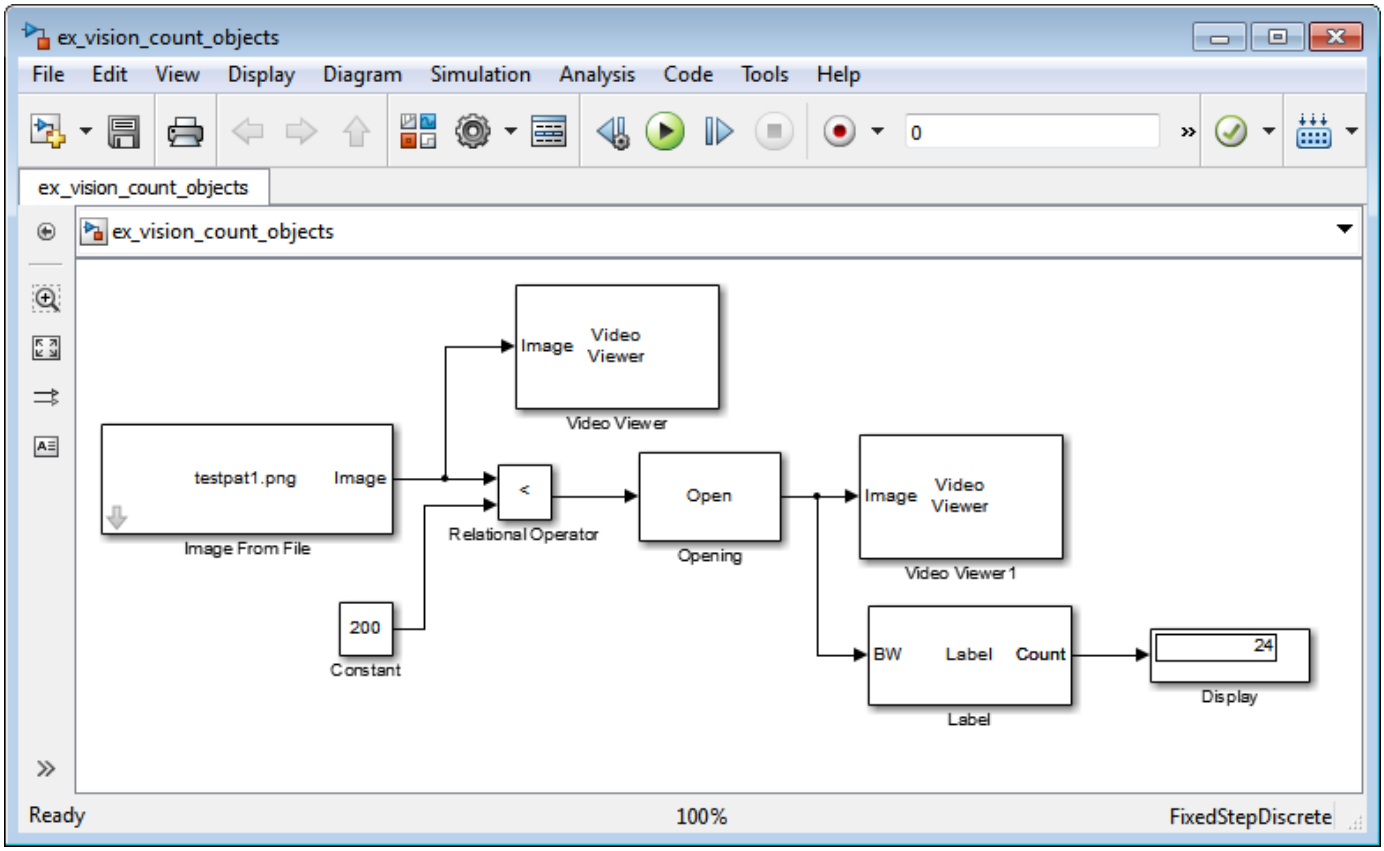
The original image appears in the Video Viewer1 window. To view the image at its true size, right-click the window and select **Set Display To True Size**.



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 21-2
- “Fixed-Point Concepts and Terminology” on page 21-4
- “Arithmetic Operations” on page 21-8
- “Fixed-Point Support for MATLAB System Objects” on page 21-15
- “Specify Fixed-Point Attributes for Blocks” on page 21-16

Fixed-Point Signal Processing

In this section...
“Fixed-Point Features” on page 21-2
“Benefits of Fixed-Point Hardware” on page 21-2
“Benefits of Fixed-Point Design with System Toolboxes Software” on page 21-2

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 21-4 “Scaling” on page 21-5 “Precision and Range” on page 21-6

Fixed-Point Data Types

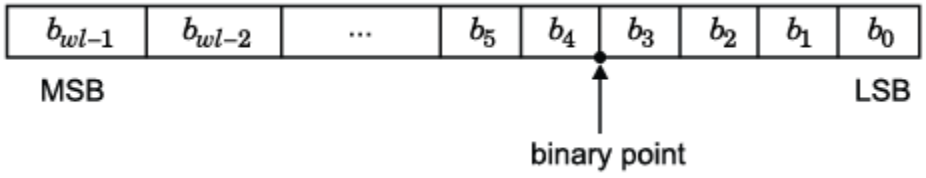
In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). The way hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

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 21-8 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

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

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.

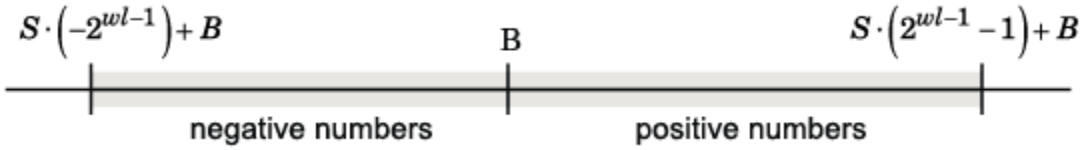
All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

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.

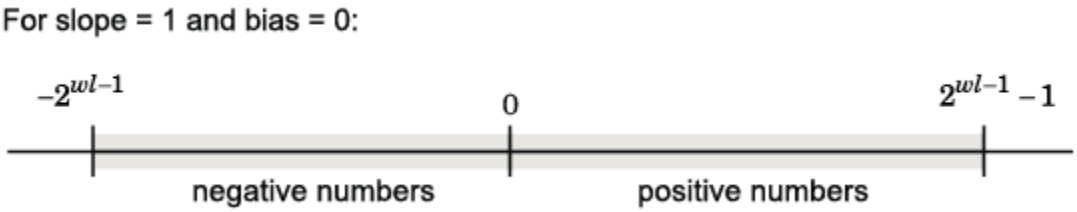
Range

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}:



The full range is the broadest range for a data type. For floating-point types, the full range is –∞ 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 21-8 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

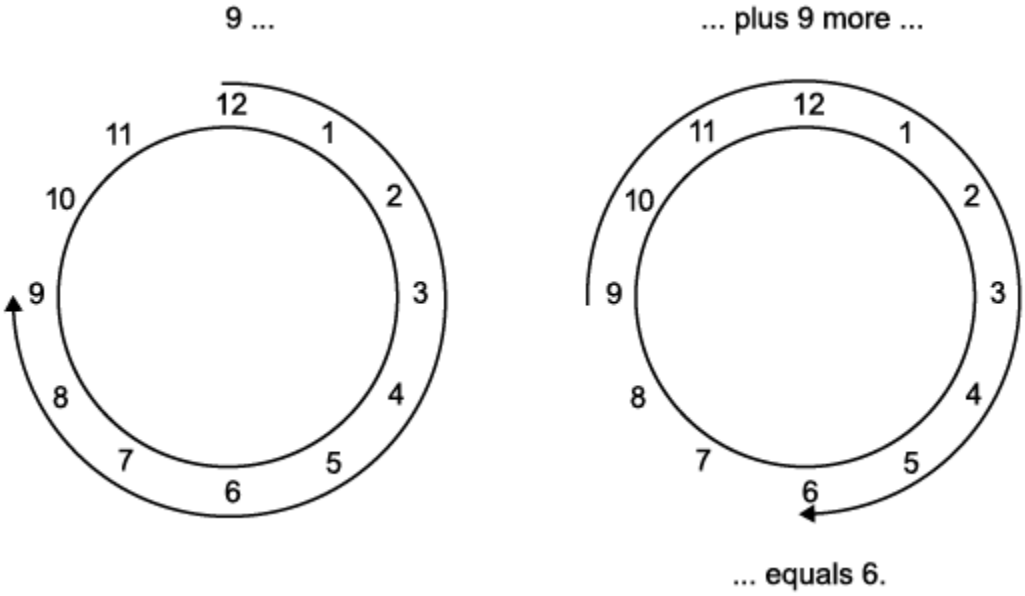
In this section...
“Modulo Arithmetic” on page 21-8
“Two's Complement” on page 21-8
“Addition and Subtraction” on page 21-9
“Multiplication” on page 21-10
“Casts” on page 21-12

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:

```
00101
  +1
00110  (6)
```

Addition and Subtraction

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

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

```
 010010.1     (18.5)
+0110.110     (6.75)
011001.010    (25.25)
```

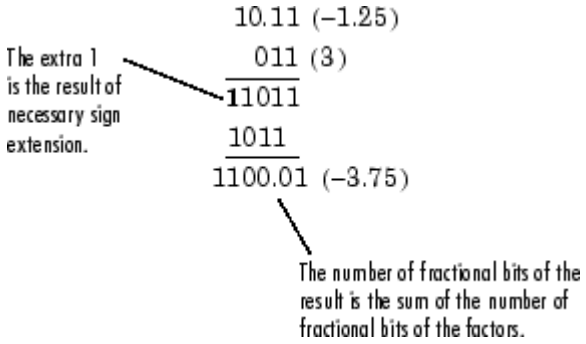
Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

Most fixed-point DSP System Toolbox blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting is necessary during the addition to line up the binary points. See “Casts” on page 21-12 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):



Multiplication Data Types

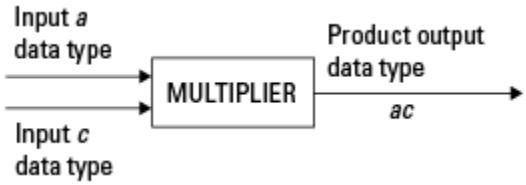
The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages to determine whether a particular 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 21-12.

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.

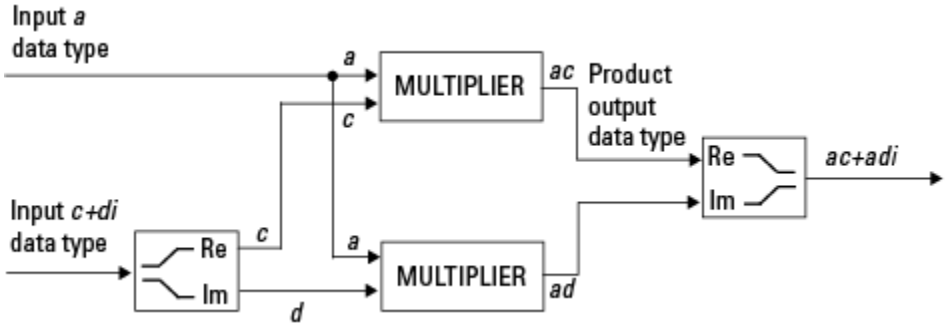
Real-Real Multiplication

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.



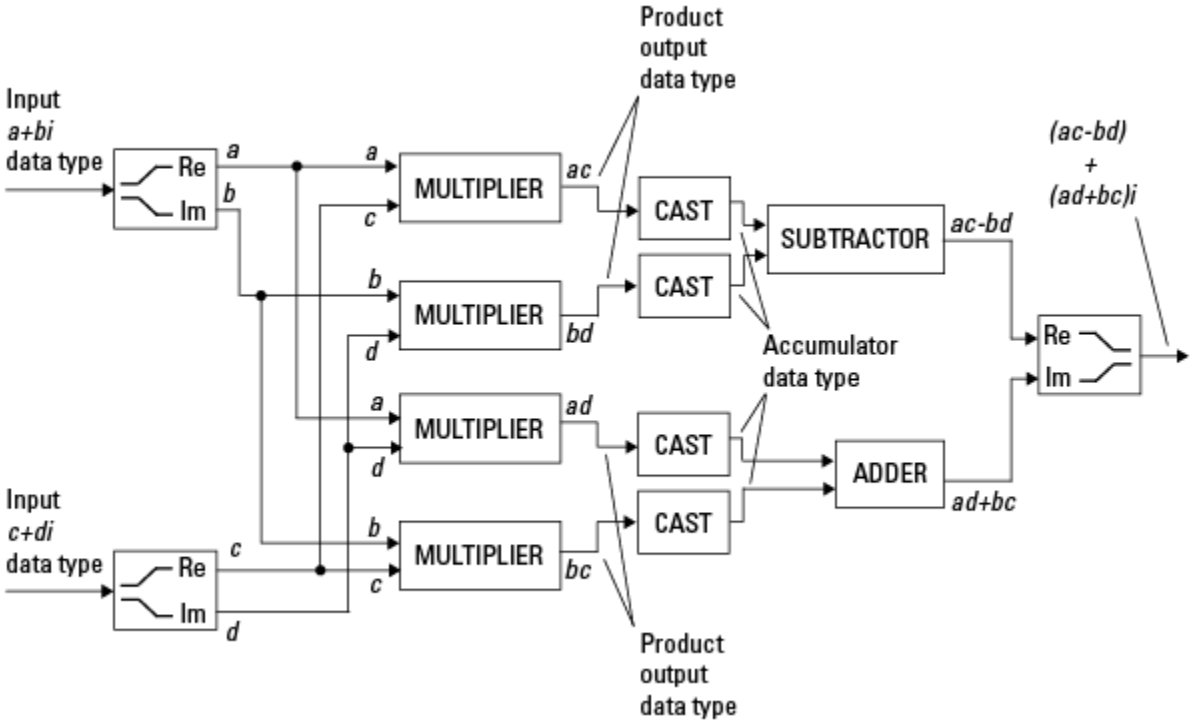
Real-Complex Multiplication

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.

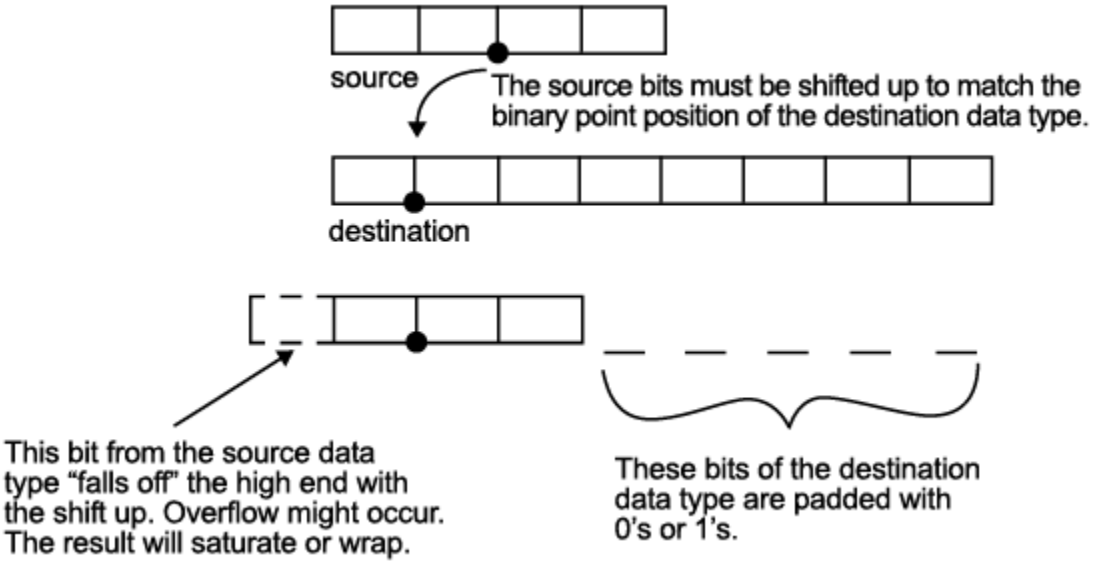
Note that although you cannot mix fixed-point and floating-point signals on the input and output ports of blocks, you can have fixed-point signals with different word and fraction lengths on the ports of blocks that support fixed-point signals.

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:



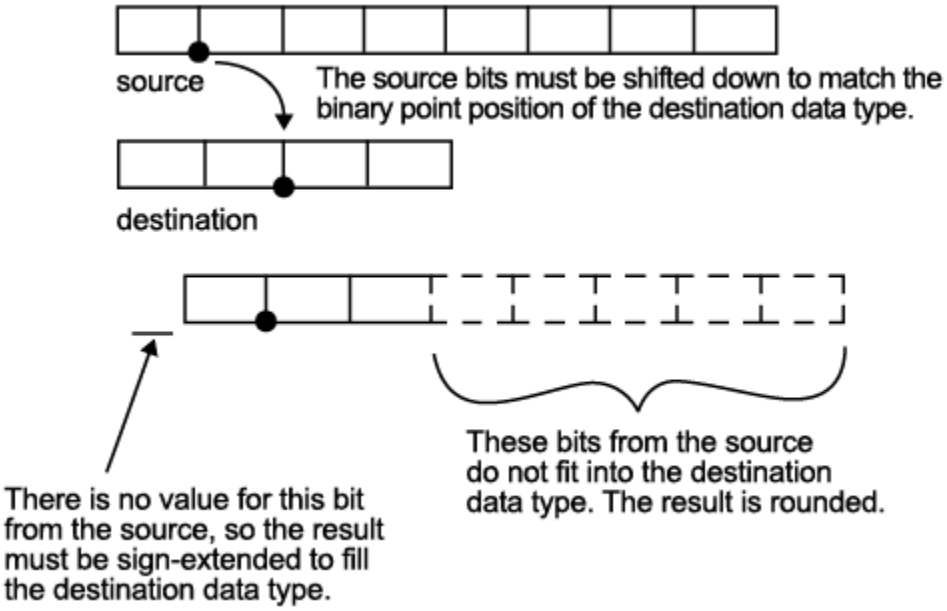
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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 21-15
“Setting System Object Fixed-Point Properties” on page 21-15

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

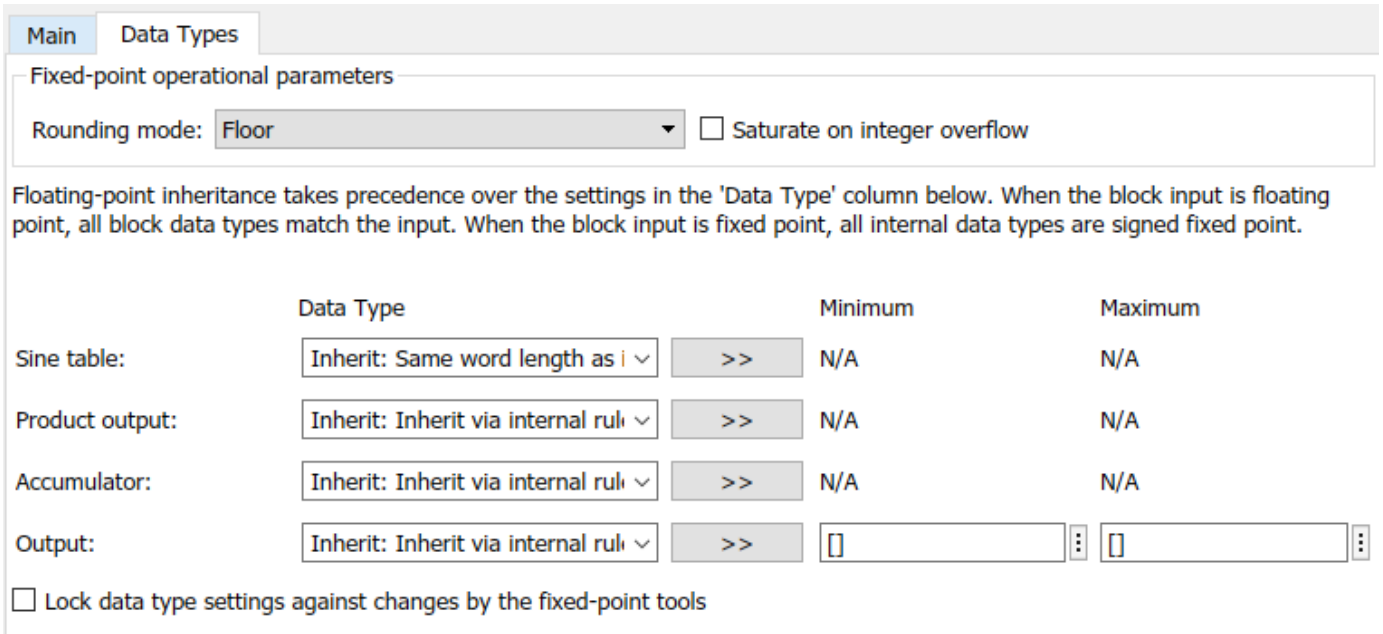
In this section...
“Fixed-Point Block Parameters” on page 21-16 “Specify System-Level Settings” on page 21-18 “Inherit via Internal Rule” on page 21-18 “Specify Data Types for Fixed-Point Blocks” on page 21-25

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.



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	Specifies the rounding mode for the block to use when the specified data type and scaling cannot exactly represent the result of a fixed-point calculation. See “Rounding Modes” on page 21-7 for more information on the available options.
Saturate on integer overflow	When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on saturate and wrap, see “Overflow Handling” on page 21-6 for fixed-point operations.
Intermediate Product	Specifies the data type and scaling of the intermediate product for fixed-point blocks. Blocks that feed multiplication results back to the input of the multiplier use the intermediate product data type. See the reference page of a specific block to learn about the intermediate product data type for that block.
Product Output	Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results. See the reference page of a specific block to learn about the product output data type for that block. For or complex-complex multiplication, the multiplication result is in the accumulator data type. See “Multiplication Data Types” on page 21-10 for more information on complex fixed-point multiplication in toolbox software.
Accumulator	Specifies the data type and scaling of the accumulator (sum) for fixed-point blocks that must hold summation results for further calculation. Most such blocks cast to the accumulator data type before performing the add operations (summation). See the reference page of a specific block for details on the accumulator data type of that block.
Output	Specifies the output data type and scaling for blocks.

Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool available on the **Data Types** pane of some fixed-point 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 21-19
- “Internal Rule for Product Data Types” on page 21-19
- “Internal Rule for Output Data Types” on page 21-19
- “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 21-19
- “Internal Rule Examples” on page 21-20

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 21-19 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 21-19 for more information.

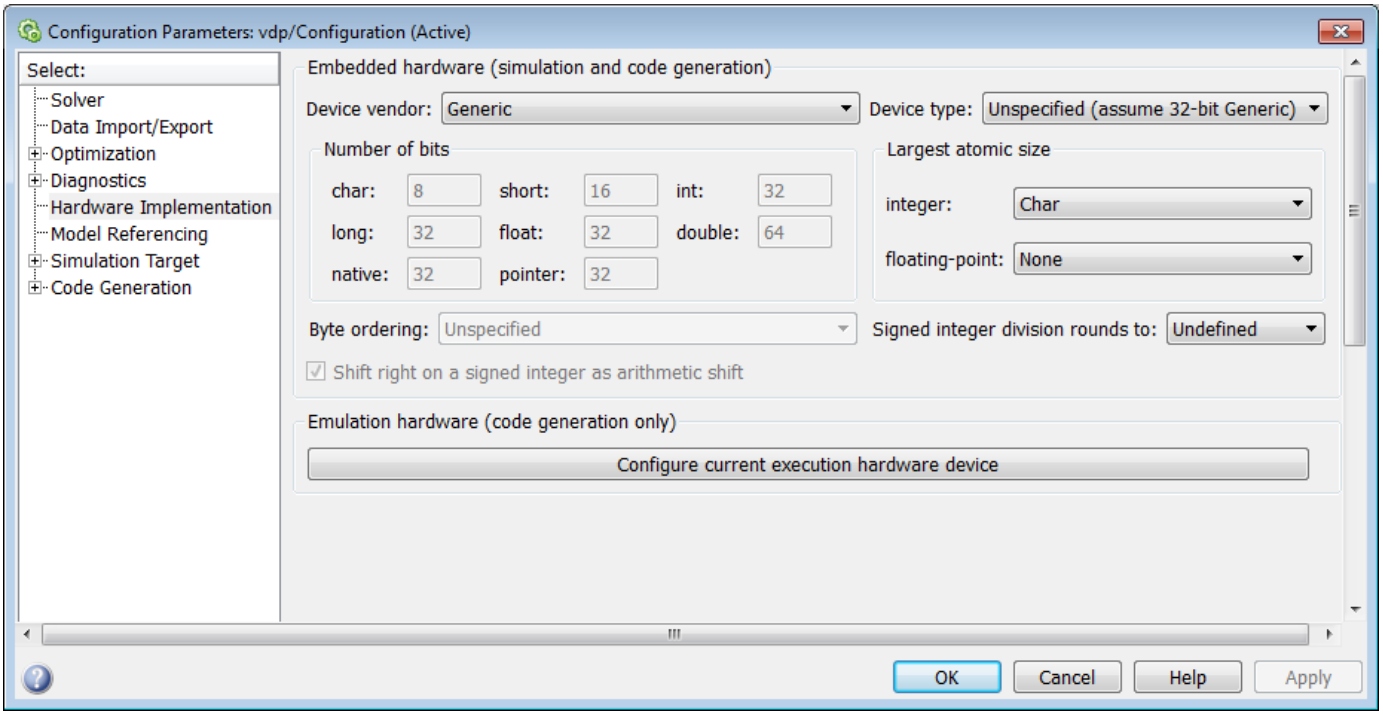
Internal Rule for Output Data Types

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 21-19.

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.

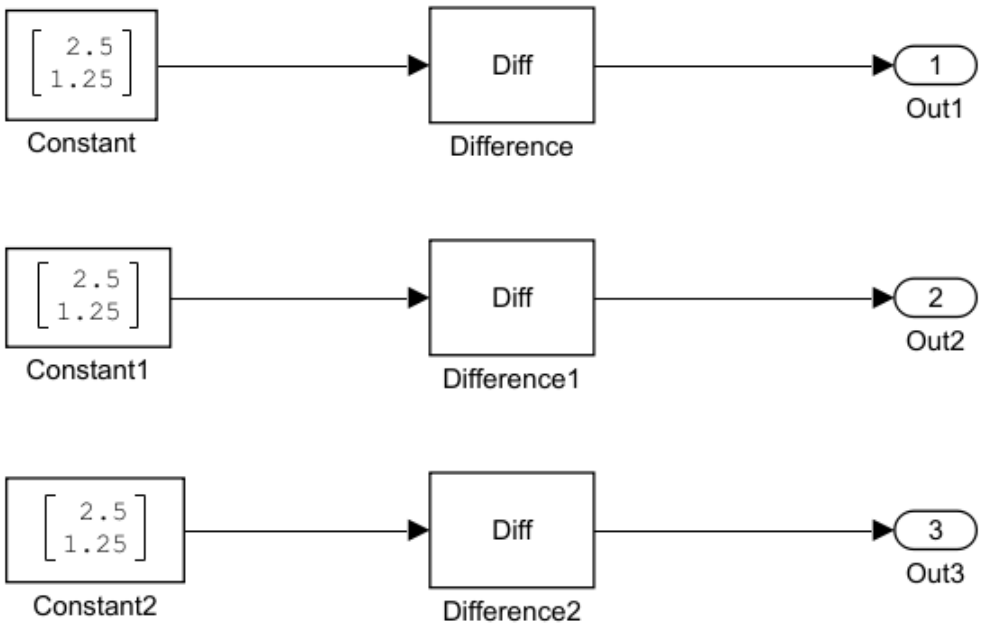
The largest word length allowed for Simulink and toolbox software on any target is 128 bits.

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 21-20 and product data types on page 21-23.

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:

$$WL_{idealaccumulator} = WL_{inputtoaccumulator} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator} = 9 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator} = 9 + 0 + 1 = 10$$

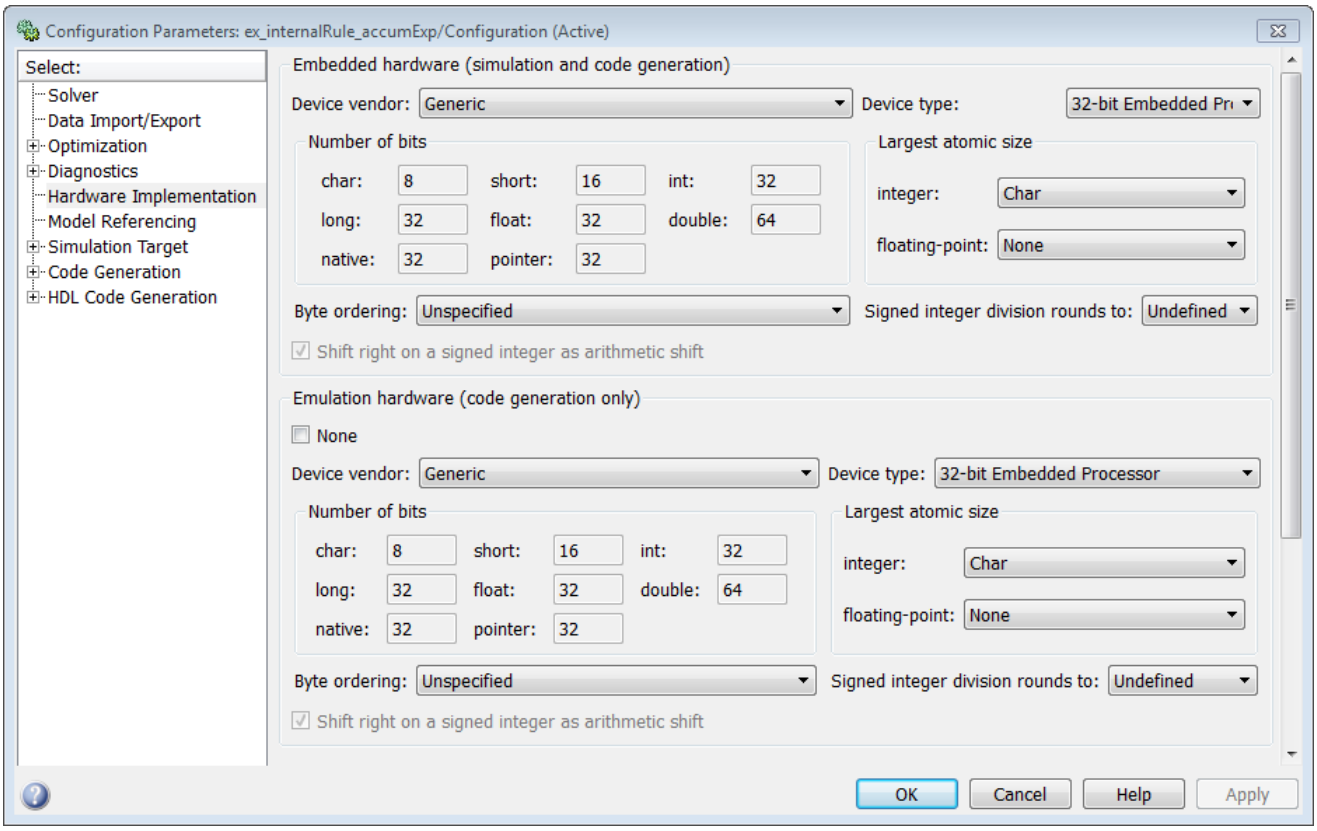
$$WL_{idealaccumulator1} = WL_{inputtoaccumulator1} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator1} = 16 + 0 + 1 = 17$$

$$WL_{idealaccumulator2} = WL_{inputtoaccumulator2} + \text{floor}(\log_2(numberofaccumulations)) + 1$$
$$WL_{idealaccumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$
$$WL_{idealaccumulator2} = 127 + 0 + 1 = 128$$

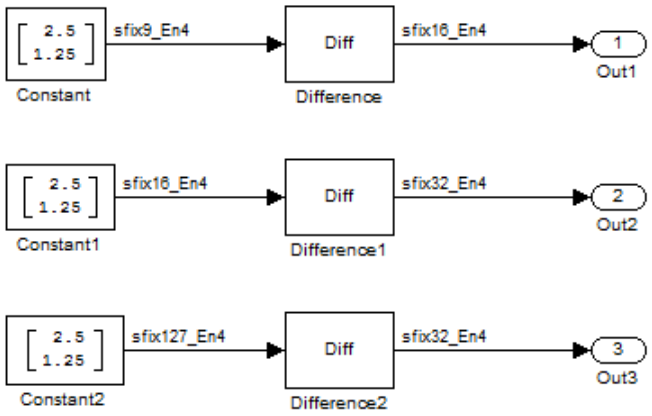
Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$
$$FL_{idealaccumulator} = 4$$

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.

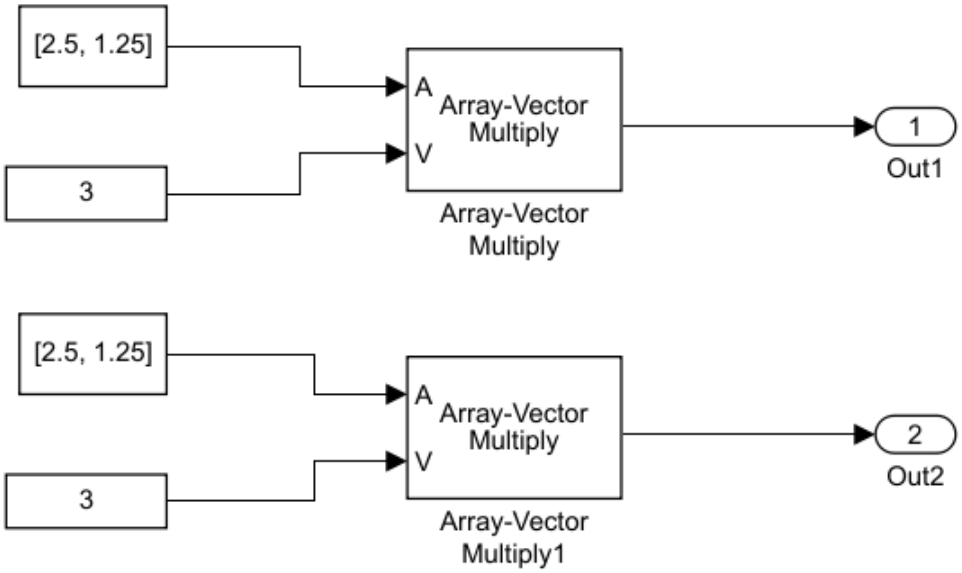


As you can see in the dialog box, this device has 8-, 16-, and 32-bit word lengths available. Therefore, the ideal word lengths of 10, 17, and 128 bits calculated by the internal rule cannot be used. Instead, the internal rule uses the next largest available word length in each case You can see this 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:

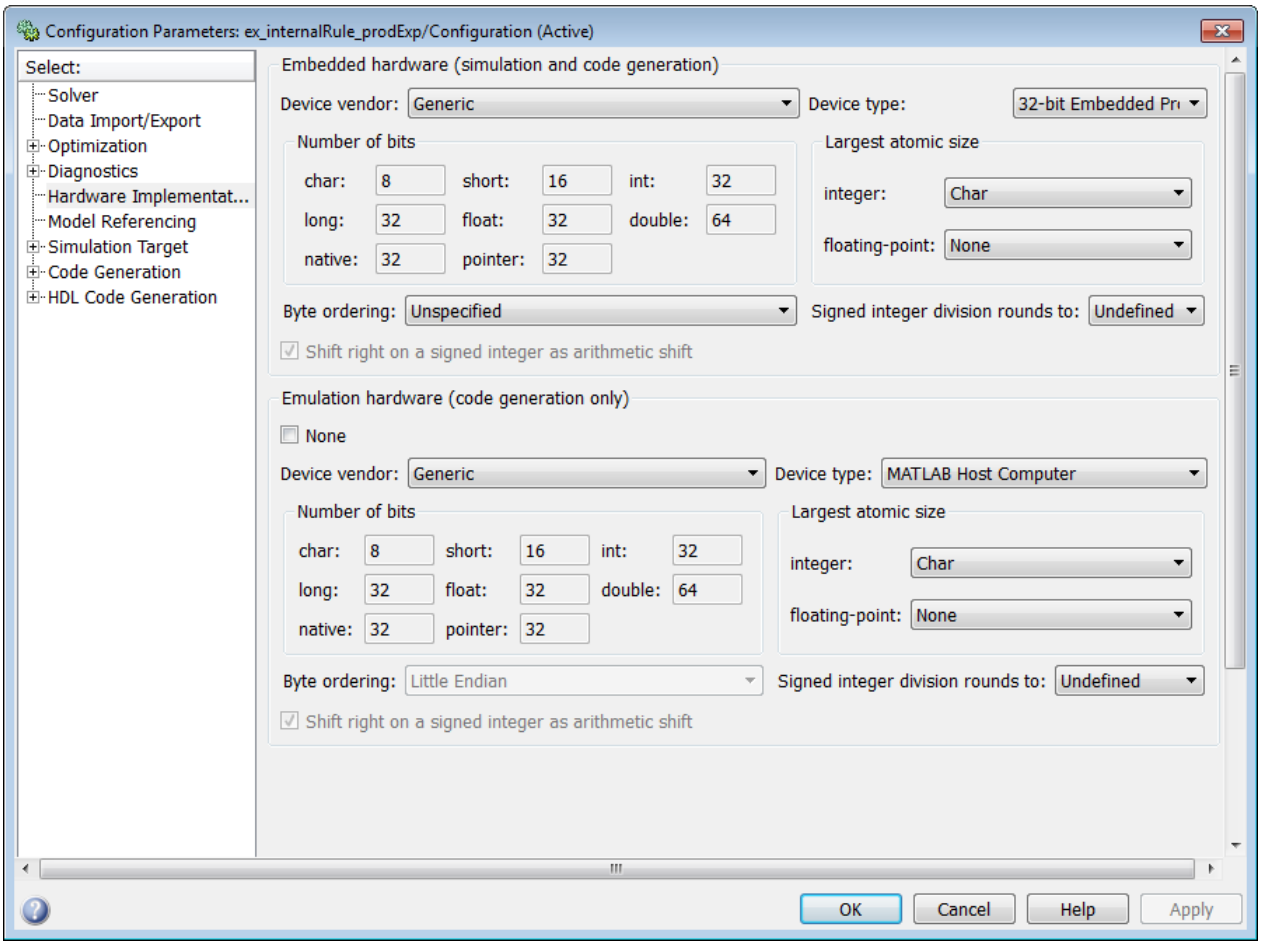
$$WL_{idealproduct} = WL_{inputa} + WL_{inputb}$$
$$WL_{idealproduct} = 7 + 5 = 12$$

$$WL_{idealproduct1} = WL_{inputa} + WL_{inputb}$$
$$WL_{idealproduct1} = 16 + 15 = 31$$

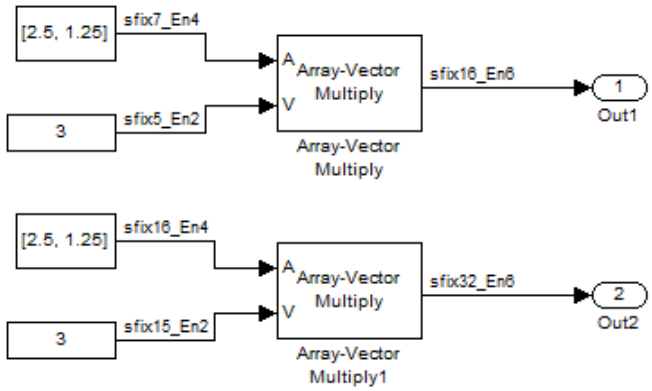
Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

$$FL_{idealaccumulator} = FL_{inputtoaccumulator}$$
$$FL_{idealaccumulator} = 4$$

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.



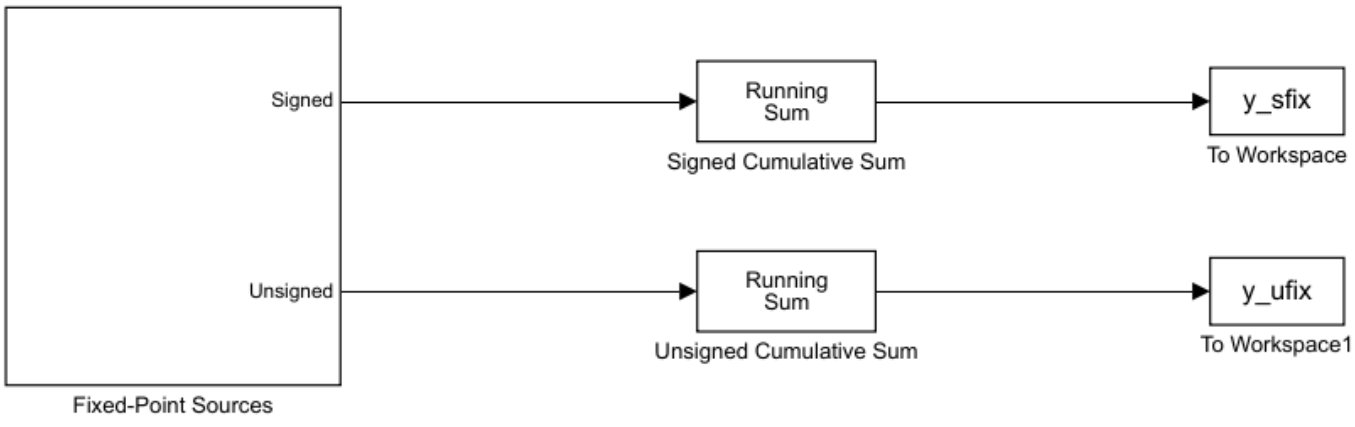
Specify Data Types for Fixed-Point Blocks

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 21-25
- "Use Data Type Override to Find a Floating-Point Benchmark" on page 21-29
- "Use the Fixed-Point Tool to Propose Fraction Lengths" on page 21-29
- "Examine the Results and Accept the Proposed Scaling" on page 21-30

Prepare the Model

1 Open the model by typing ex_fixedpoint_tut at the MATLAB command line.



This model uses the Cumulative Sum block to sum the input coming from the Fixed-Point Sources subsystem. The Fixed-Point Sources subsystem outputs two signals with different data types:

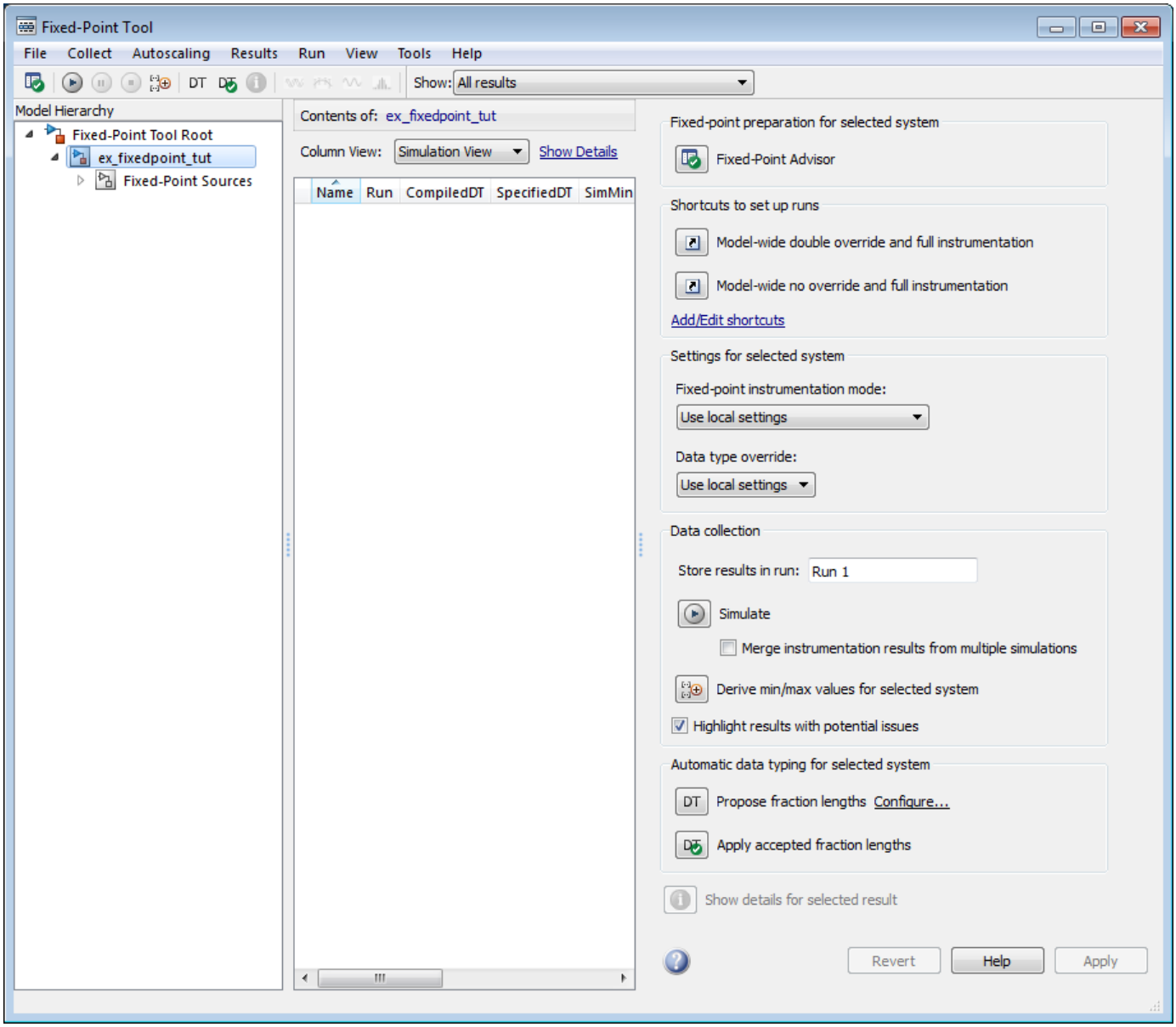
- The Signed source has a word length of 16 bits and a fraction length of 15 bits.
- The Unsigned source has a word length of 16 bits and a fraction length of 16 bits.

2. Run the model to check for overflow. MATLAB displays the following warnings at the command line:

```
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Signed Cumulative Sum'.
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Unsigned Cumulative Sum'.
```

According to these warnings, overflow occurs in both Cumulative Sum blocks.

3. To investigate the overflows in this model, use the Fixed-Point Tool. You can open the Fixed-Point Tool by selecting **Tools > Fixed-Point > Fixed-Point Tool** from the model menu. Turn on logging for all blocks in your model by setting the **Fixed-point instrumentation mode** parameter to `Minimums, maximums and overflows`.
4. Now that you have turned on logging, rerun the model by clicking the Simulation button.



5 The results of the simulation appear in a table in the central **Contents** pane of the Fixed-Point Tool. Review the following columns:

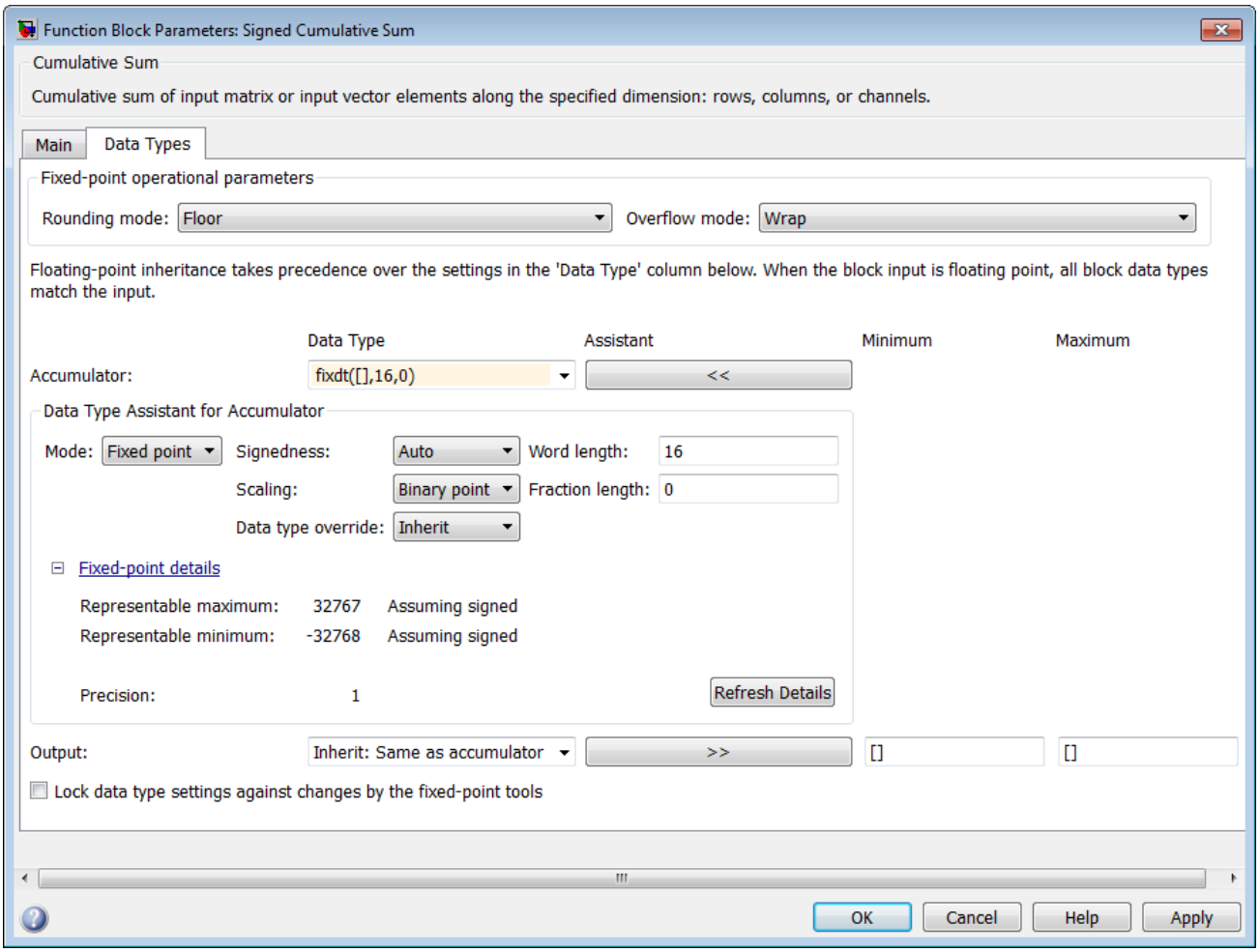
- **Name** — Provides the name of each signal in the following format: `Subsystem Name/Block Name: Signal Name`.
- **SimDT** — The simulation data type of each logged signal.
- **SpecifiedDT** — The data type specified on the block dialog for each signal.
- **SimMin** — The smallest representable value achieved during simulation for each logged signal.
- **SimMax** — The largest representable value achieved during simulation for each logged signal.
- **OverflowWraps** — The number of overflows that wrap during simulation.

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 22-2
- “Accelerating Simulink Models” on page 22-3
- “Portable C Code Generation for Functions That Use OpenCV Library” on page 22-4

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 or Share Generated Code” (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:

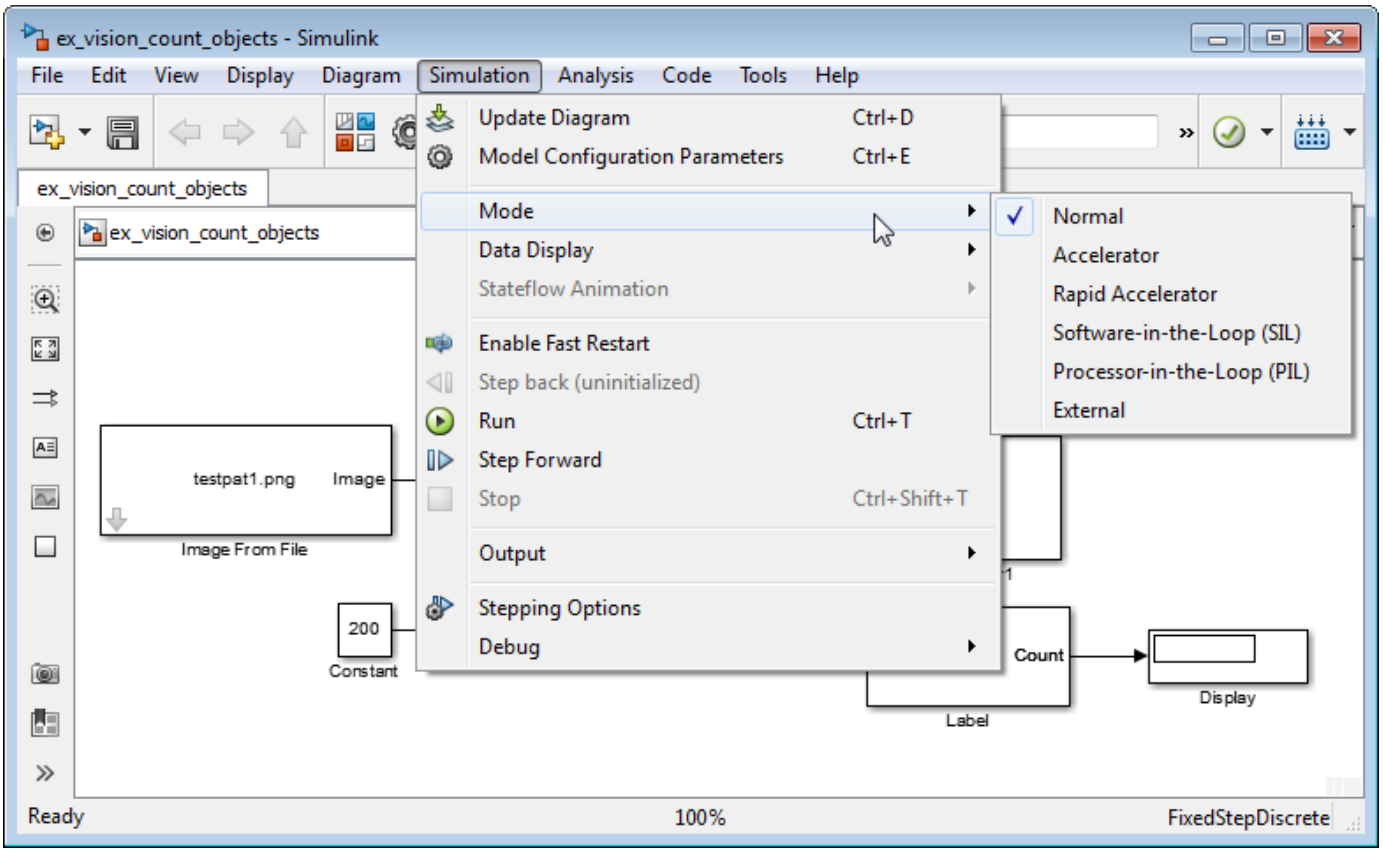
`Host computer only. Excludes` Simulink Desktop Real-Time™ `target`.

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

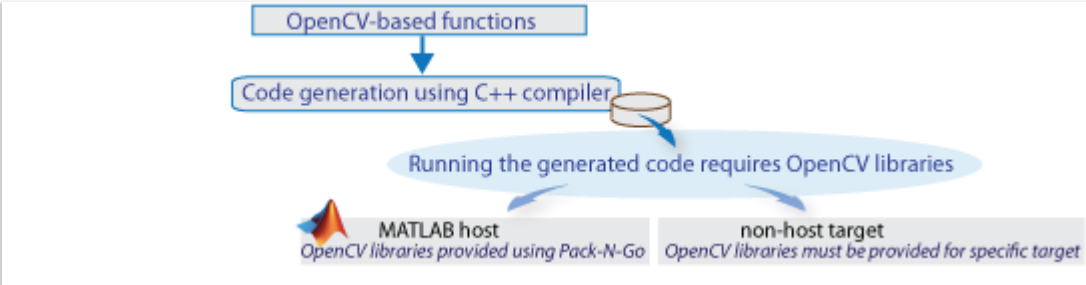
The Simulink software offer `Accelerator` and `Rapid Accelerator` simulation modes that remove much of the computational overhead required by Simulink models. These modes compile target code of your model. Through this method, the Simulink environment can achieve substantial performance improvements 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 or later (Professional or Community editions)
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

- “Rotate ROI in Image” on page 23-3
- “Apply Horizontal Shear Transformation to Image” on page 23-6
- “Find Location of Object in Image Using Template Matching” on page 23-9
- “Compute Optical Flow Velocities” on page 23-12
- “Rotate an Image” on page 23-14
- “Generate Image Histogram” on page 23-17
- “Export Image to MATLAB Workspace” on page 23-19
- “Import Video from MATLAB Workspace” on page 23-22
- “Find Minimum Value in ROI” on page 23-24
- “Write Image to Binary File” on page 23-28
- “Compute Standard Deviation of ROIs” on page 23-29
- “Read Video Stored as Binary Data” on page 23-32
- “Compare Image Quality Using PSNR” on page 23-36
- “Compute Autocorrelation of Input Matrix” on page 23-38
- “Compute Correlation between Two Matrices” on page 23-39
- “Find Statistics of Circular Blobs in Image” on page 23-40
- “Replace Intensity Values in ROI with its Maximum Value” on page 23-44
- “Median based Image Thresholding” on page 23-48
- “Import Image From MATLAB Workspace” on page 23-51
- “Import Image from Specified Location” on page 23-53
- “Remove Interlacing Effect From Image” on page 23-57
- “Estimate Motion between Two Images” on page 23-60
- “Enhance Contrast of Grayscale Image Using Histogram Equalization” on page 23-62
- “Enhance Contrast of Color Image Using Histogram Equalization” on page 23-65
- “Compute Mean of ROIs in Image” on page 23-68
- “Detect Corners in Image” on page 23-71
- “Edge Detection of Intensity Image” on page 23-75
- “Read, Process, and Write Video Frames to File” on page 23-78
- “Find Local Maxima in Image” on page 23-80
- “Read, Convert, and View Video from File” on page 23-83
- “Read and Display YCbCr Video from File” on page 23-85
- “Display Frame Rate of Input Video” on page 23-87
- “Draw Rectangles on Image” on page 23-88
- “Draw Circles on Image” on page 23-90
- “Overlay Images Using Binary Mask” on page 23-92

- “Linearly Combine Two Images” on page 23-97
- “Pad Zeros to Image” on page 23-101
- “Insert Text into Image” on page 23-104
- “Compress Image Using 2-D DCT” on page 23-107
- “Draw Markers on Image” on page 23-111
- “Read and Display RGB Video from File” on page 23-114
- “Label Objects in Binary Image” on page 23-116
- “Boundary Extraction of Binary Image” on page 23-120
- “Select String to Insert into Image” on page 23-124
- “Insert Two Strings into Image at Different Locations” on page 23-127
- “Dilation of Binary Image” on page 23-129
- “Find Complement of Intensity Image” on page 23-131
- “Perform Top-Hat Filtering of Binary Image” on page 23-134
- “Perform Bottom-hat Filtering of Binary Image” on page 23-137
- “Perform Opening of Binary Image” on page 23-140
- “Perform Closing of Binary Image” on page 23-143
- “Blur Image Using Gaussian Kernel” on page 23-146
- “Convert Image Color Space from RGB to YCbCr” on page 23-149
- “Convert Data Type and Color Space of Image from RGB to HSV” on page 23-152
- “Perform Gamma Correction of Image” on page 23-155
- “Adjust Contrast of Image” on page 23-158
- “Remove Impulse Noise from Image” on page 23-161
- “Draw Hough Lines on Image” on page 23-164
- “Construct Laplacian Pyramid Image” on page 23-166

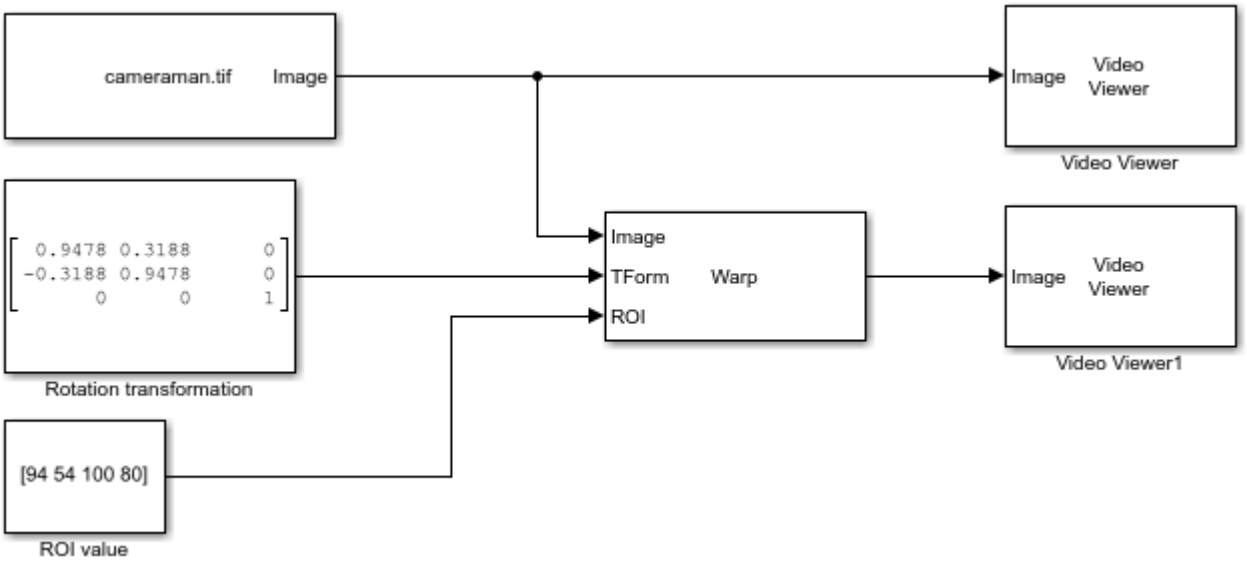
Rotate ROI in Image

This example shows how to apply rotation transformation to a region of interest (ROI) in the input image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_rotatewarp.slx';
open_system(modelname)
```



The model reads an input image by using the Image From File block. To apply rotation transformation to a particular region in the input image, you must

- specify the rotation transformation matrix
- specify the ROI

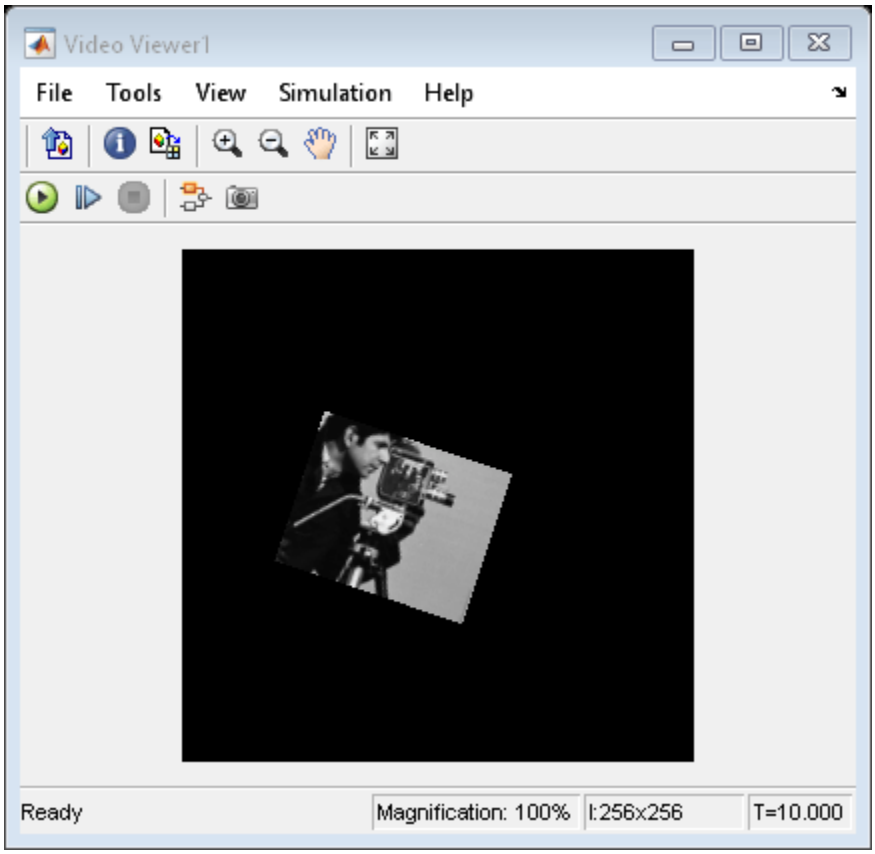
The model then rotates the selected ROI by using the `Warp` block with these parameter values:

- **Transformation matrix source** - `Input port`. This enables the `TForm` input port. Specify the rotation transformation matrix value as $\begin{bmatrix} 0.9478 & 0.3188 & 0 \\ -0.3188 & 0.9478 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ by using the `Constant` block.
- **Interpolation method** - `Bilinear`
- **Background fill value** - `0`
- **Output image position source** - `Same as input image`
- Select the **Enable ROI input port** parameter. Specify the value for ROI input port as $\begin{bmatrix} 94 & 54 & 100 & 80 \end{bmatrix}$ by using the `Constant` block.

Simulate and Display Results

Run the model and display the results by using the Video Viewer block.

```
sim(modelname);
```


Video Viewer1
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=10.000

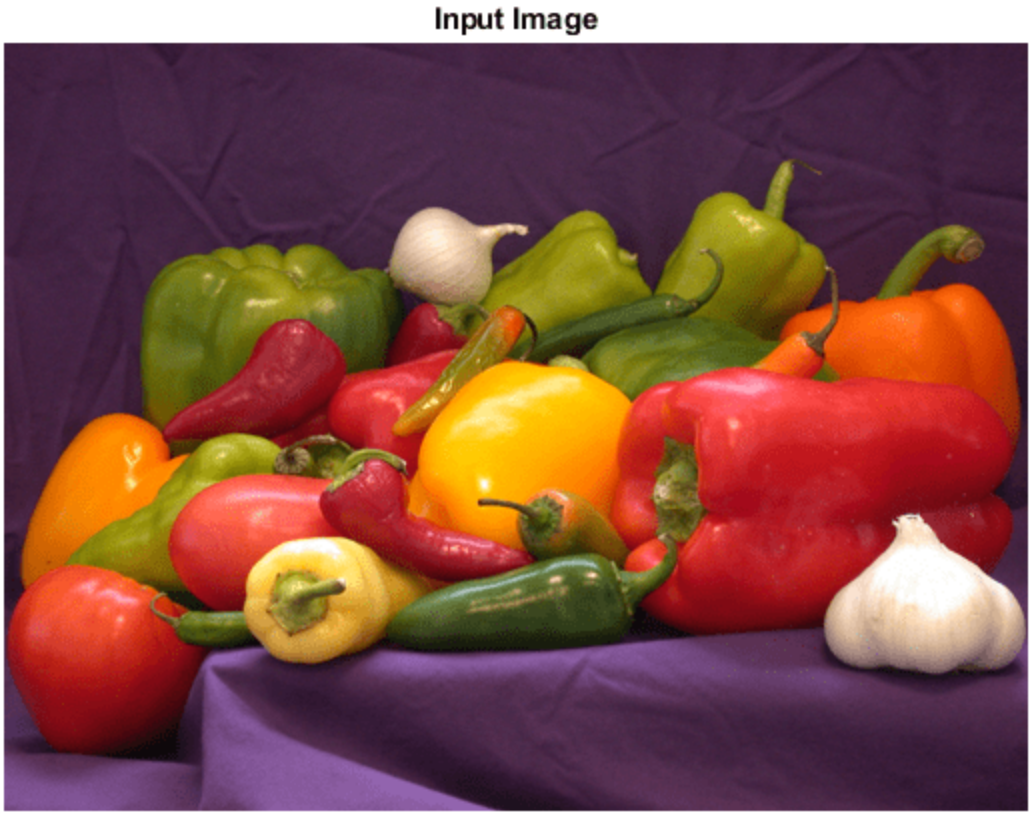
Apply Horizontal Shear Transformation to Image

Read an image into the MATLAB workspace.

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

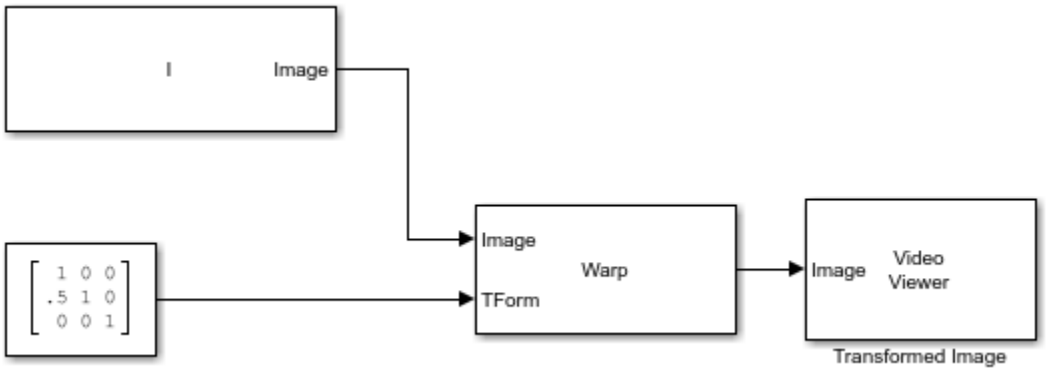
Display the input image.

```
figure
imshow(I)
title('Input Image')
```



Open the Simulink® model.

```
modelname = 'ex_shearblkwarp.slx';
open_system(modelname)
```



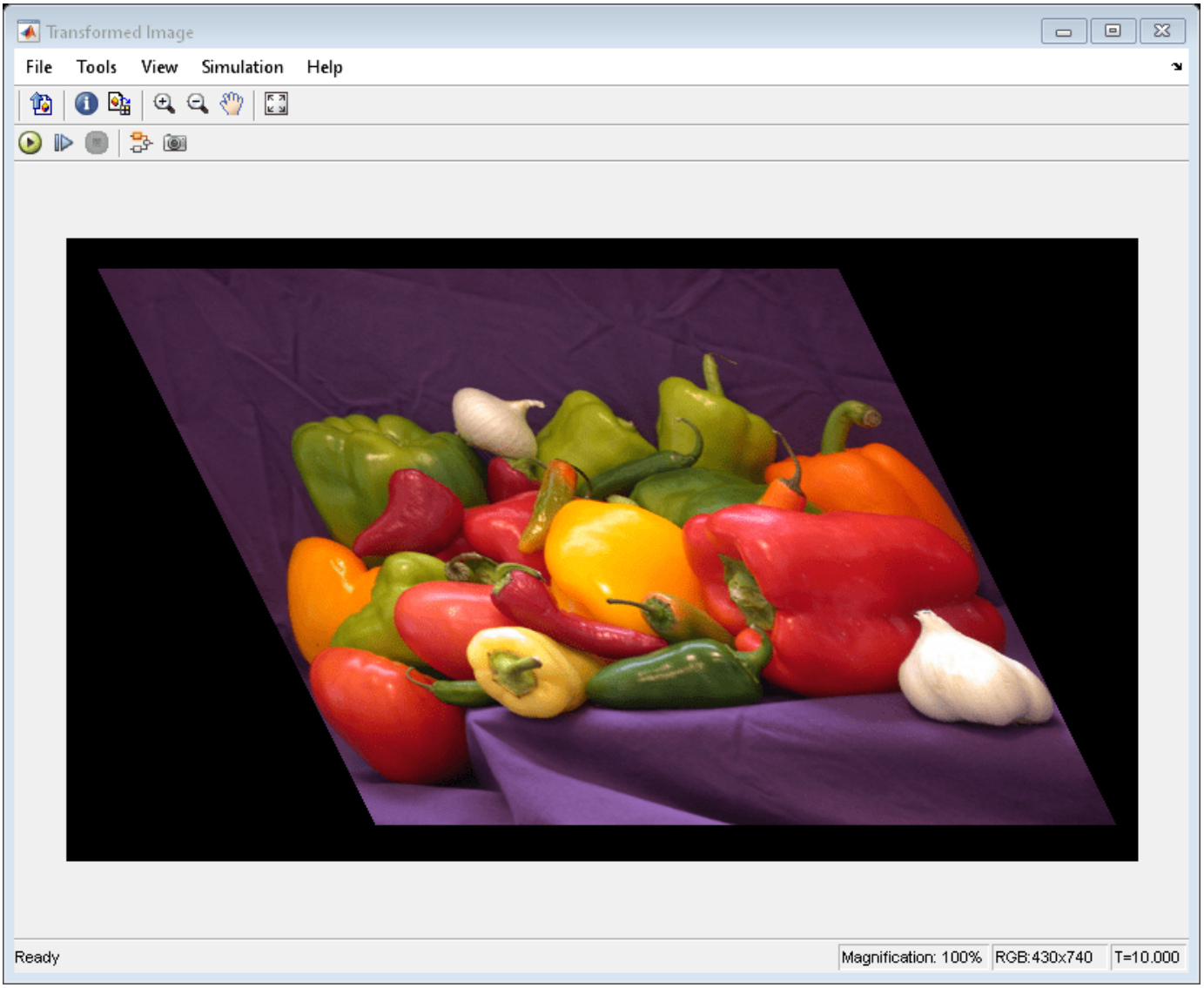
The model reads the input image from the workspace by using the Image From Workspace block. To apply horizontal shear transformation, you must specify the transformation matrix at the input by using `TForm` input port.

The model then shears the image to the horizontal direction by using the `Warp` block with these parameter values:

- **Transformation matrix source** - `Input port`. This enables the `TForm` input port. Specify the horizontal shear transformation matrix value as $\begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ by using the `Constant` block.
- **Interpolation method** - `Bilinear`
- **Background fill value** - `0`
- **Output image position source** - `Custom`
- **Output image position vector [x y width height]** - `[-20 -20 740 430]`

Run the model and display the output transformed image by using the Video Viewer block.

```
sim(modelname);
```


Transformed Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
RGB:430x740
T=10.000

Find Location of Object in Image Using Template Matching

Read the reference image and the template image into the MATLAB workspace.

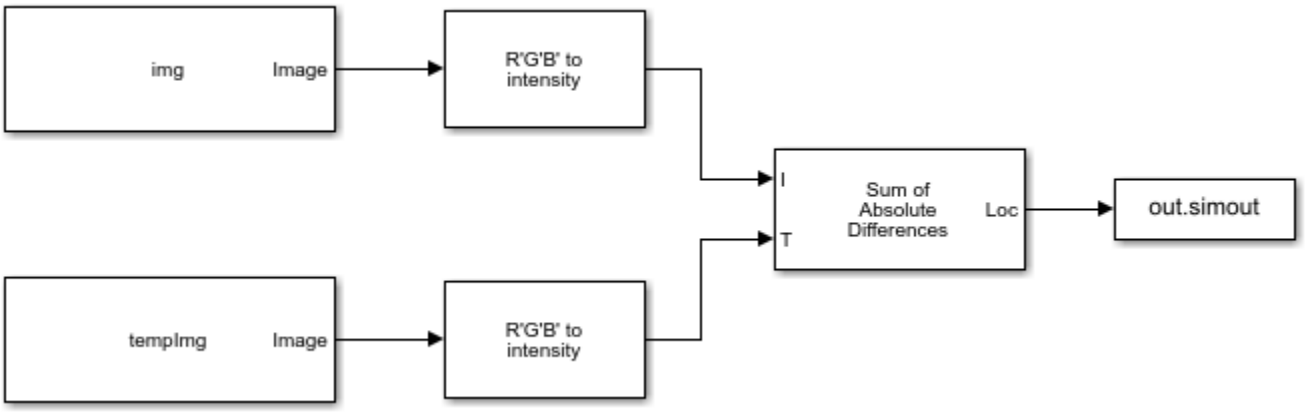
```
img = imread('stopSignTest.jpg');
tempImg = imread('vipwarningsigns_stop_template.png');
```

Display the images. The reference image is a scene containing the stop sign board. The template is a low spatial resolution image of the stop sign board.

```
figure
imshow(img)
title('Reference Image')
figure
imshow(tempImg)
title('Template')
```

Open the Simulink® model.

```
modelname = 'ex_blktemplatematching.slx';
open_system(modelname)
```



The model reads the images by using the Image From Workspace block. To perform template matching, you must first convert the input color images to intensity images by using the Color Space Conversion block. Then, find the location of the template image in the reference image by using the `Template Matching` block with these parameter values:

- **Match Metric** - `Sum of absolute differences`
- **Output** - `Best match location`
- **Search method** - `Three-step`

The `Template Matching` block outputs a location in the reference image for which the pixel regions around it best matches with the template image.

Run the model.

```
out = sim(modelname);
```

Read the output value.

```
location = out.simout;
```

Draw a circle to highlight the region around the best matching pixel location. Display the results.

```
img = insertShape(img,'circle',[location(1) location(2) 20]);
figure
imshow(img);
hold on
plot(location(1),location(2),'*r')
title('Results of Template Matching')
```

Results of Template Matching

Compute Optical Flow Velocities

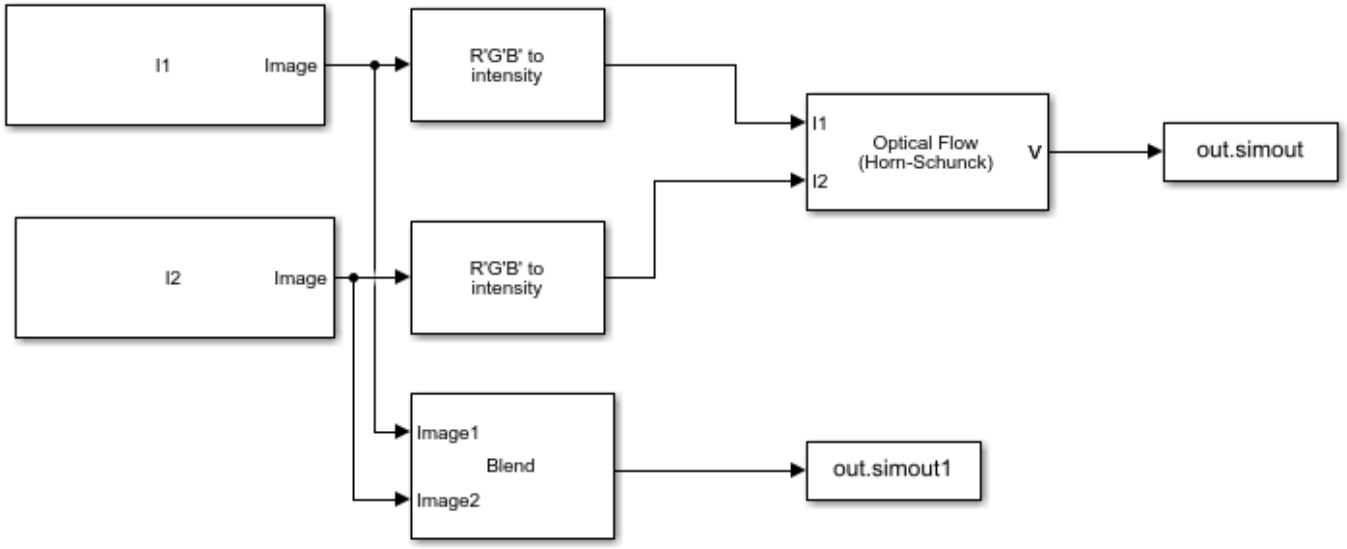
This example shows how to compute the optical flow velocities for a moving object in a video or image sequence.

Read two image frames from an image sequence into the MATLAB workspace.

```
I1 = imread('car_frame1.png');
I2 = imread('car_frame2.png');
```

Open the Simulink® model.

```
modelname = 'ex_blkopticalflow.slx';
open_system(modelname)
```



The model reads the images by using the Image From Workspace block. To compute the optical flow velocities, you must first convert the input color images to intensity images by using the Color Space Conversion block. Then, find the velocities by using the `Optical Flow` block with these parameter values:

- **Method** - `Horn-Schunck`
- **Compute optical flow between** - `Two images`
- **Smoothness factor** - `1`
- **Stop iterative solution** - `When maximum number of iterations is reached`
- **Maximum number of iterations** - `10`
- **Velocity output** - `Horizontal and vertical components in complex form`

Overlay both the image frames by using the Compositing block and use the overlaid image to plot the results.

Run the model.

```
out = sim(modelname);
```

Read the output velocities and the overlaid image.

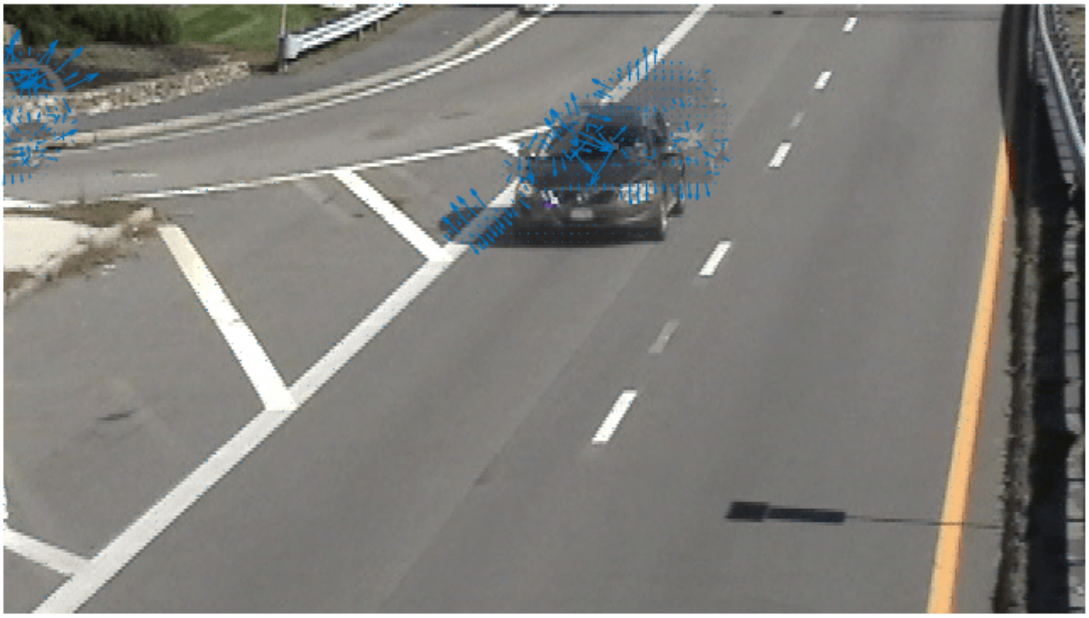
```
Vx = real(out.simout);
Vy = imag(out.simout);
img = out.simout1;
```

Create an optical flow object by using the `opticalFlow` function.

```
flow = opticalFlow(Vx,Vy);
```

Display the overlaid image and plot the velocity vectors by using the `plot` function.

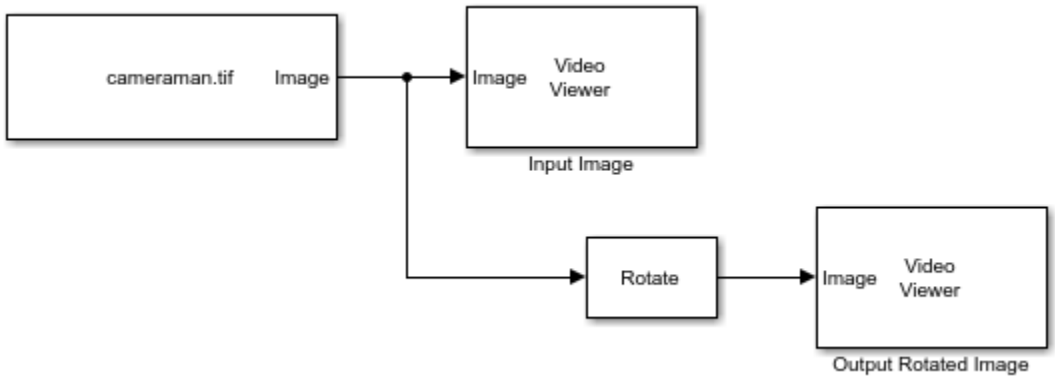
```
figure
imshow(img)
hold on
plot(flow,'DecimationFactor',[5 5],'ScaleFactor',40)
```



Rotate an Image

Open the Simulink® model.

```
modelname = 'ex_blkrotate.slx';
open_system(modelname)
```



The model reads image by using the Image From File block. To rotate the input image, use the `Rotate` block with these parameter values:

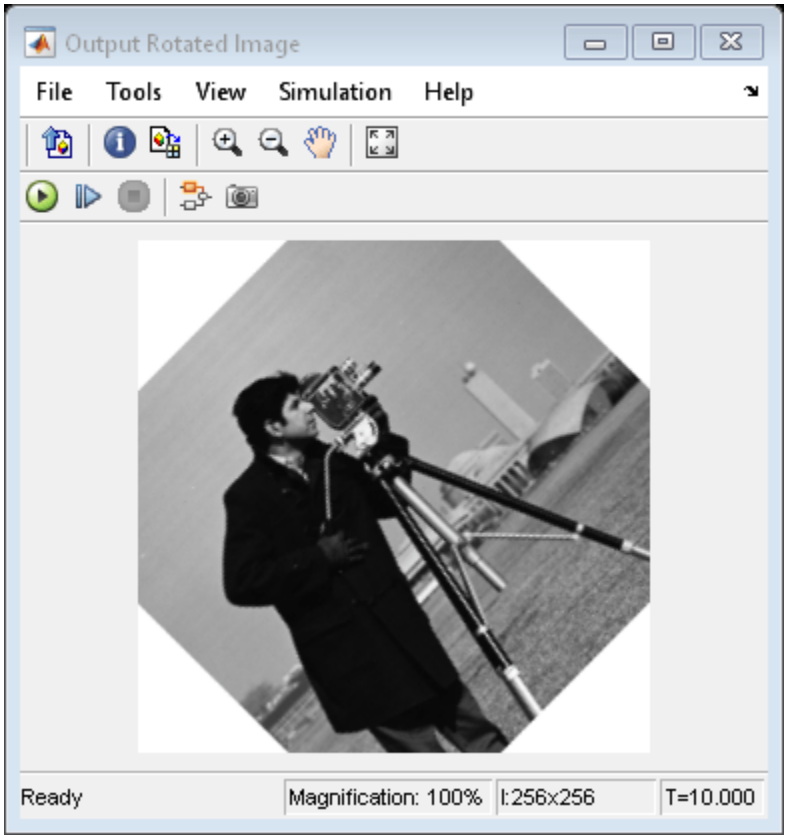
- **Output size** - `Same as input image`
- **Rotation angle source** - `Specify via dialog`
- **Angle (radians)** - `pi/4`
- **Sine value computation method** - `Table lookup`
- **Background fill value** - `255`
- **Interpolation method** - `Bilinear`

To display the input and the output images, use the Video Viewer block.

Run the model.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=10.000


Output Rotated Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=10.000

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.

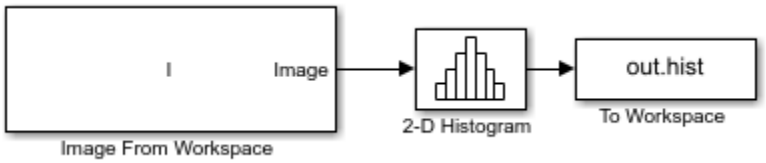
```
I = imread('cameraman.tif');
```

Find the maximum intensity value in the input image.

```
maxI = max(I(:));
```

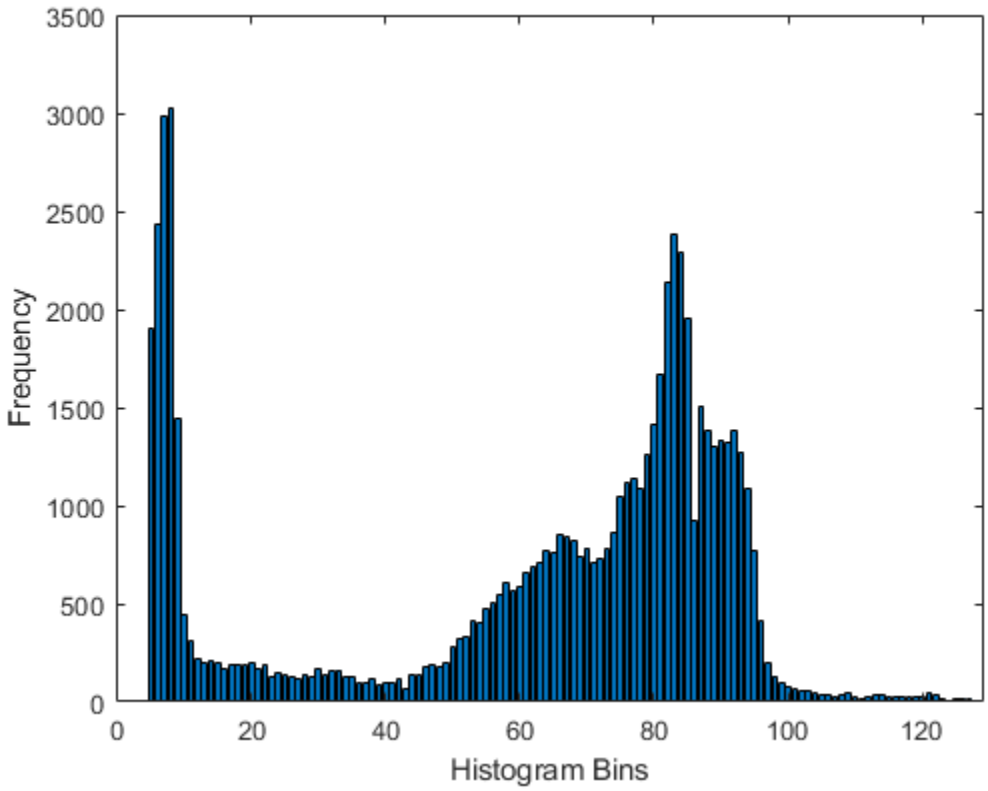
Open the simulink model. The model reads the image stored in variable `I` from the MATLAB workspace. The `Upper limit of histogram` parameter of the `2-D Histogram` block is set to the maximum value of the intensity image. The `Number of Bins` parameter of the `2-D Histogram` block is set to 128 and 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')
```


3500
3000
2500
2000
1500
1000
500
0
Frequency
0
20
40
60
80
100
120
Histogram Bins

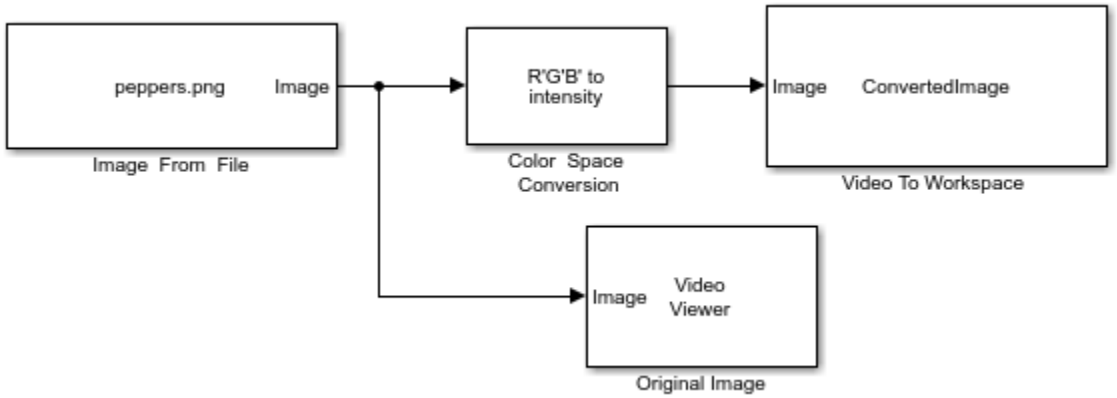
Export Image to MATLAB Workspace

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);
```

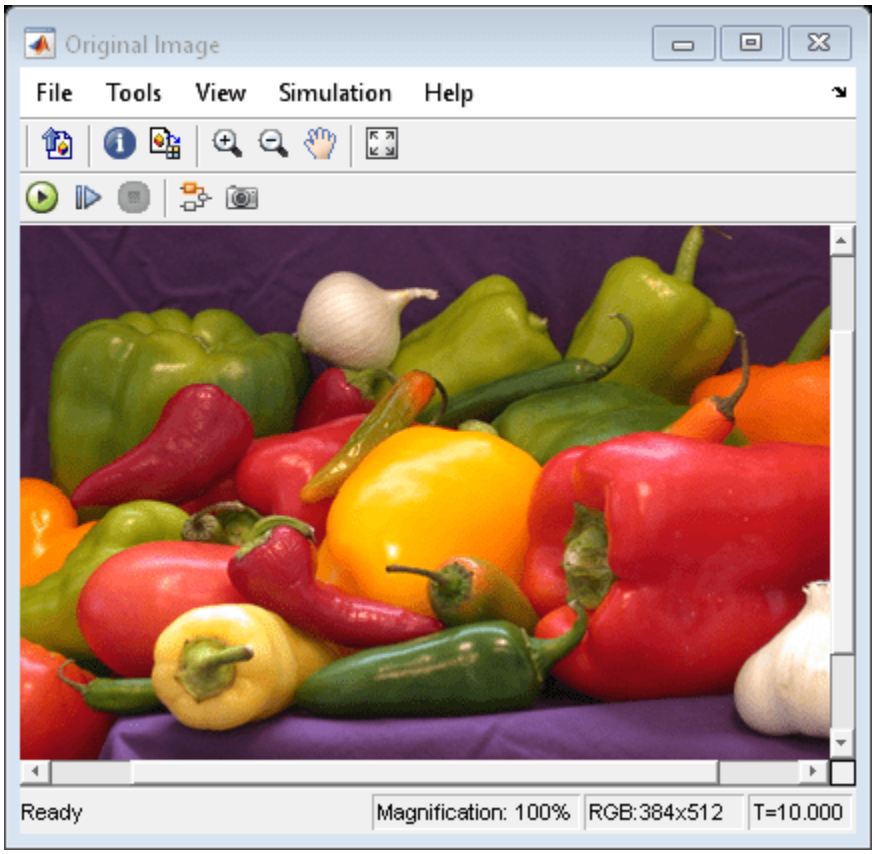


To convert the original image into gray scale, set the `Conversion` parameter of `Color Space Conversion` block to `R'G'B to intensity`. The original image is of size 384-by-512-by-3 and the gray scale image output from the `Color Space Conversion` block is of size 384-by-512.

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

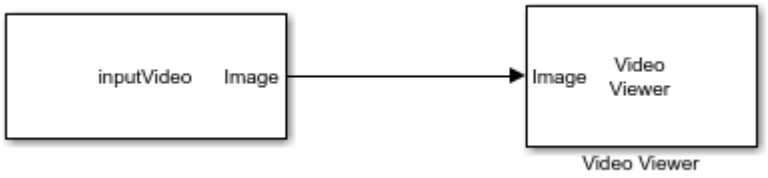
This example shows how to import a video from MATLAB to Simulink workspace by using `Video From Workspace` block.

Load the video data to MATLAB workspace.

```
load('videosignal.mat')
```

Open the Simulink model.

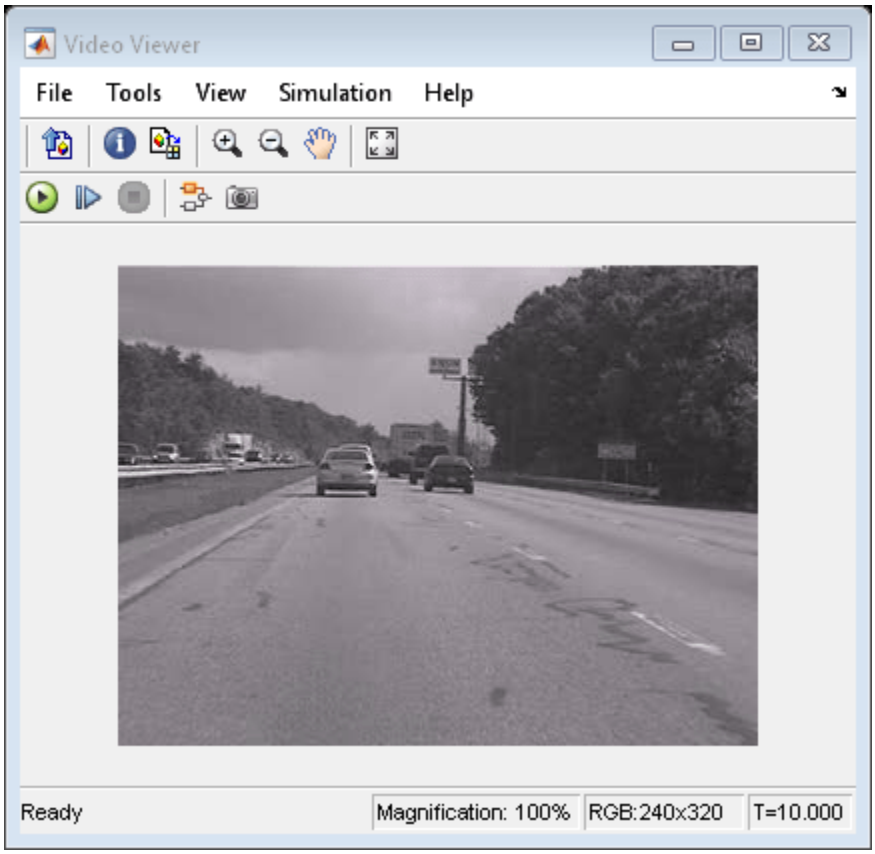
```
modelname='ex_blkvideofromworkspace.slx';
open_system(modelname);
```



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);
```


Video Viewer
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
RGB:240x320
T=10.000

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

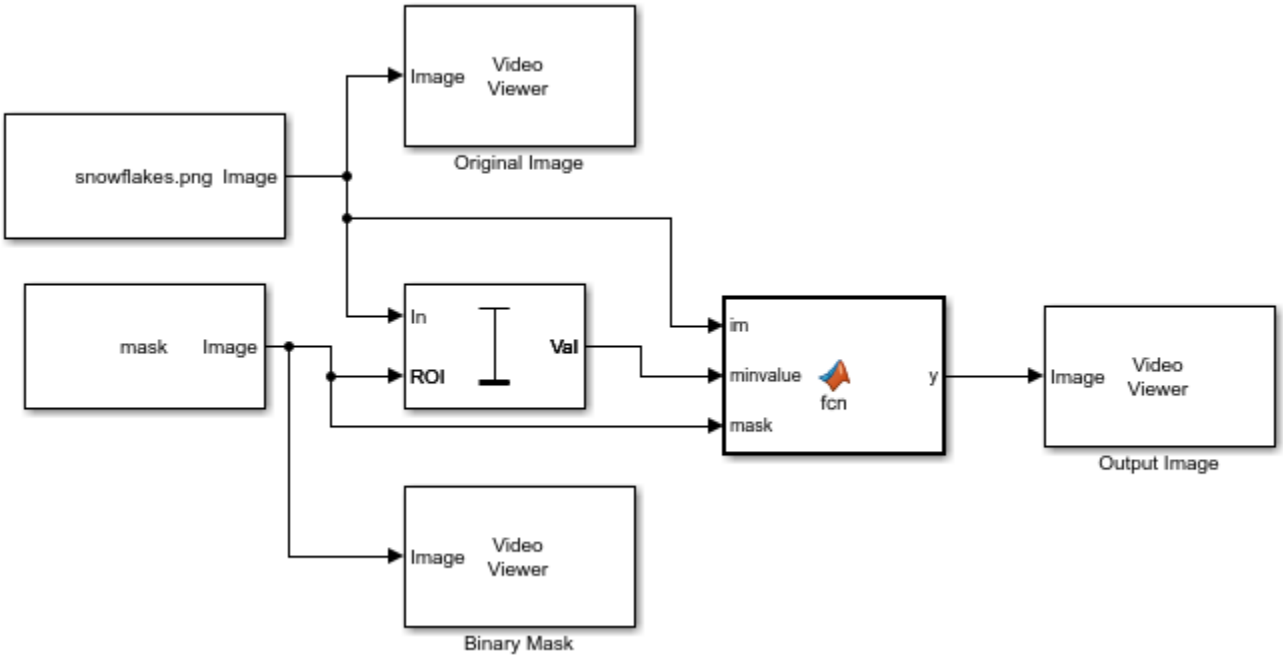
The model reads the original image and the binary mask comprising the ROI for which the minimum value has to be computed. The original image consists of regions with large image structures that are circular in shape and regions with small, indistinct image structures. The binary mask isolates the indistinct 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);
```



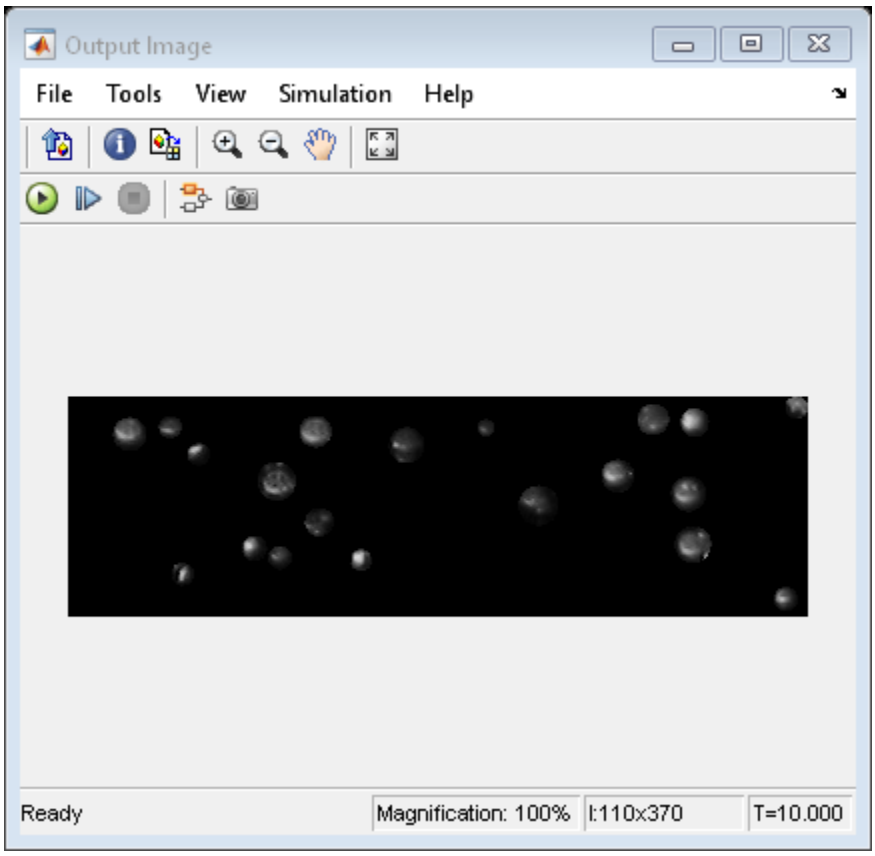
For the `2-D Minimum` block to output only the computed minimum, the `Mode` parameter of the block is set to `Value`. To perform ROI processing, the `Find the minimum value over` parameter is set to `Entire input`. The ROI input to the `2-D Minimum` block is a binary image. Hence, the `ROI Type` parameter 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.

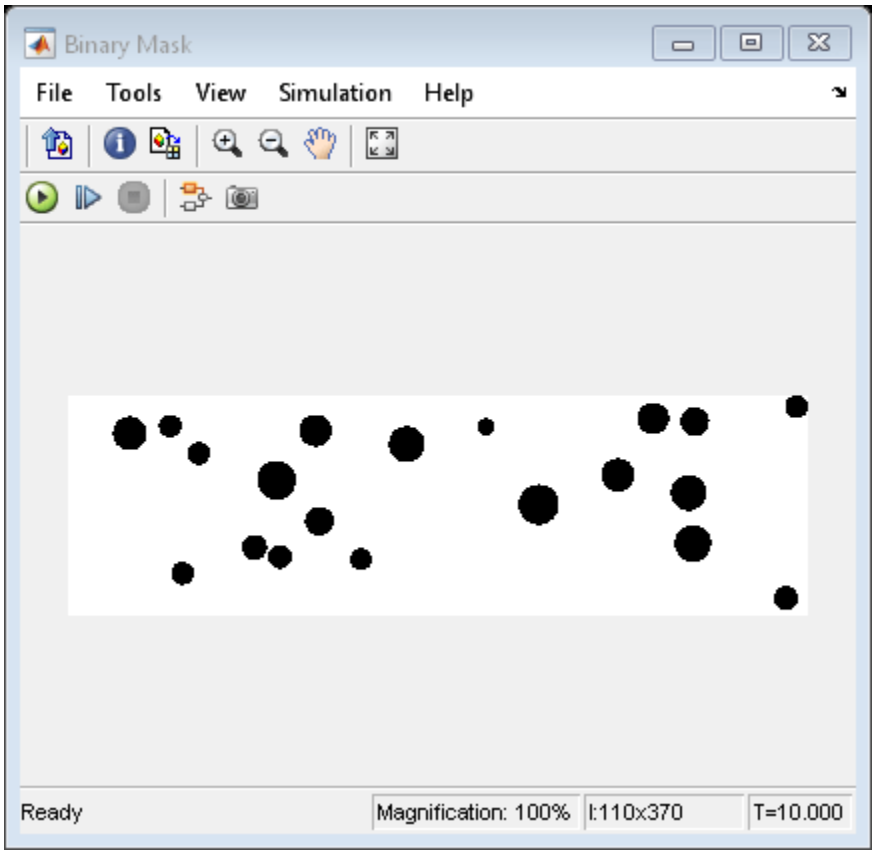
The MATLAB `function block` replaces pixel values in the ROI with the computed minimum.

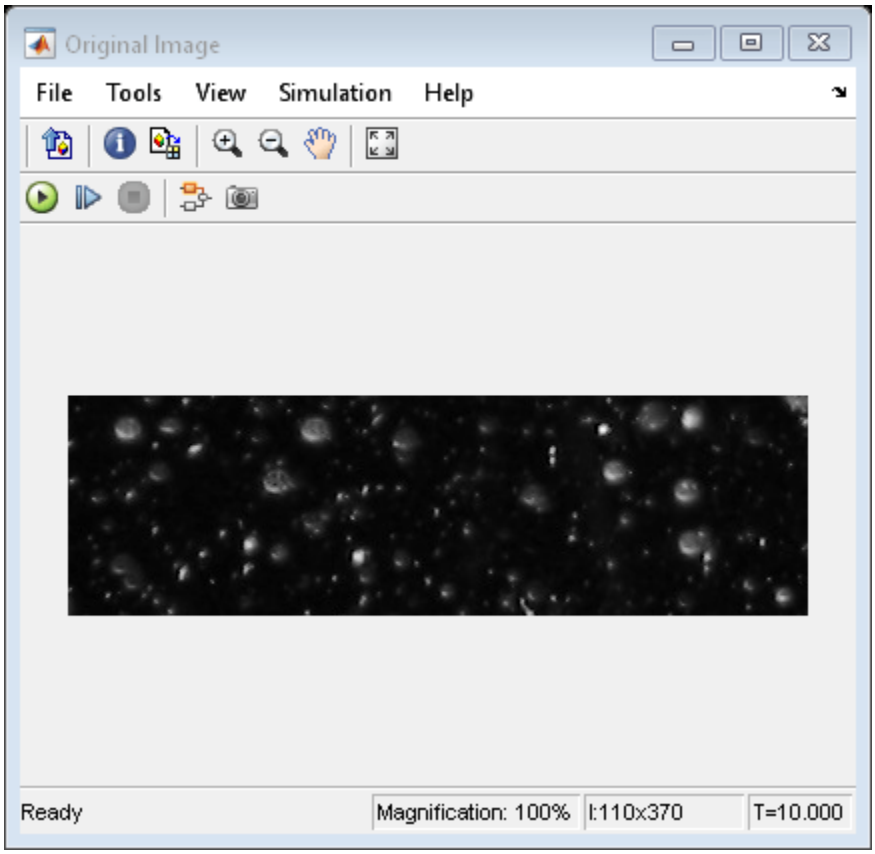
Simulate and Display Results

The model outputs a clean image with only distinct image structures.

```
sim(modelname);
```




Binary Mask
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:110x370
T=10.000

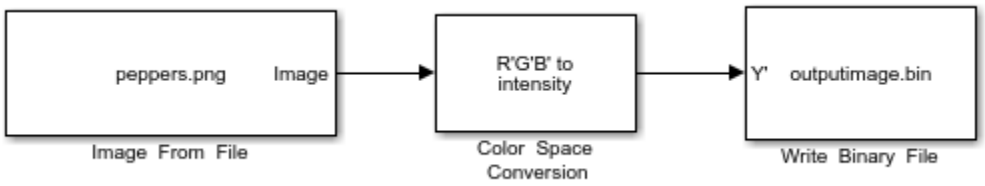

Original Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:110x370
T=10.000

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 coverts 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');
```

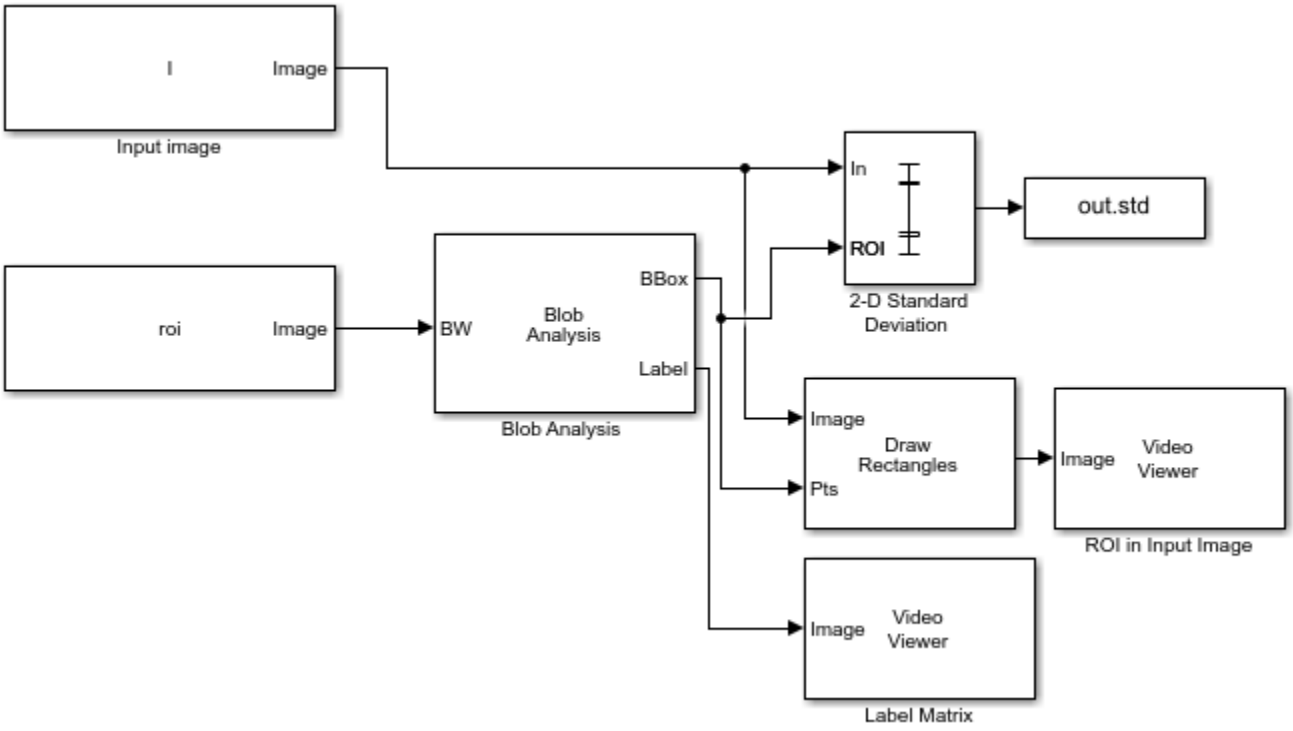
Load the mask image that specifies the ROIs in the input image.

```
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.

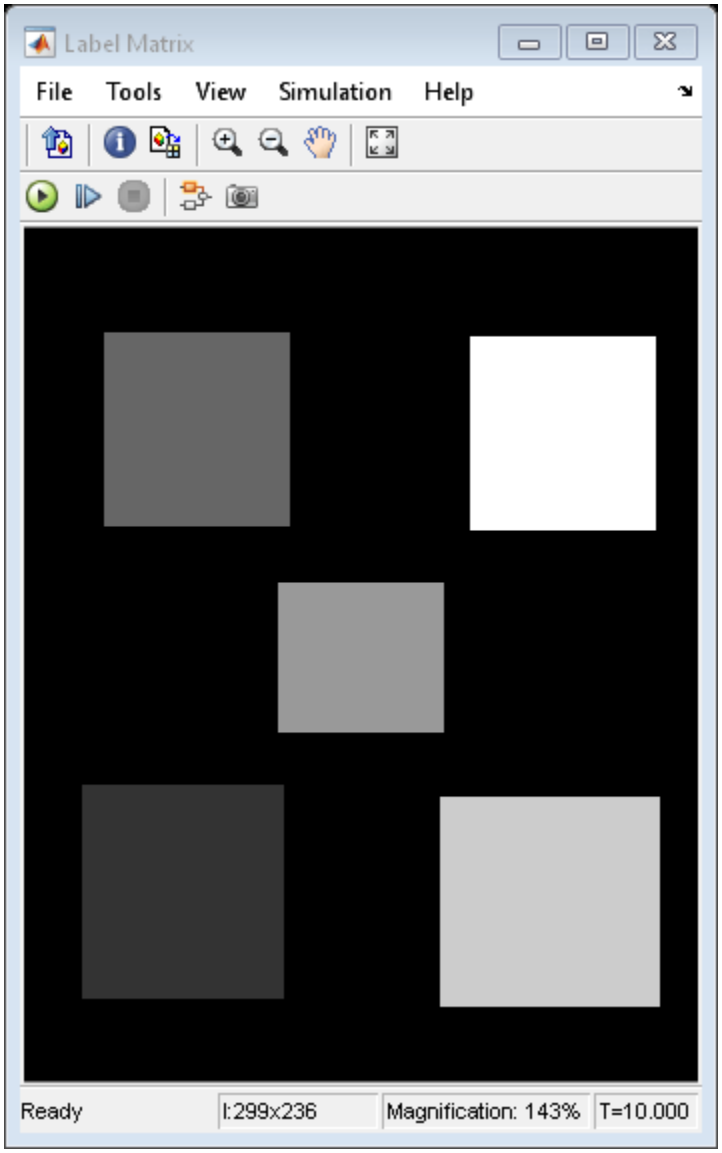
Set these parameters of the `2-D Standard Deviation` block to the specified value inorder to compute individual statistics for each ROI.

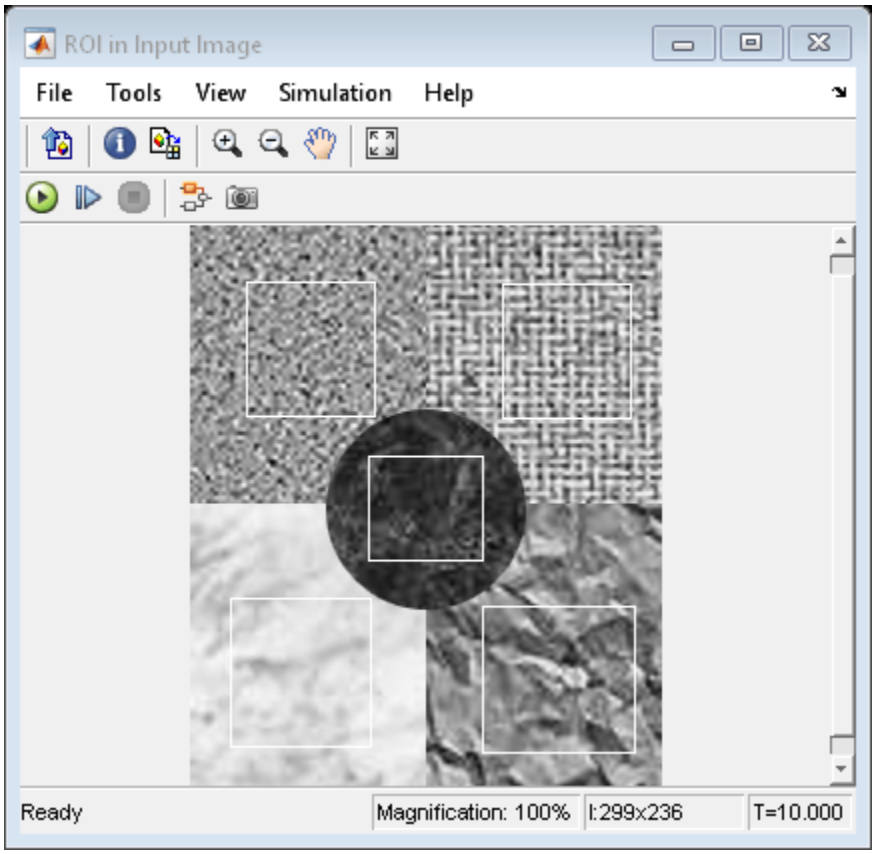
- Set `Find the standard deviation value over` parameter to `Entire input`
- Select `Enable ROI processing` parameter
- Set `ROI type` parameter to `Rectangles`
- Set `Output` parameter to `Individual statistics for each ROI`

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 overlayed 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
```

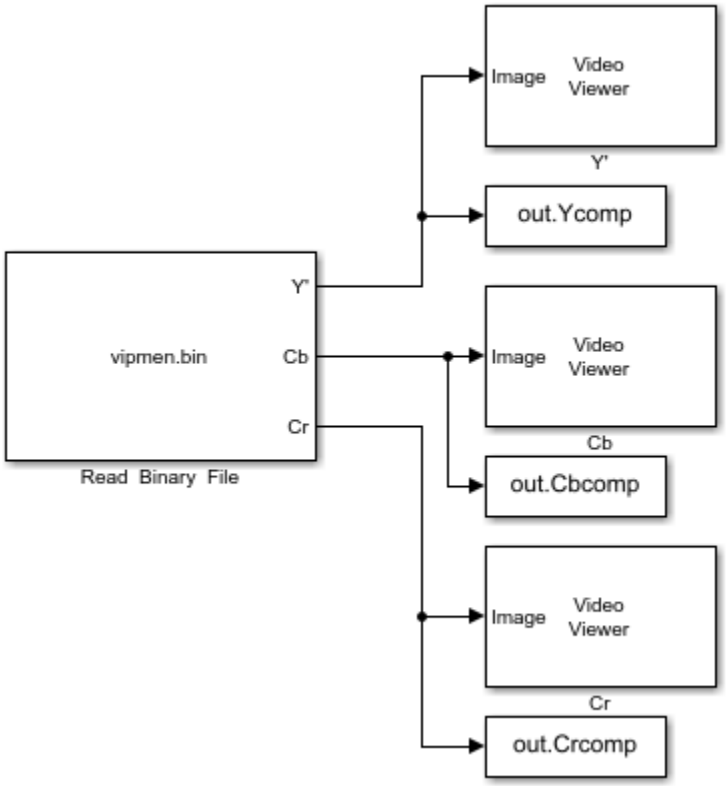
Read Video Stored as Binary Data

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);
```

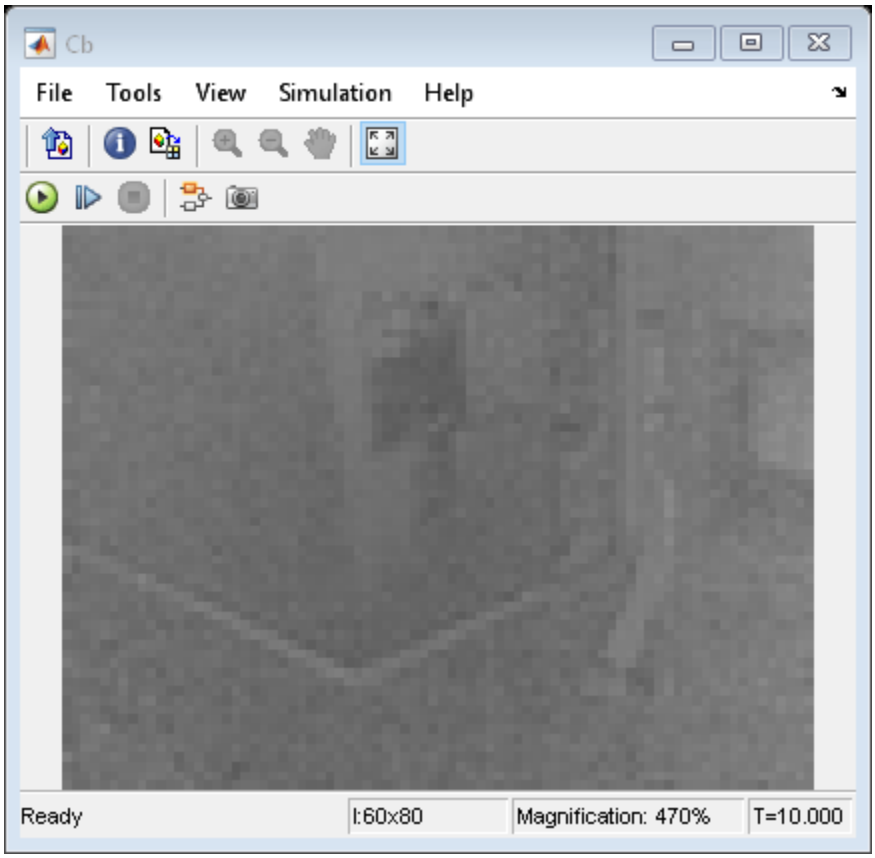


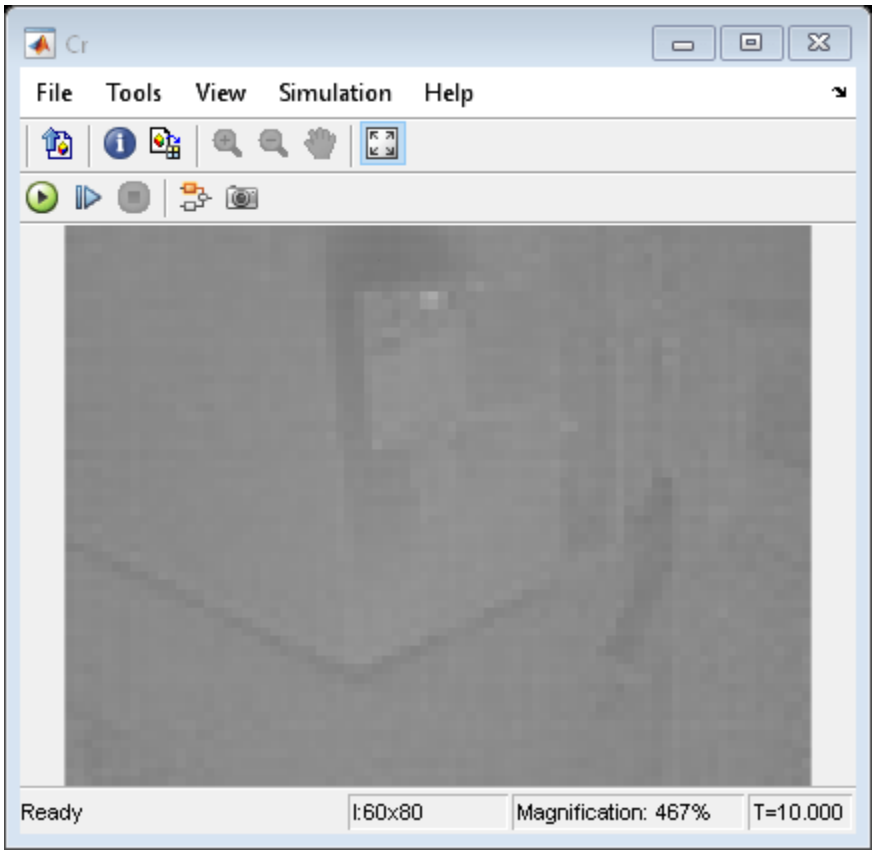
The model reads the binary file specified as `'vipmen.bin'` in the `File name` parameter of the `Read Binary File` block. The file is played until the end of the simulation because the `Number of times to play file` parameter of the `Read Binary File` block is set to `inf`.

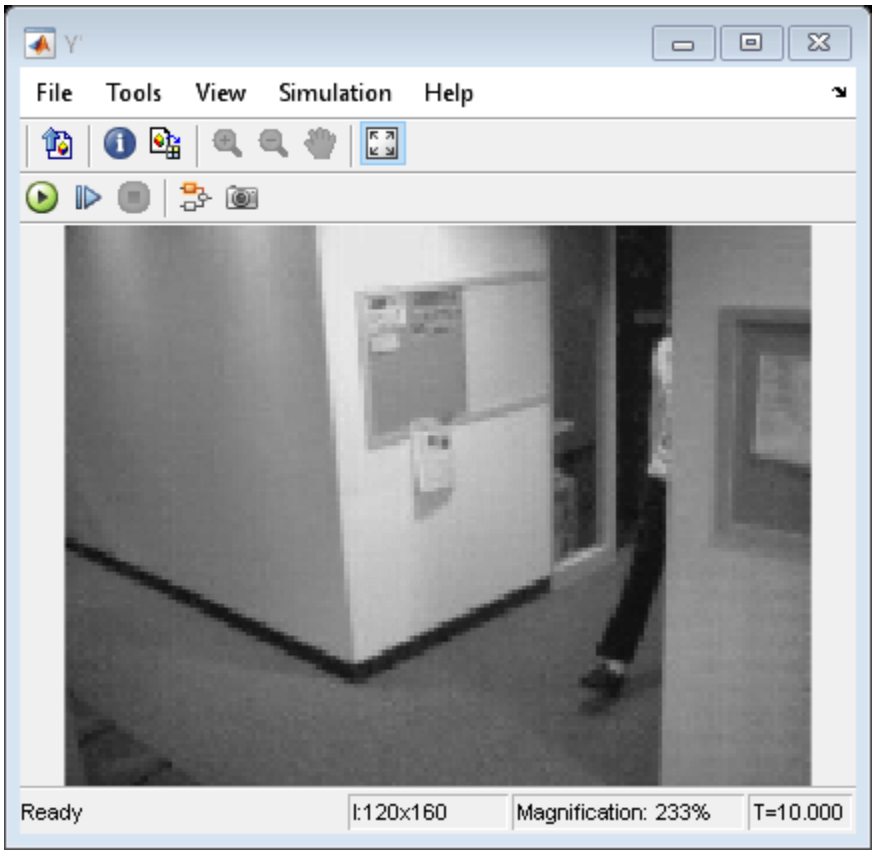
Simulate and Display Results

Run the simulation.

```
sim(modelname);
```


Cb
File
Tools
View
Simulation
Help
Ready
I:60x80
Magnification: 470%
T=10.000


Cr
File
Tools
View
Simulation
Help
Ready
I:60x80
Magnification: 467%
T=10.000



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');
```

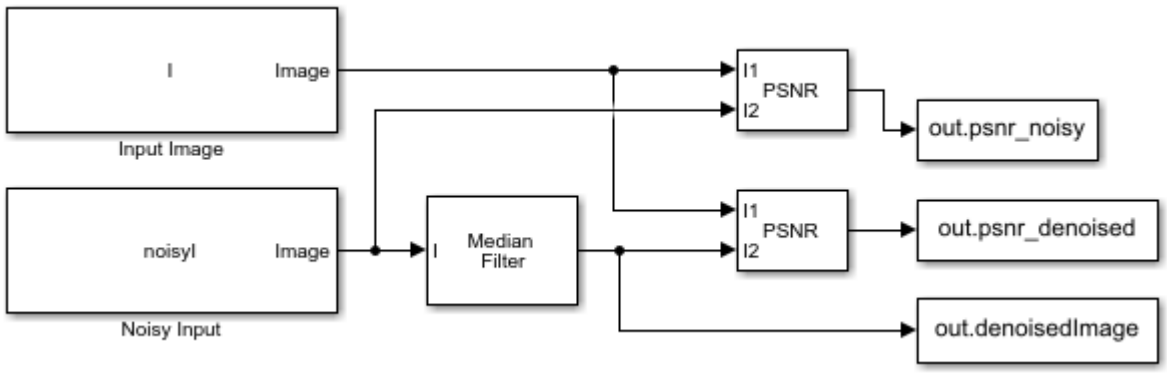
Read the corresponding noisy image into the MATLAB workspace.

```
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.

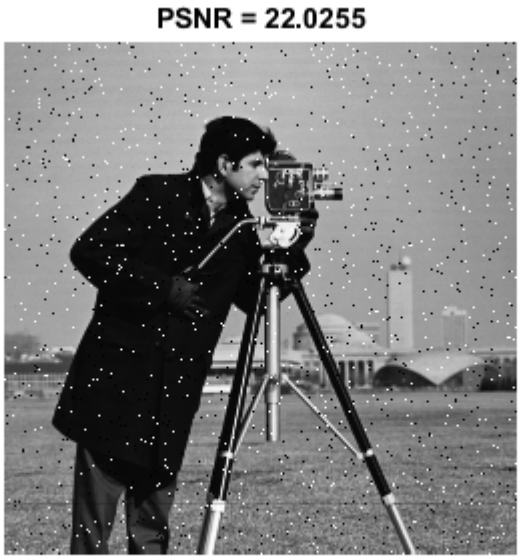
Simulate and Display Results

Simulate the model.

```
out = sim(modelname);
```

Display the noisy image and the corresponding PSNR value

```
imshow(noisyI,[]);
title(['PSNR = ', num2str(out.psnr_noisy)]);
```



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)]);
```

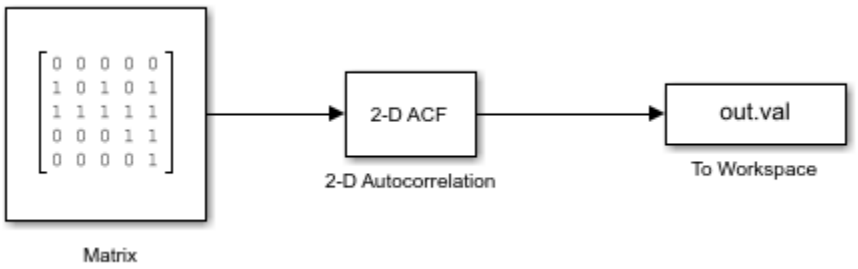
Compute Autocorrelation of Input Matrix

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
```

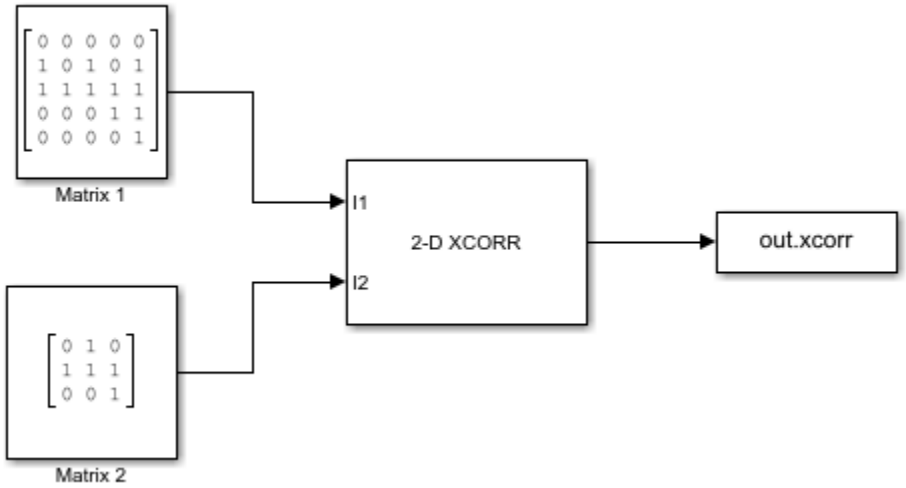
Compute Correlation between Two Matrices

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.

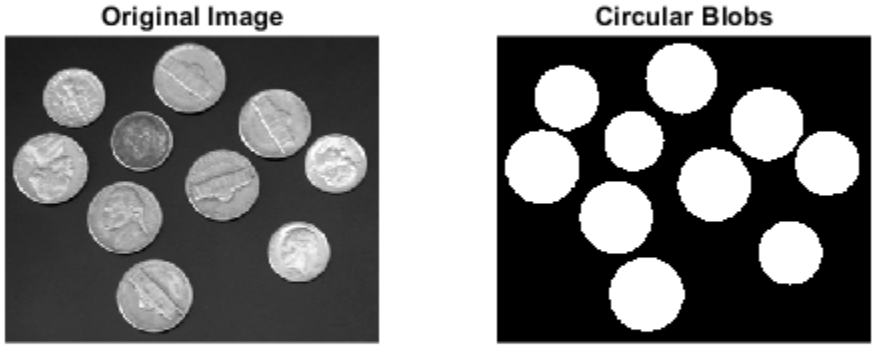
```
I = imread('coins.png');
```

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.

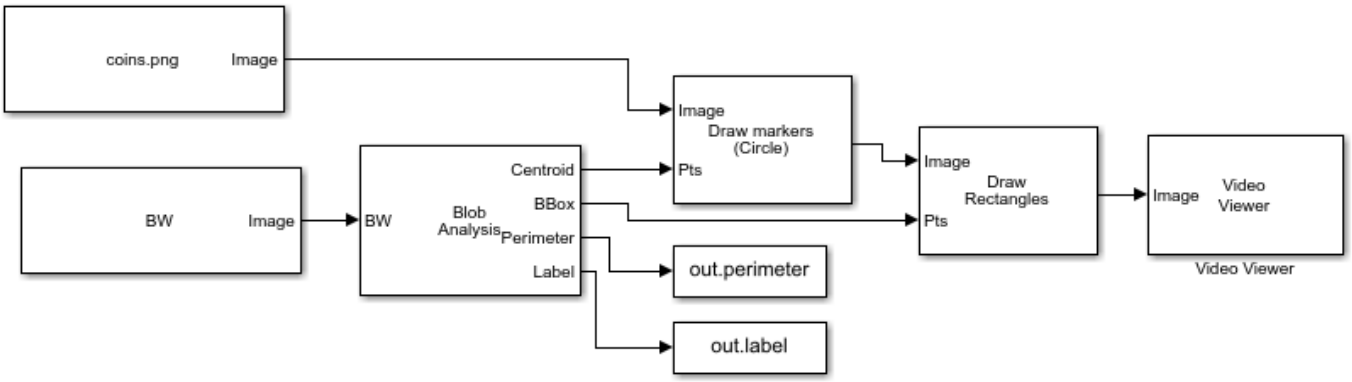
```
figure
subplot(1,2,1)
imshow(I,[]);
title('Original Image');
subplot(1,2,2)
imshow(BW)
title('Circular Blobs');
```



Example Model

Open the simulink model.

```
open_system('ex_blkblobanalysis.slx')
```



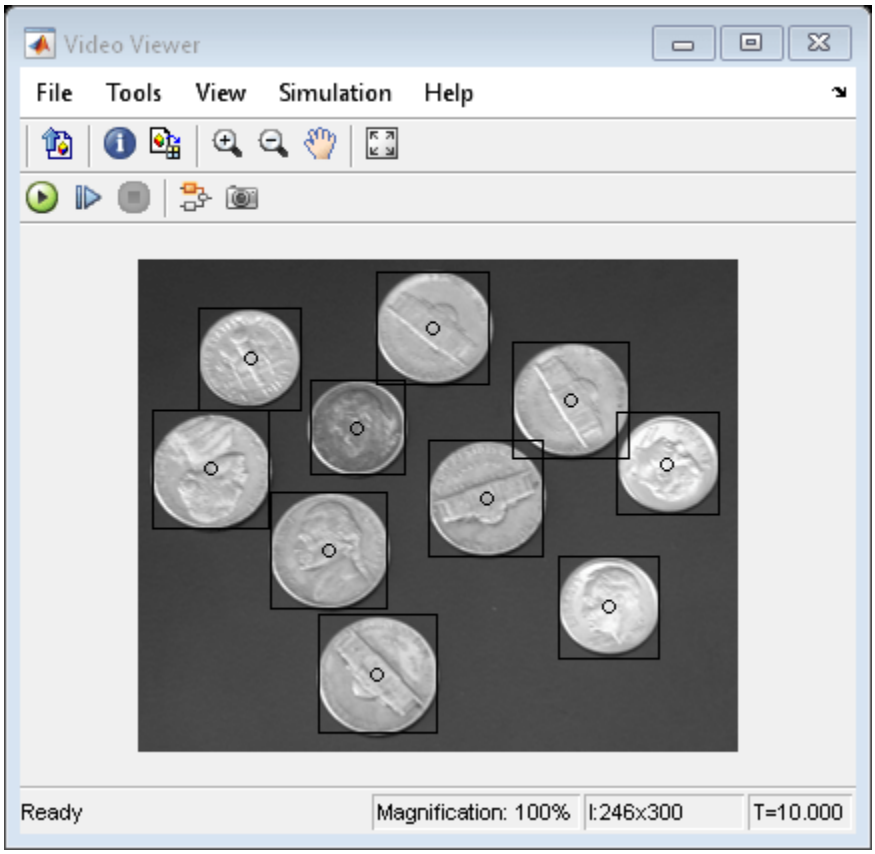
The model computes the centroid, perimeter, and bounding box coordinates for the blobs in the binary image. The computed statistics is overlayed on the input image by using the `Draw Markers` and `Draw Rectangles` blocks.

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 overlayed 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.

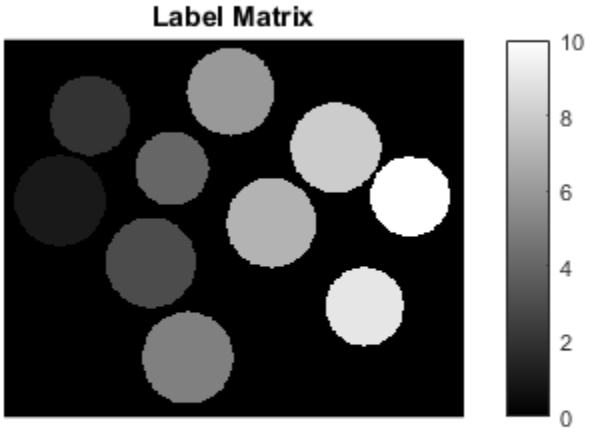
```
table(lb(2:end),out.perimeter,'VariableNames',{'Label','Perimeter'})
```

```
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
```

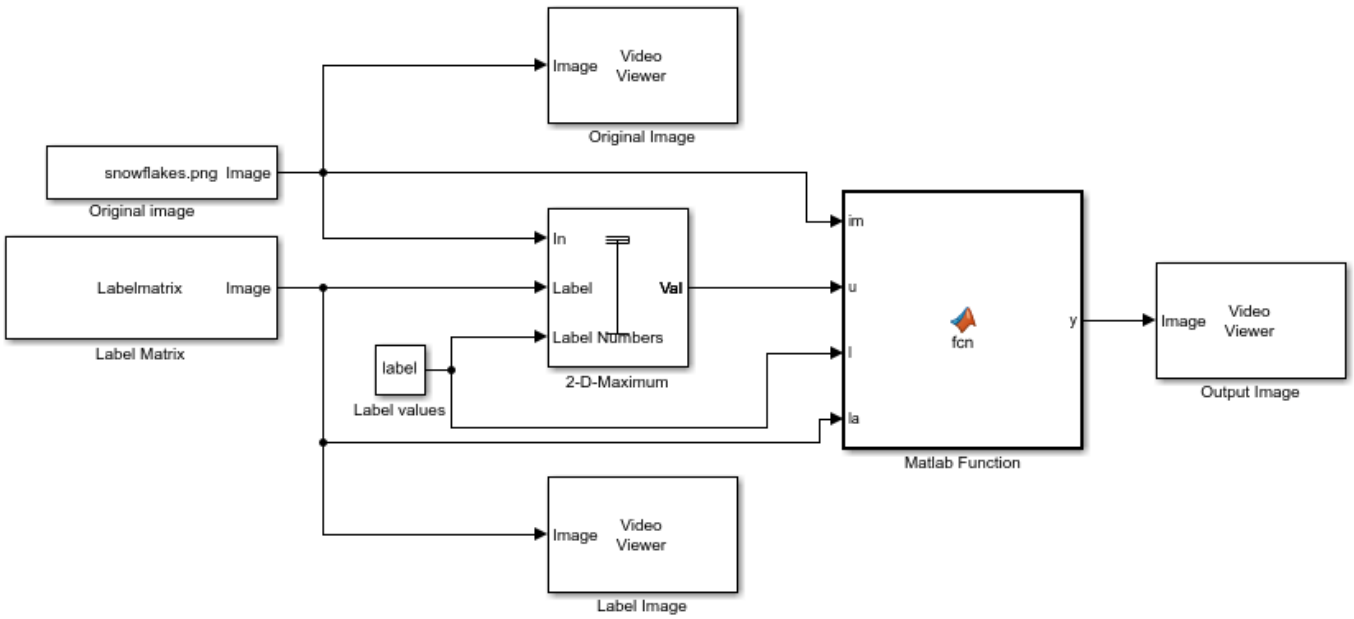
Find the unique label values in the label matrix. The label value 0 corresponds to the background and must be ignored.

```
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,

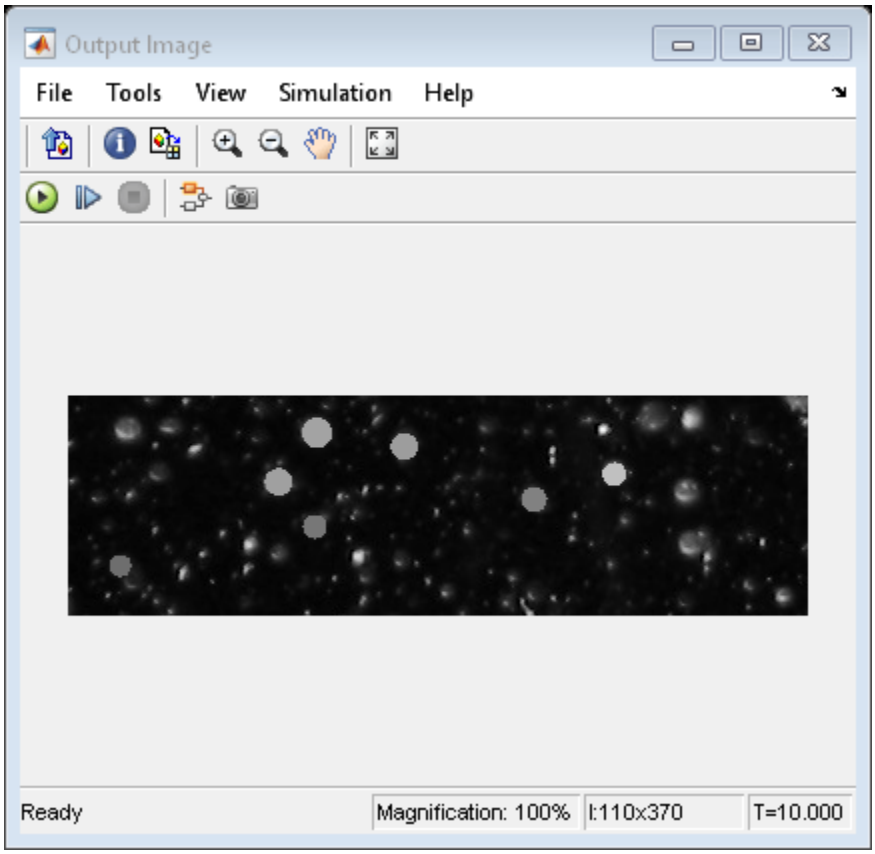
- Set the `Mode` parameter to `Value`.
- Set the `Find the maximum value over` parameter to `Entire input`.
- Set the `Enable ROI processing` parameter and set the `ROI type` parameter as `Label matrix`.

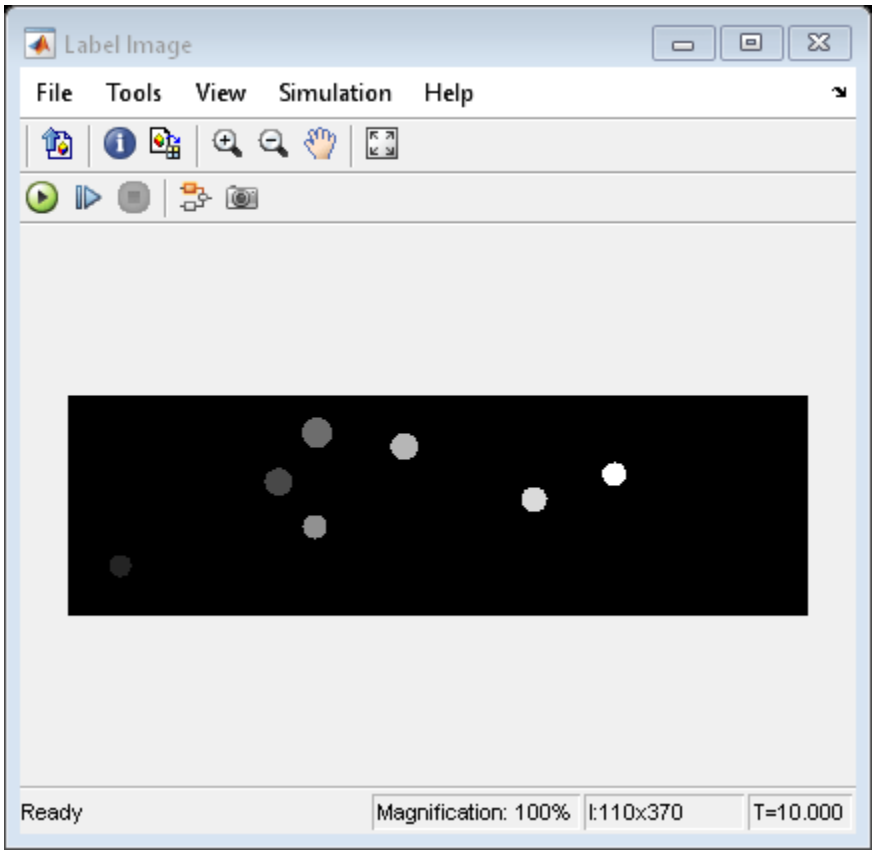
The `Matlab Function` block replaces the individual ROI's with its maximum intensity value and outputs the resultant image.

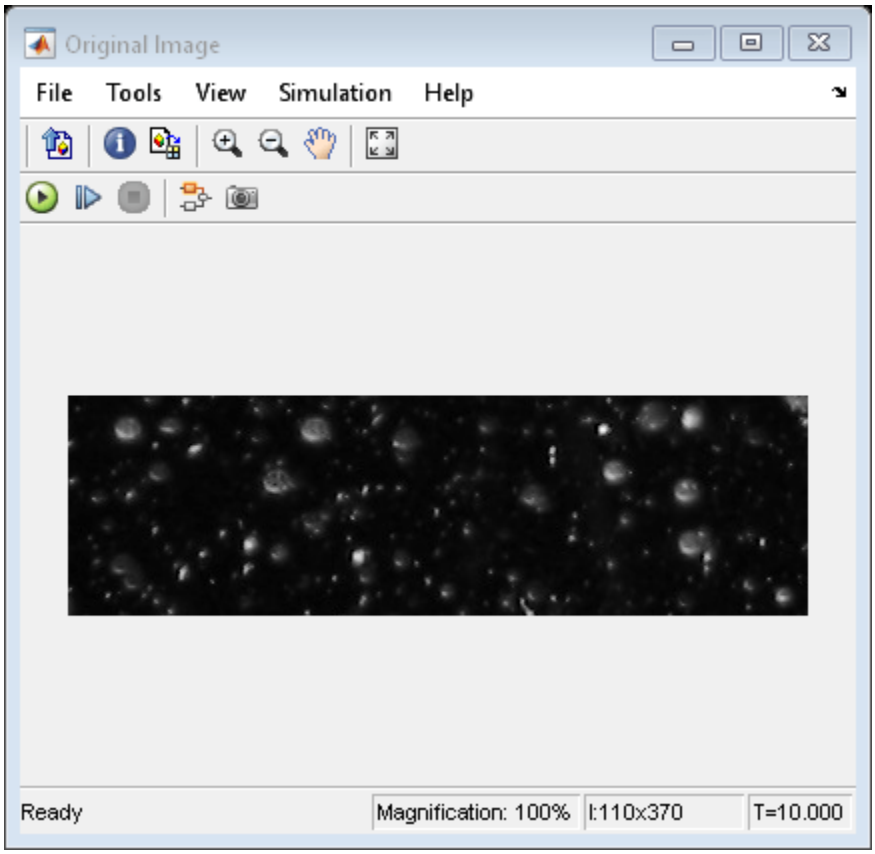
Simulate and Display Results

Run the model and display the images using `Video Viewer` block.

```
sim('ex_blk2dmaximum.slx');
```




Label Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:110x370
T=10.000


Original Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:110x370
T=10.000

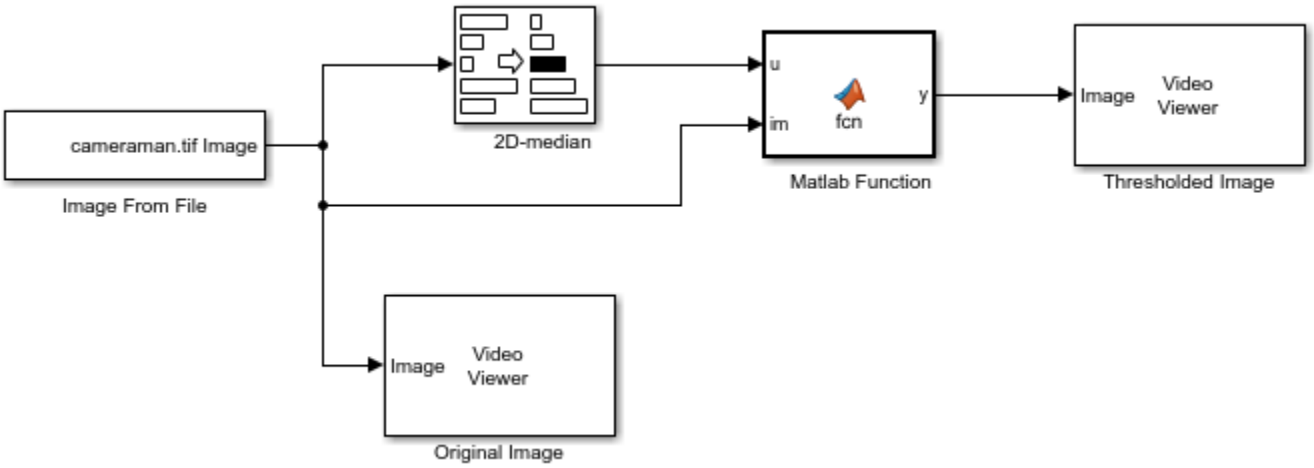
Median based Image Thresholding

This example shows how to perfom image thresholding with the median value of the image as a global threshold.

Example Model

Open the simulink model.

```
open_system('ex_blk2dmedian.slx');
```

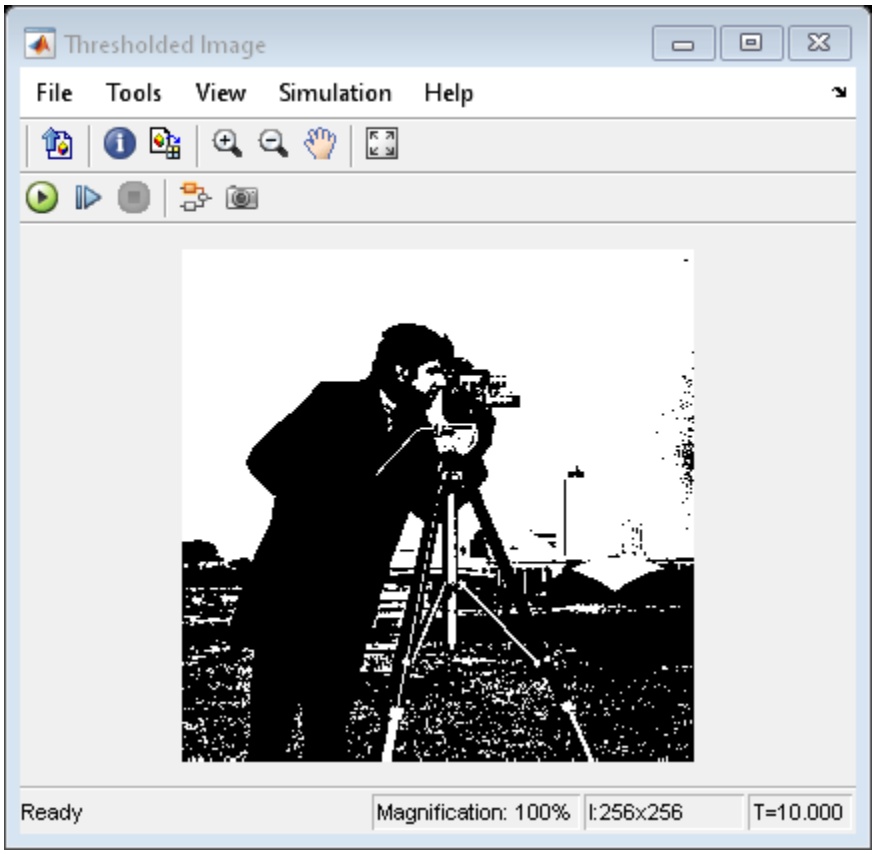


The model computes the median value of an input image by setting the `Find the median value over` parameter to `Entire input` in `2-D Median` block. The output of the `2-D Median` block is a scalar. The `Matlab Function` block performs image thresholding by taking the output median as a global threshold. 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');
```


Thresholded Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=10.000


Original Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=10.000

Import Image From MATLAB Workspace

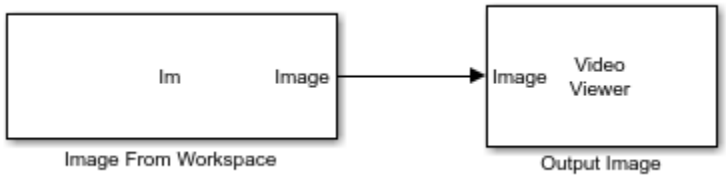
This example shows how to import an image from MATLAB to Simulink workspace using `Image From Workspace` block.

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')
```

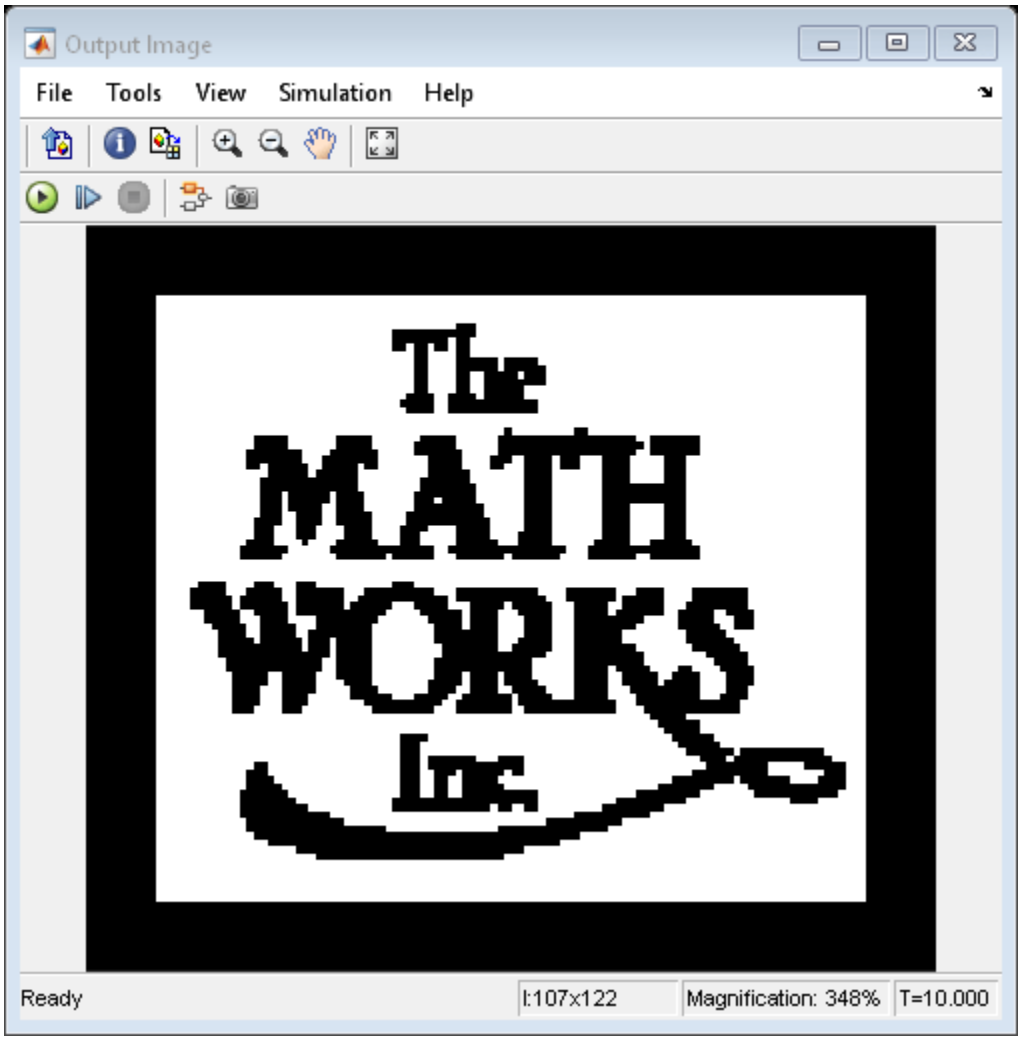
Open the Simulink model.

```
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');
```


Output Image
File Tools View Simulation Help
The MATH WORKS Inc.
Ready
I:107x122
Magnification: 348%
T=10.000

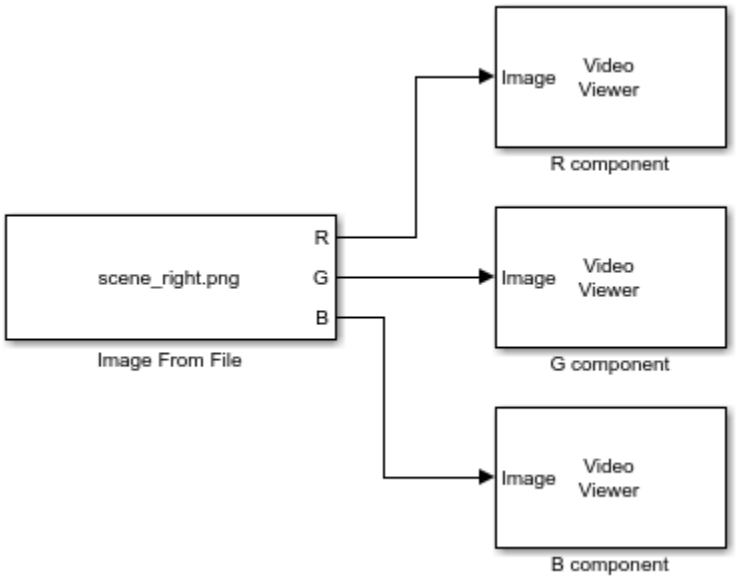
Import Image from Specified Location

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.

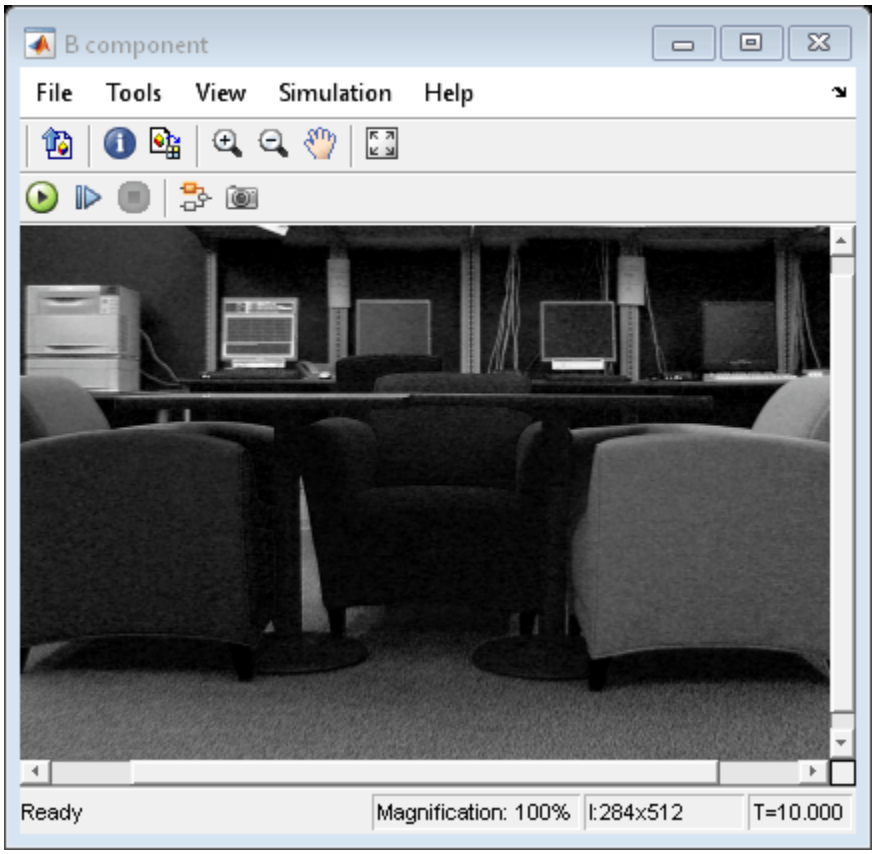
```
open('ex_readcolorimage');
```

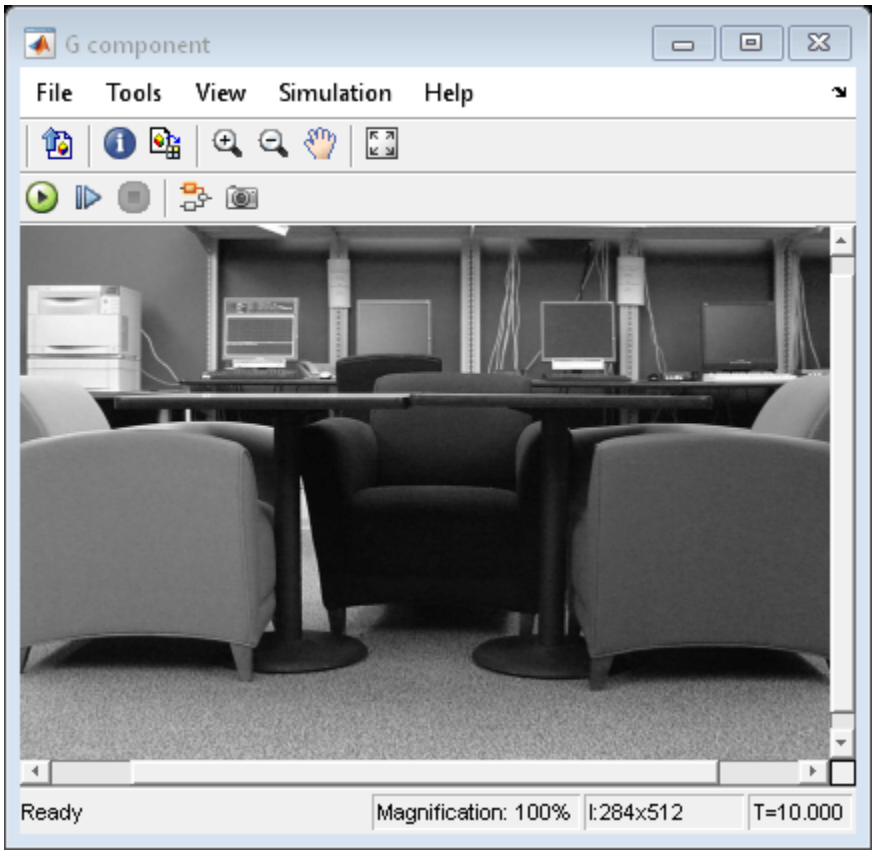


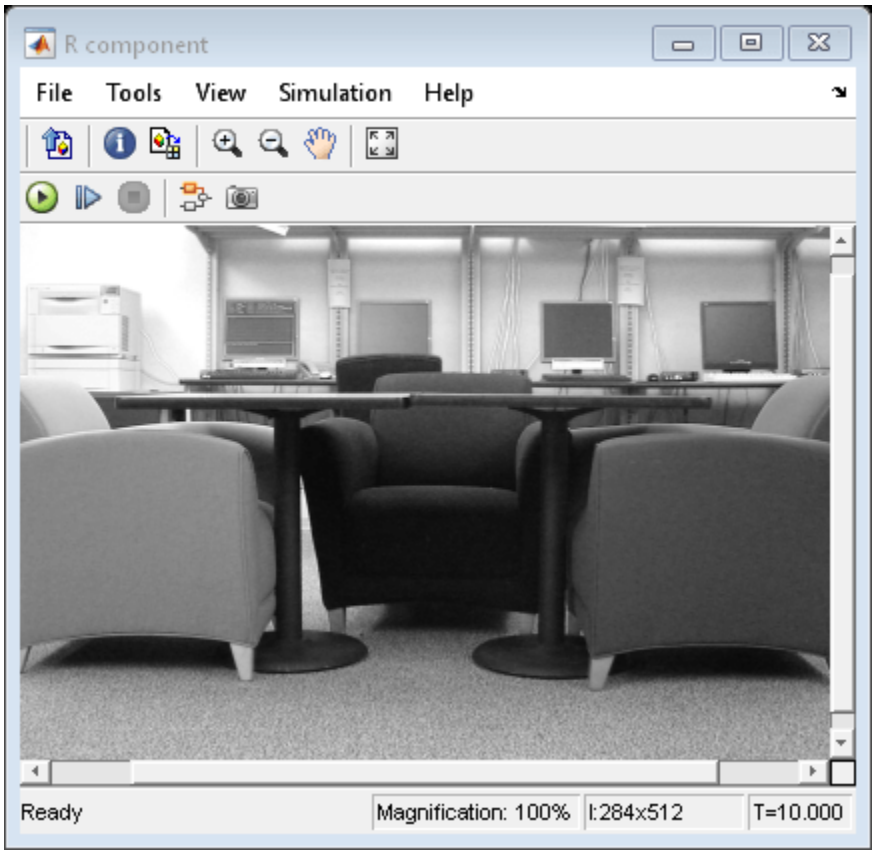
The model imports a color image to the Simulink workspace and displays the R, G, and B color components of the image by using the `Video Viewer` block. The **Image Signal** parameter of the `Image From File` block is set to `Separate color signals` inorder to import the RGB color components separately.

Simulate and Display Results

```
sim('ex_readcolorimage');
```


B component
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:284x512
T=10.000


G component
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:284x512
T=10.000


R component
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:284x512
T=10.000

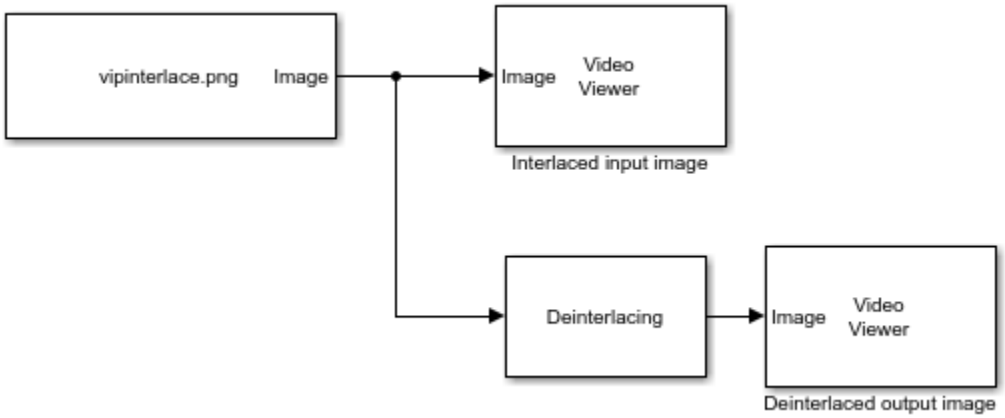
Remove Interlacing Effect From Image

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');
```

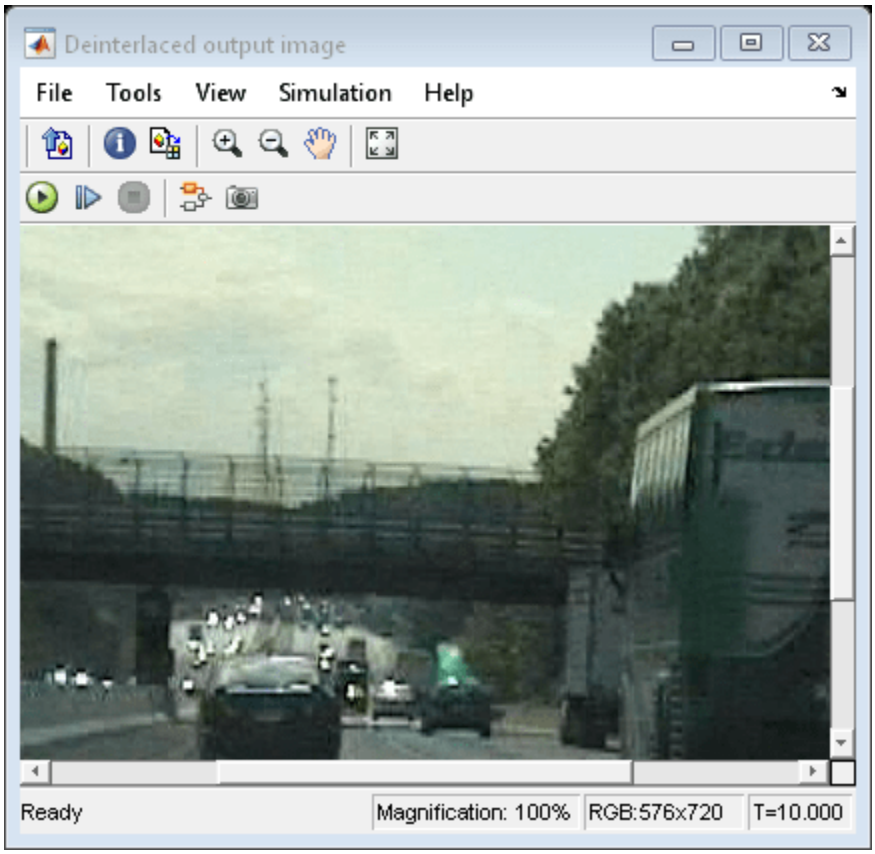


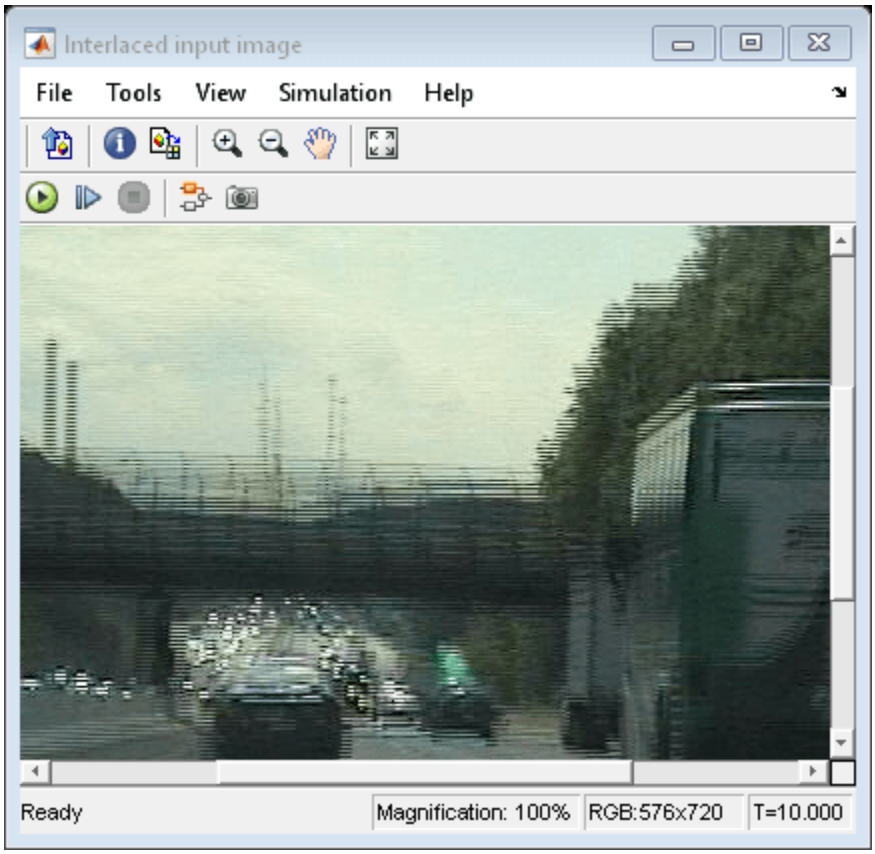
The model uses the `Image From File` block to read an interlaced image from a specified file location. The `Method` parameter of the `Deinterlacing` block is set to `Linear interpolation`. Hence, the model converts the interlaced image at the input into a deinterlaced image by using the linear interpolation 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');
```


Deinterlaced output image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
RGB:576x720
T=10.000



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.

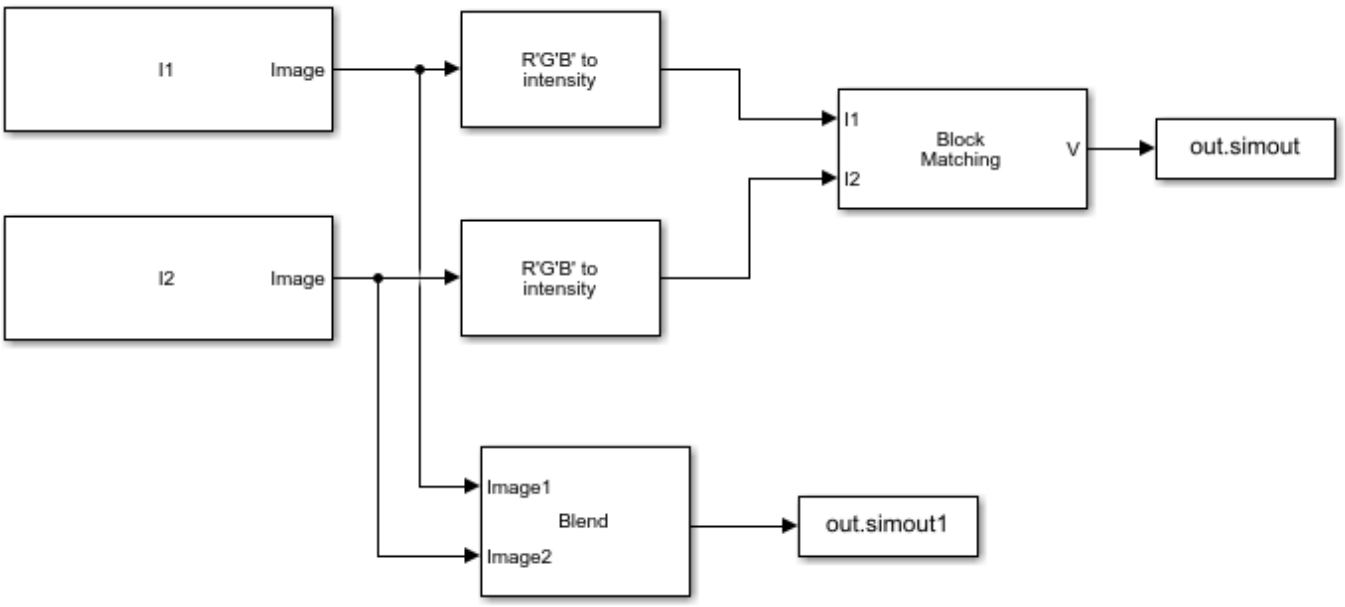
Read image frames for which motion has to be estimated.

```
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 overlayed image.

```
out = sim('ex_blockmatching.slx');
```

Display Results

Read the output motion vector and the overlayed 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 overlayed 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);
```

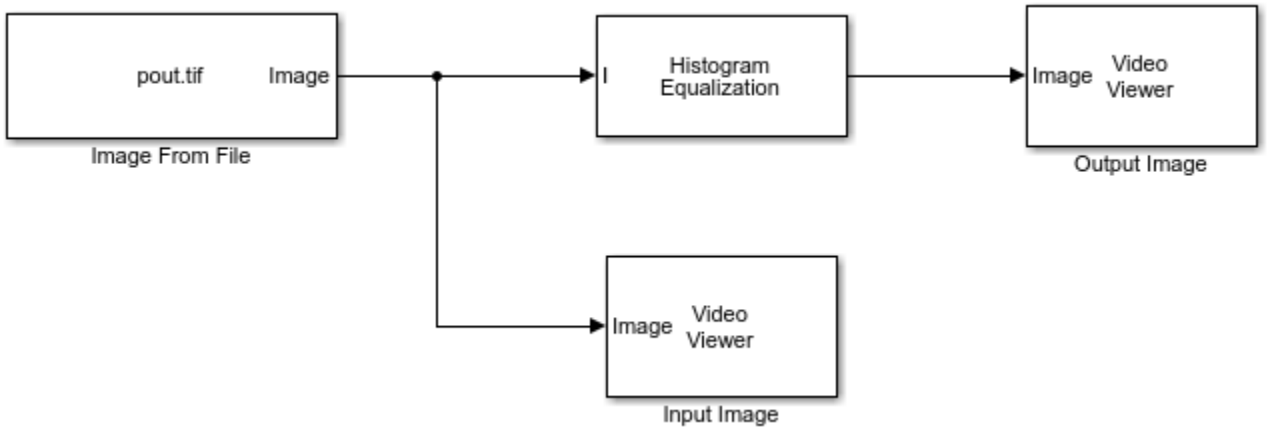
Enhance Contrast of Grayscale Image Using Histogram Equalization

This example shows how to enhance the contrast of a grayscale image using the Histogram Equalization block.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkHistEq.slx';
open_system(modelname)
```



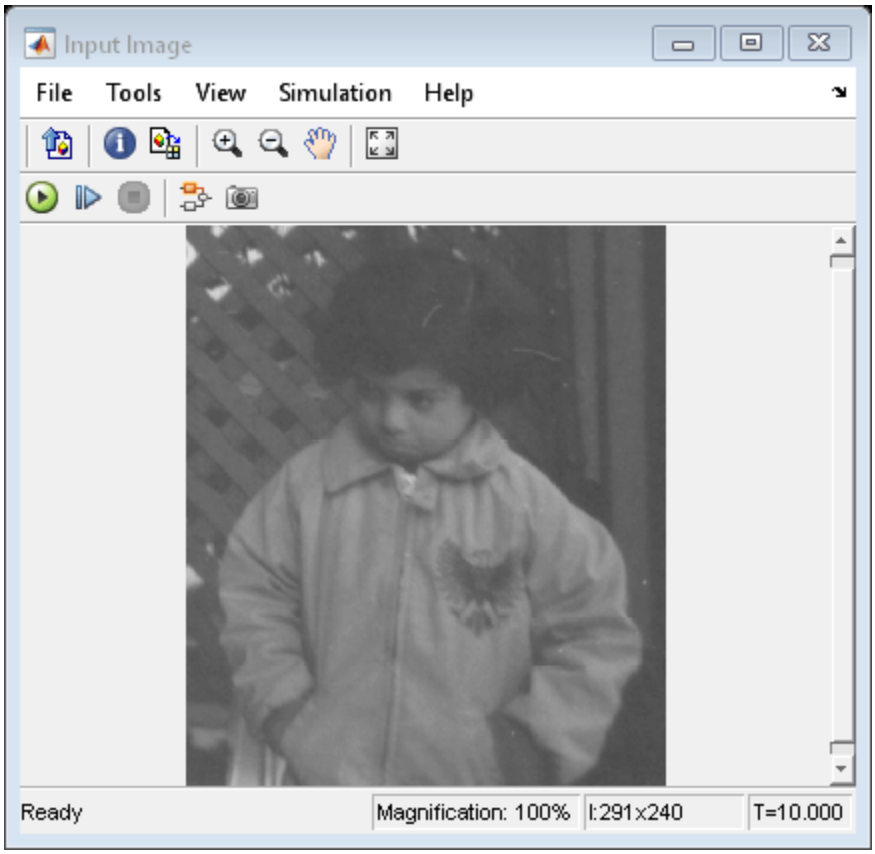
The model reads an input image using the Image From File block. The input image in this example has very low contrast. To enhance the contrast of the input image, the model uses the Histogram Equalization block with these default parameters:

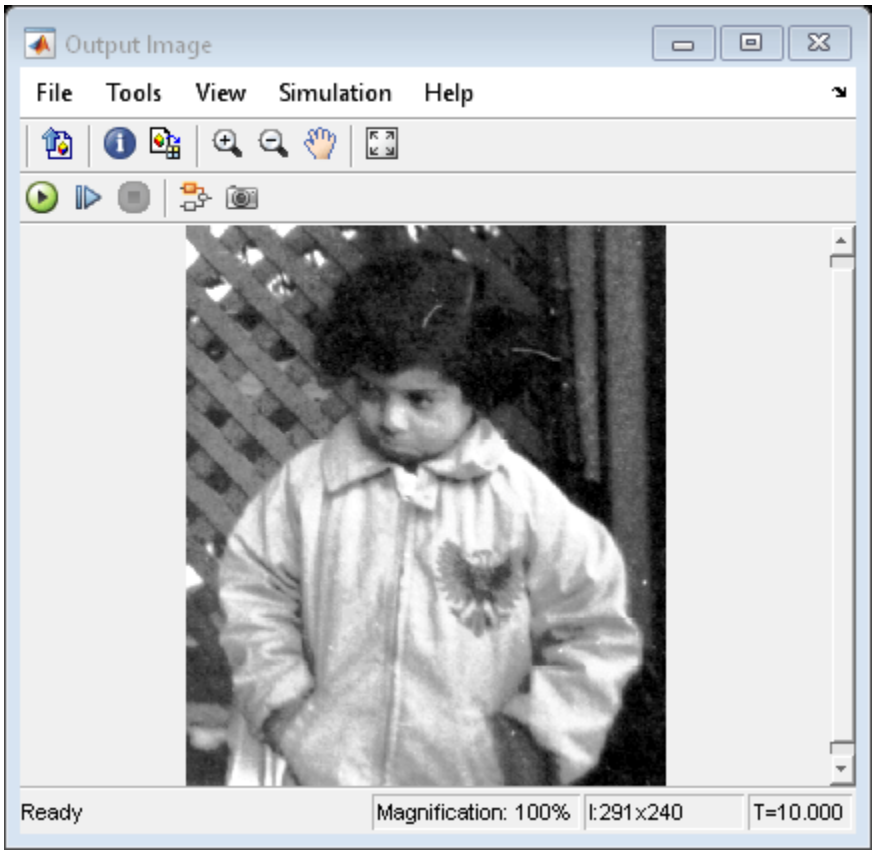
- **Target Histogram**: `Uniform`
- **Number of bins**: 64

Simulate and Display Results

Run the model and verify the change in contrast from the input image to the output image. For comparison, the model displays the input image and the output image using Video Viewer blocks.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:291x240
T=10.000


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:291x240
T=10.000

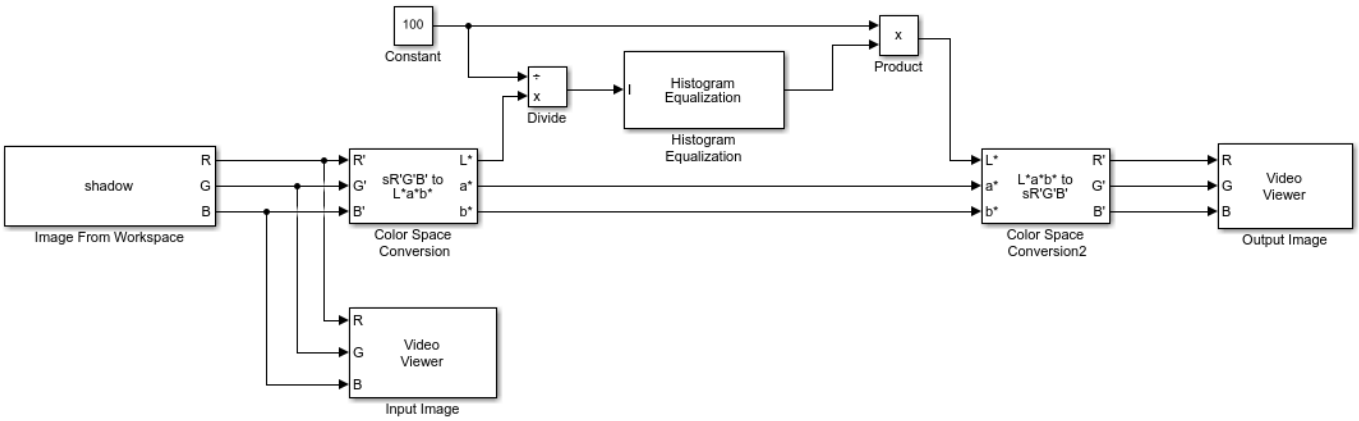
Enhance Contrast of Color Image Using Histogram Equalization

This example shows how to enhance the contrast of a color image using the Histogram Equalization block.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkHistEqColor.slx';
open_system(modelname)
```



Initialize Parameters

The model uses the initialization function callback `InitFcn` to initialize the input parameters. To configure the `InitFcn`, on the **Modeling** tab, click **Model Settings** and select **Model Properties**. In the Model Properties dialog box, click the **Callbacks** tab and select `InitFcn` from the list.

The `InitFcn` of this model executes the following code to read an indexed image and convert it to an RGB image.

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

This model reads the input image from workspace using the Image From Workspace block with these block parameters:

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

Enhance Contrast of Image

The model extracts luminance information from the color image by converting it from the RGB color space to L*a*b* color space, using the Color Space Conversion block with these parameters:

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

The values of the luminance parameter `L*` range from 0 to 100. The Histogram Equalization block expects floating point input values in the range from [0, 1]. The model normalizes the luminance

values of the image by using a Constant block with the **Constant value** parameter set to `100` and a Divide block with default parameters.

To enhance the contrast of the image, the model uses a Histogram Equalization block with default parameters. The block transforms the normalized luminance values in the input color image such that the histogram of the output image is approximately uniform.

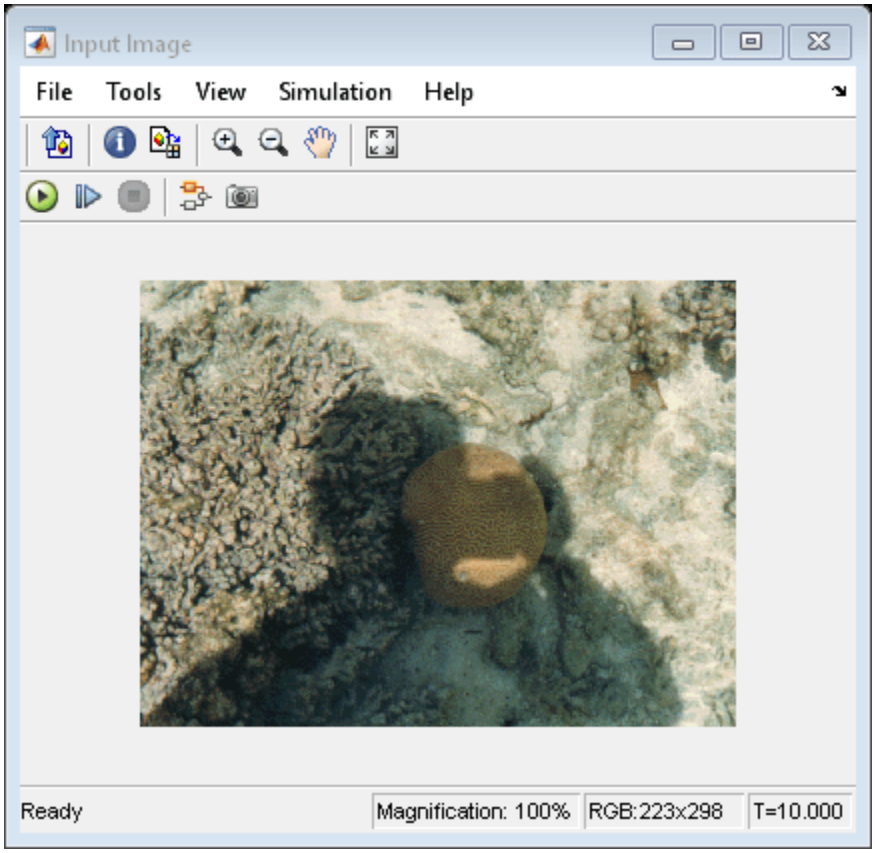
The model uses a Product block with default parameters and the existing Constant block to scale the luminance of the histogram-equalized output image to a range of 0 to 100. The model then converts the output image from the L*a*b* color space to the RGB color space using a Color Space Conversion block with these parameters:

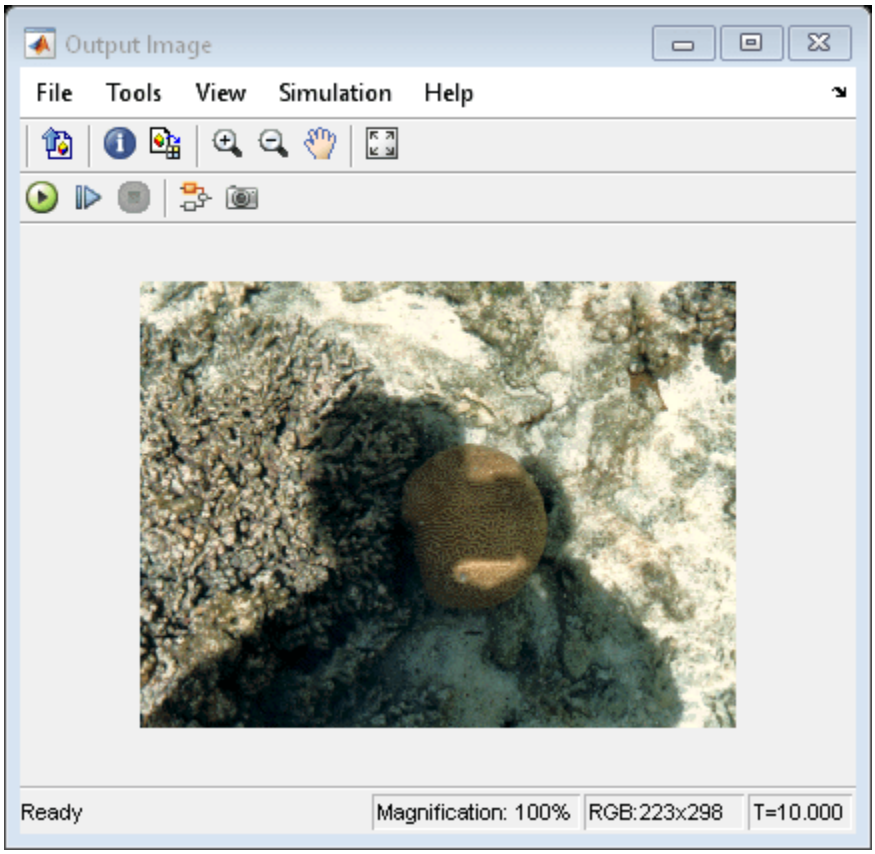
- **Conversion**: `L*a*b* to sR'G'B'`
- **Image signal**: `Separate color signals`

Simulate and Display Results

Run the model and verify the change in contrast from input image to the output image. For comparison, the model displays the input image and the output image using Video Viewer blocks. The **Image signal** parameter of each Video Viewer block is set to `Separate Color Signals` .

```
sim(modelname);
```




Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
RGB: 223x298
T=10.000

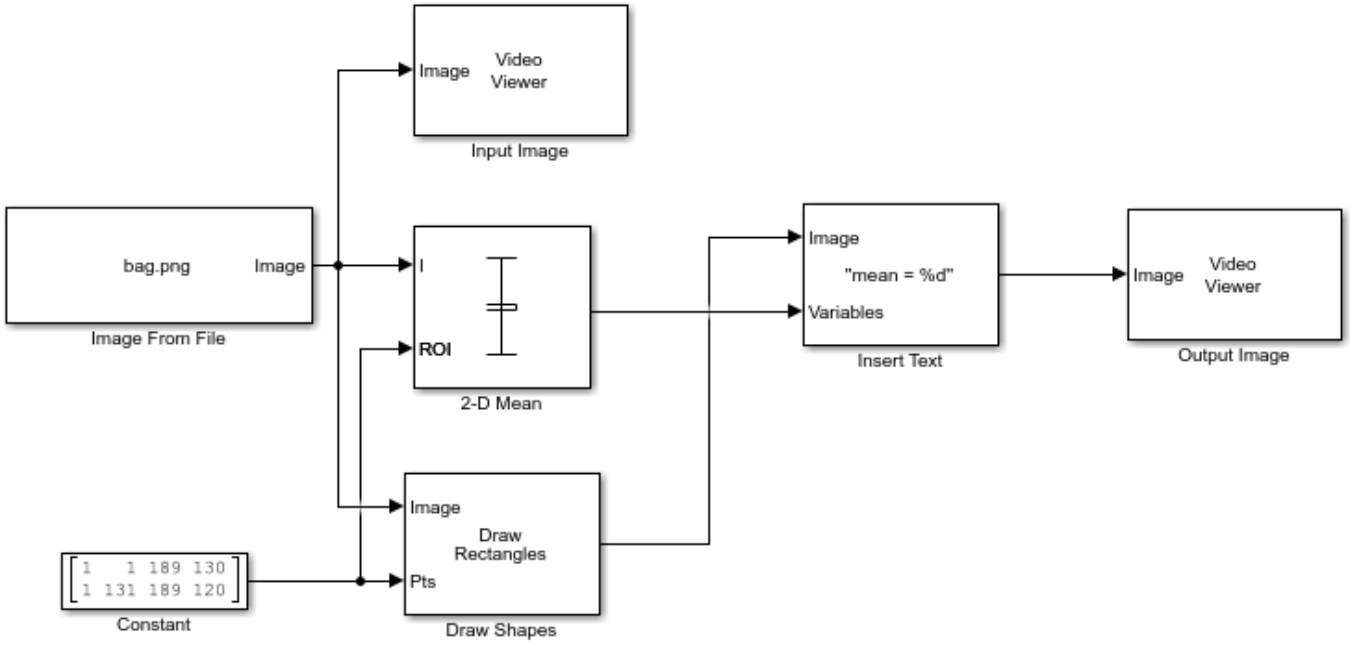
Compute Mean of ROIs in Image

This example shows how to compute the mean of regions of interest (ROIs) in an input image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blk2dmean.slx';
open_system(modelname)
```



The model reads an input image using the Image From File source block. The input image in the example has two different texture regions. To compute the mean of these regions of interest, the model uses a 2-D Mean block with these parameters:

- **Running mean**: `off`
- **Find the mean value over**: `Entire input`
- **Enable ROI processing**: `Rectangles`
- **ROI portion to process**: `Entire ROI`
- **Output**: `Individual statistics for each ROI`

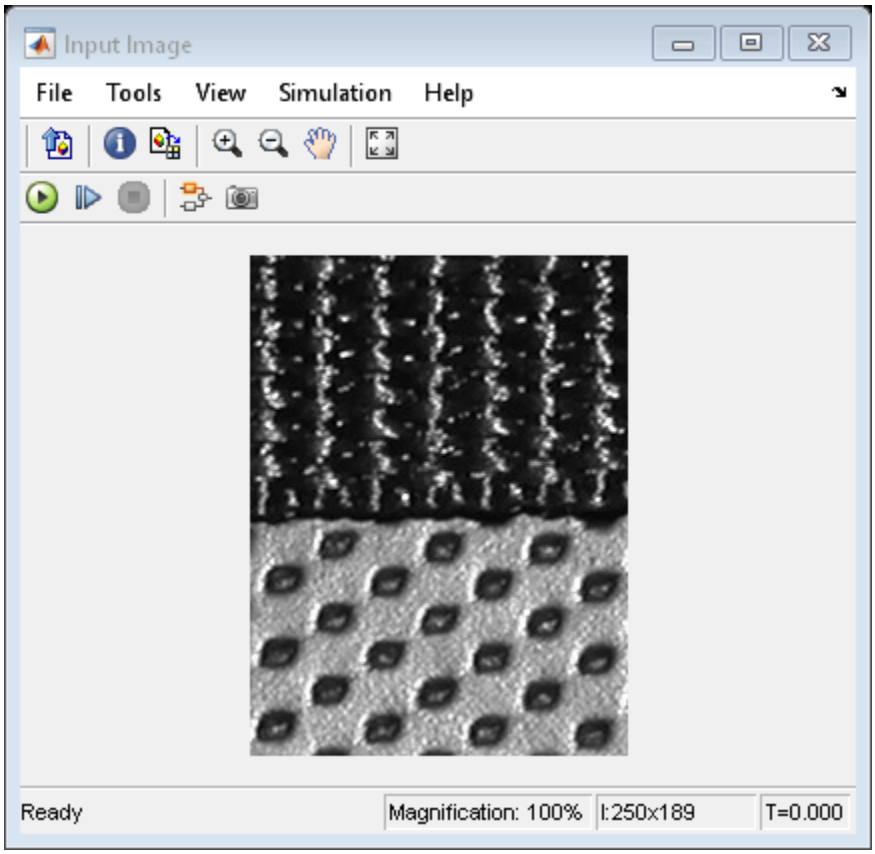
The Constant block specifies the coordinates and dimensions for two rectangular ROIs. The Draw Shapes block overlays the bounding boxes for each rectangular ROI onto the output image. The model also displays the computed mean values in the output image using an Insert Text block.

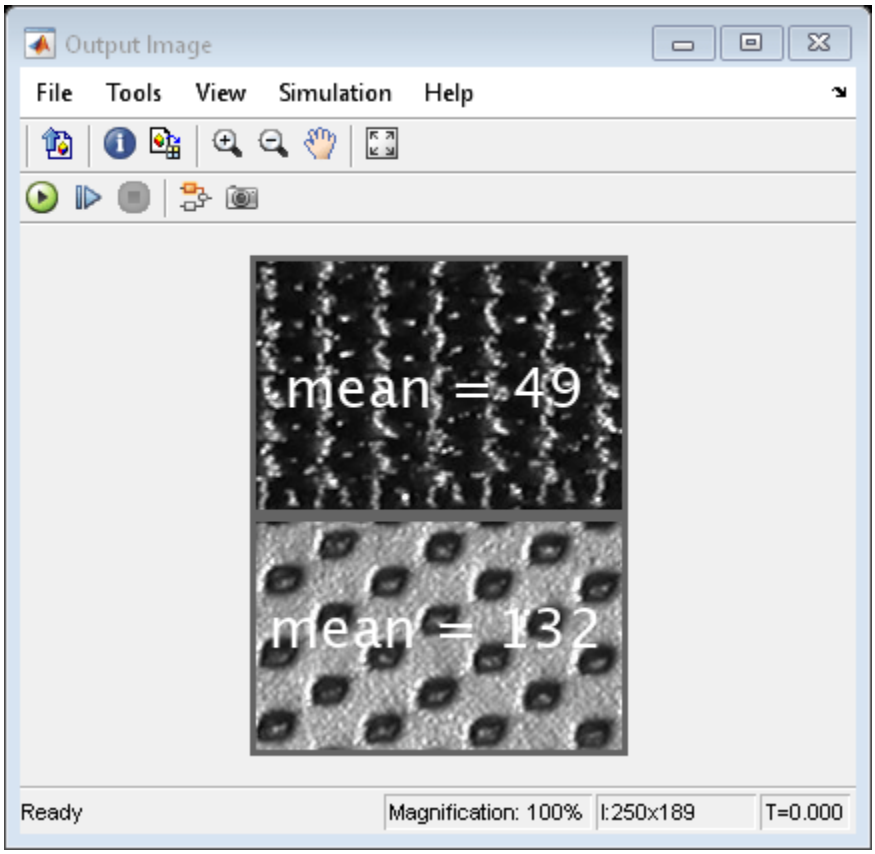
Simulate and Display Results

Run the model. The model displays both the input image and output image using Video Viewer blocks. Verify that the ROI around the darker region of the image shows a lower mean value than the ROI

around the brighter region. The lower mean value indicates that the darker region has a lower average intensity value than the brighter region.

```
sim(modelname);
```




Output Image
File
Tools
View
Simulation
Help
mean = 49
mean = 132
Ready
Magnification: 100%
I:250x189
T=0.000

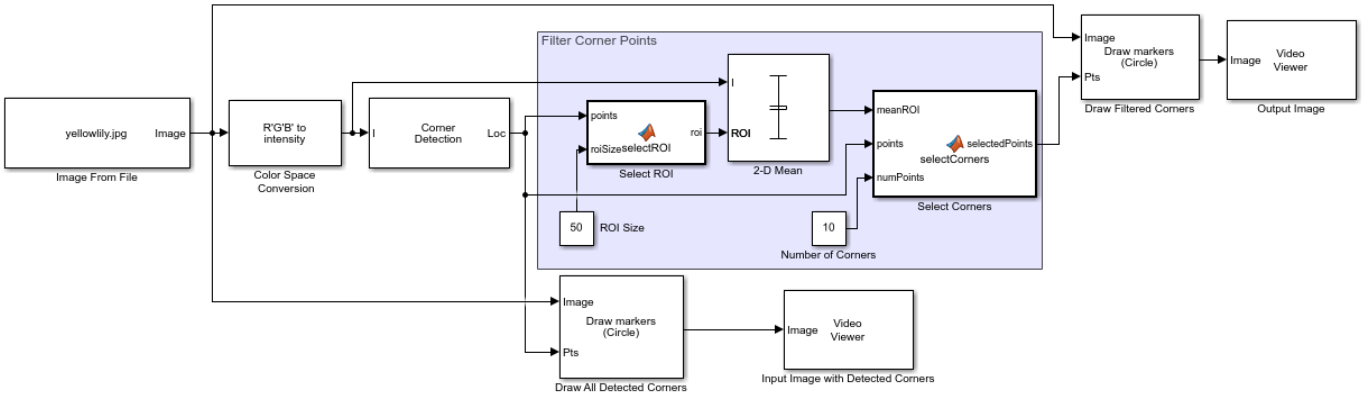
Detect Corners in Image

This example shows how to detect corners in an image using the Corner Detection block.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkCornerDetection.slx';
open_system(modelname)
```



Read Input Image

The model reads an RGB input image using the Image From File block and converts it to an intensity image using the Color Space Conversion block.

Detect and Visualize Corners

To detect corners in the input image, the model uses the Corner Detection block. The block uses the Harris corner detection method, and the value of the **Maximum number of corners** parameter is `25`.

The model uses a Draw Markers block and a Video Viewer block to display all the corners detected in the input image.

Filter Detected Corners

The displayed image shows some spurious corners within the dark background of the image. The model uses other blocks in the Filter Corner Points area to remove these spurious corners and retain the corners of the flower only. The Filter Corner Points area selects fixed-size, square regions of interest (ROIs) around each detected corner point, and then finds the mean value of each ROI. The mean values of ROIs around corners detected in the dark background are lower than the other ROIs. To filter these spurious corners, the model selects the desired number of corners from those corresponding to the ROIs with the highest mean values. The Filter Corner Points area includes these blocks:

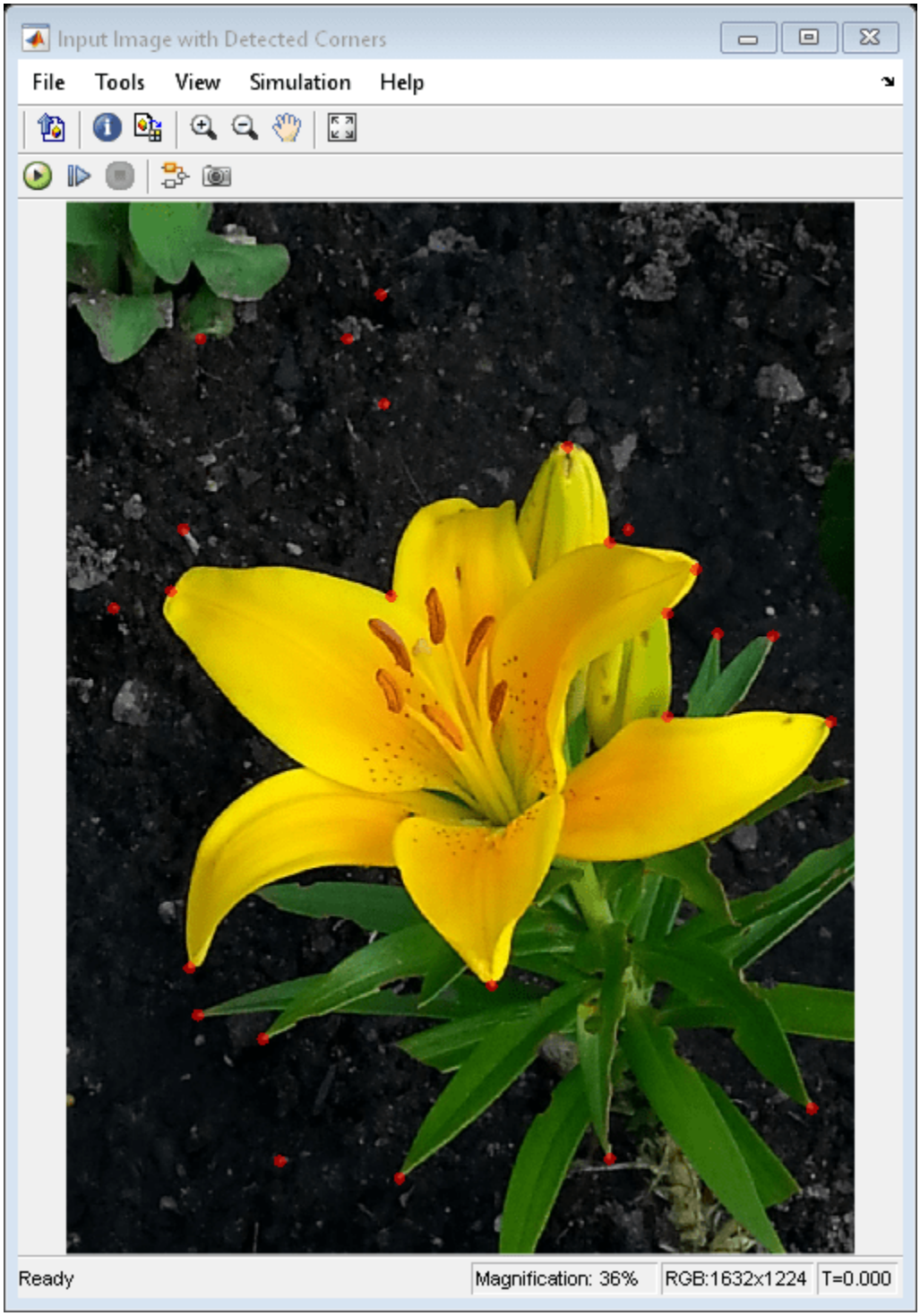
- Select ROI — This MATLAB Function (Simulink) block generates the specifications of the square ROIs around each detected corner. The specifications include the coordinates of the upper-left corner and the width and height parameters of each ROI in the form `[ x y width height ]`. Since each ROI is square, the values of the width and the height parameters are equal. The model uses a Constant (Simulink) block to set the width of each ROI to `50`.

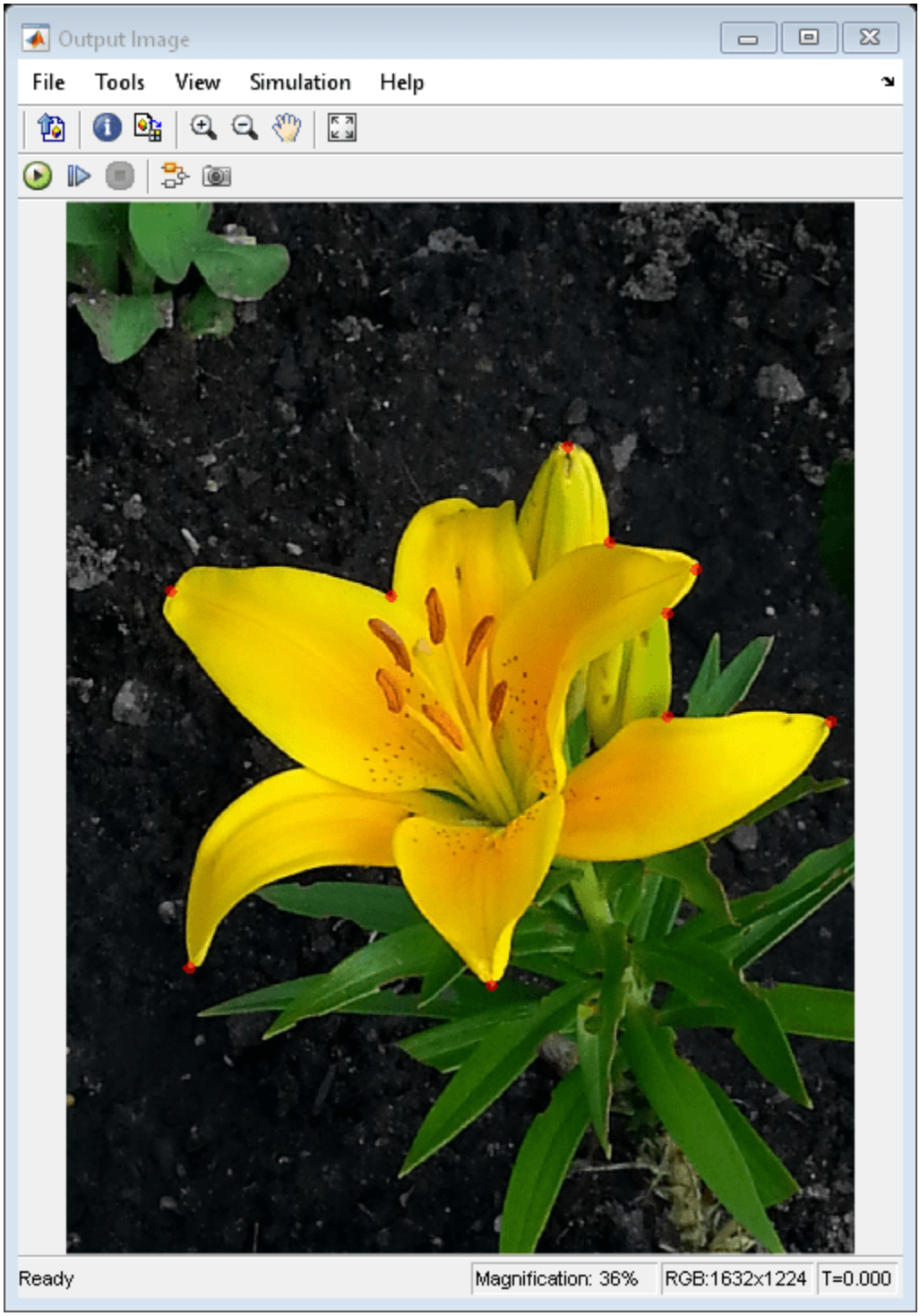
- 2-D Mean — This block computes the mean value of each selected ROI in input image.
- Select Corners — This MATLAB Function (Simulink) block selects the desired number of corner points corresponding to the ROIs with the highest mean values. The model uses a Constant (Simulink) block to set the desired number of corner points to `10`.

Simulate and Display Results

To display the filtered corner points in the image, the model uses a Draw Markers block and a Video Viewer block. Simulate the model and display the results.

```
sim(modelname);
```


Input Image with Detected Corners
File
Tools
View
Simulation
Help
Ready
Magnification: 36%
RGB:1632x1224
T=0.000


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 36%
RGB:1632x1224
T=0.000

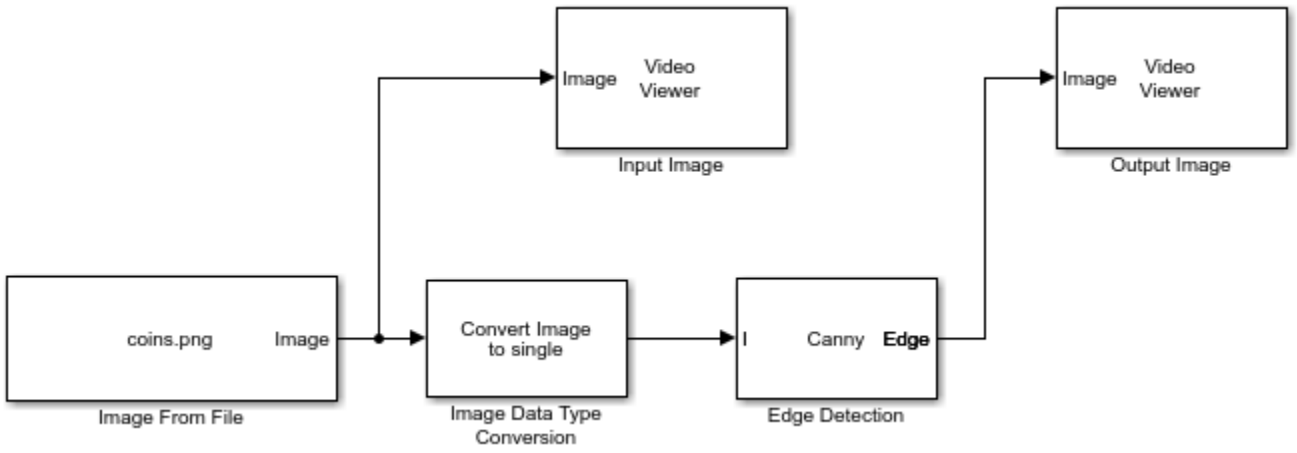
Edge Detection of Intensity Image

This example shows how to detect edges in an intensity image using the Canny edge detector.

Example Model

Open the Simulink® model.

```
modelname = 'ex_EdgeDetection.slx';
open_system(modelname)
```



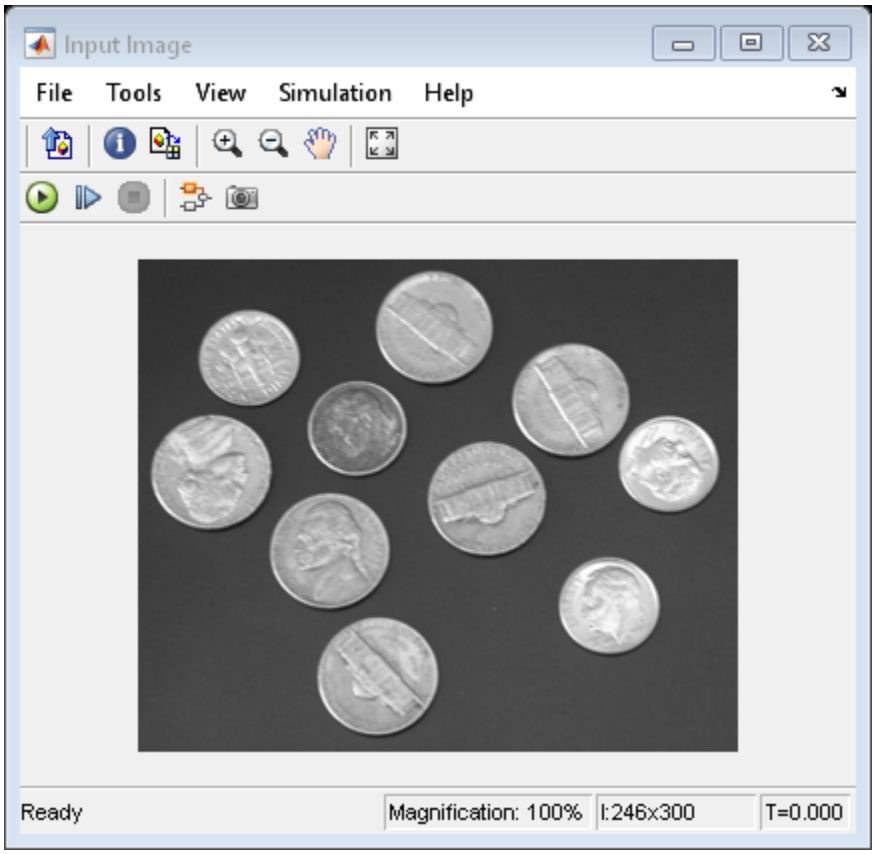
The model reads an input image using the Image From File block and converts the image data type to `single` using the Image Data Type Conversion block. The model then detects the edges in the input image using the Edge Detection block with these parameter values:

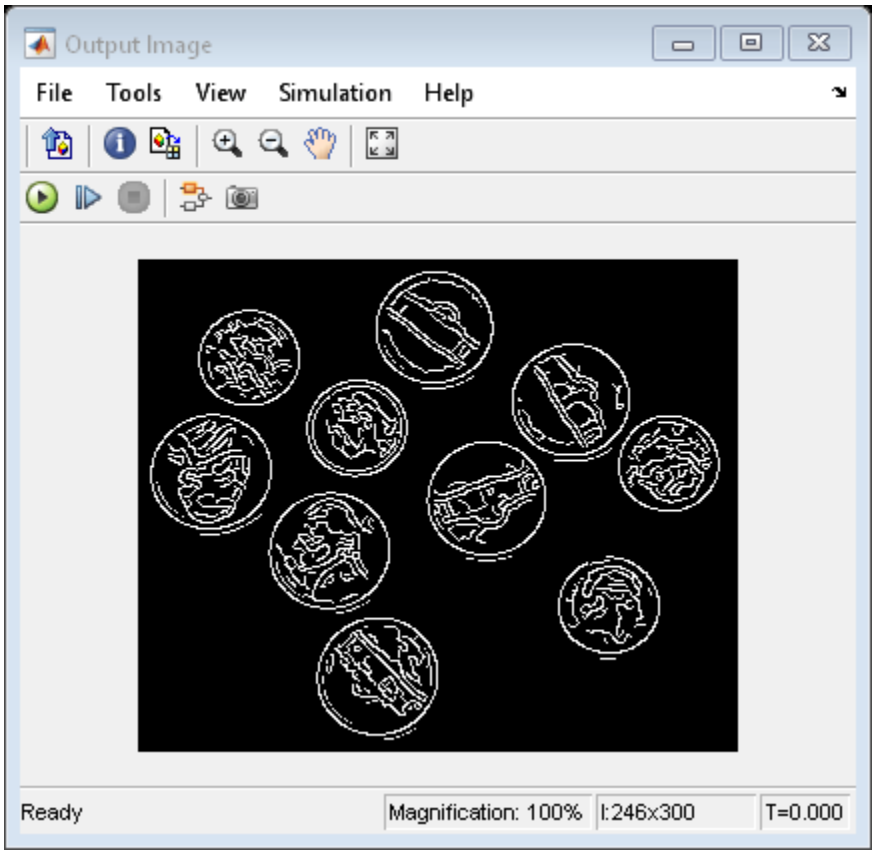
- **Method** — `Canny`
- **User-defined threshold** — `off`
- **Approximate percentage of weak edge and nonedge pixels (used to automatically calculate threshold values)** — `90`
- **Standard deviation of Gaussian filter** — `1`

Simulate and Display Results

Run the model and display the results using Video Viewer blocks.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:246x300
T=0.000


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:246x300
T=0.000

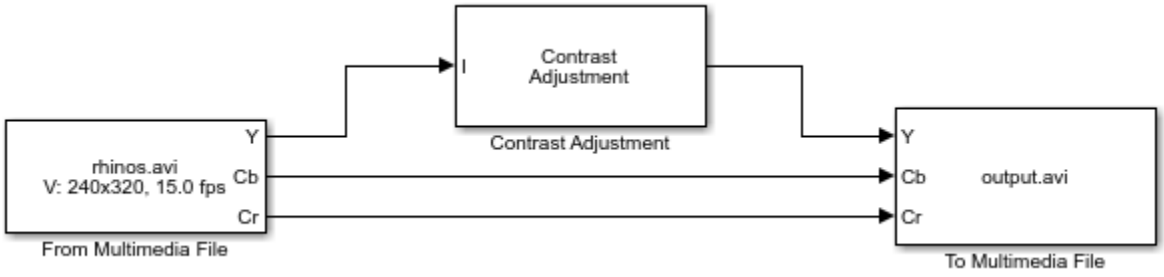
Read, Process, and Write Video Frames to File

This example shows you how to read video frames from a multimedia file, process them, and write them back to a new multimedia file.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkMultiMediaFile.slx';
open_system(modelname)
```



The model reads an input video file using a From Multimedia File block with these parameters:

- **File name** — `rhinos.avi`
- **Output color format** — `YCbCr 4:2:2`

To adjust the contrast of the input video, the model uses a Contrast Adjustment block with default settings. This block adjusts the contrast of the luminance (Y) component of each frame of the video.

The model writes the processed video frames to a file using a To Multimedia File block with the **File color format** parameter set to `YCbCr 4:2:2`. This block uses the default output file name of `output.avi`.

Simulate Model

Run the model and confirm that the model writes the output file to the working directory. Play the output video and verify the change in contrast from the input video to the output video as shown in this figure. The top image shows one frame of the input video file, with default contrast, and the bottom image shows the corresponding frame of the output video file, with adjusted contrast.

```
sim(modelname);
```

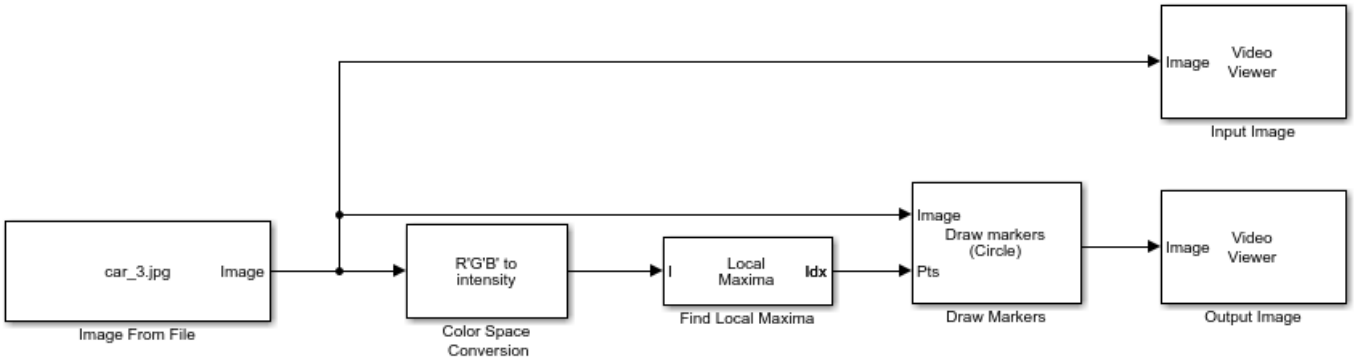
Find Local Maxima in Image

This example shows you how to find local maxima in an image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkLocalMaxima.slx';
open_system(modelname)
```

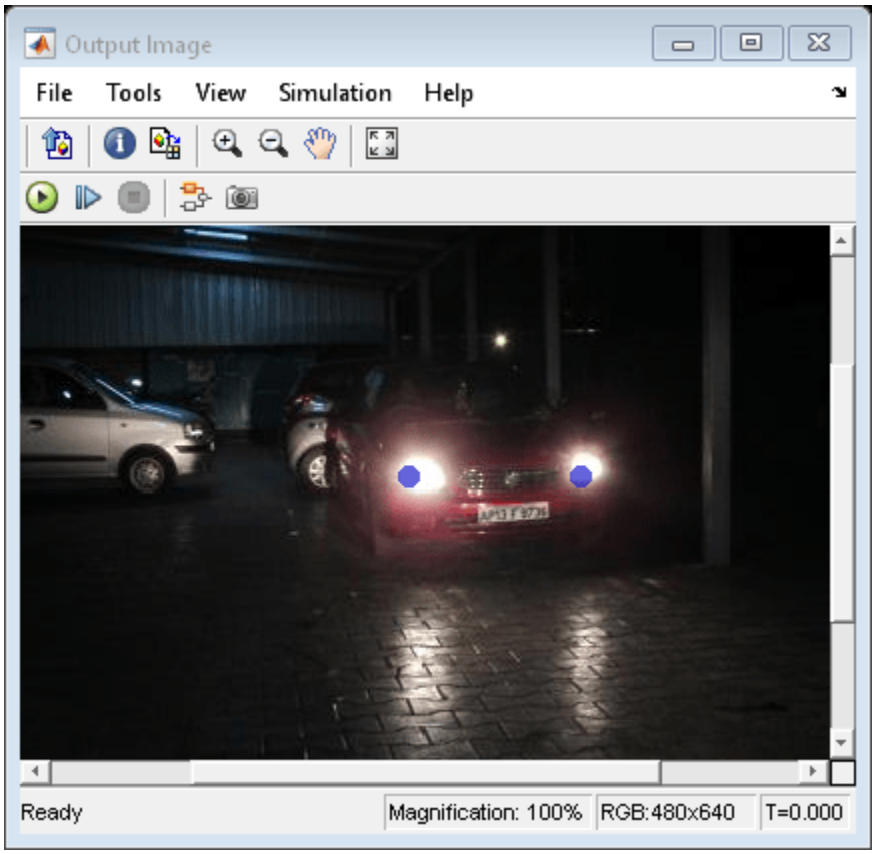


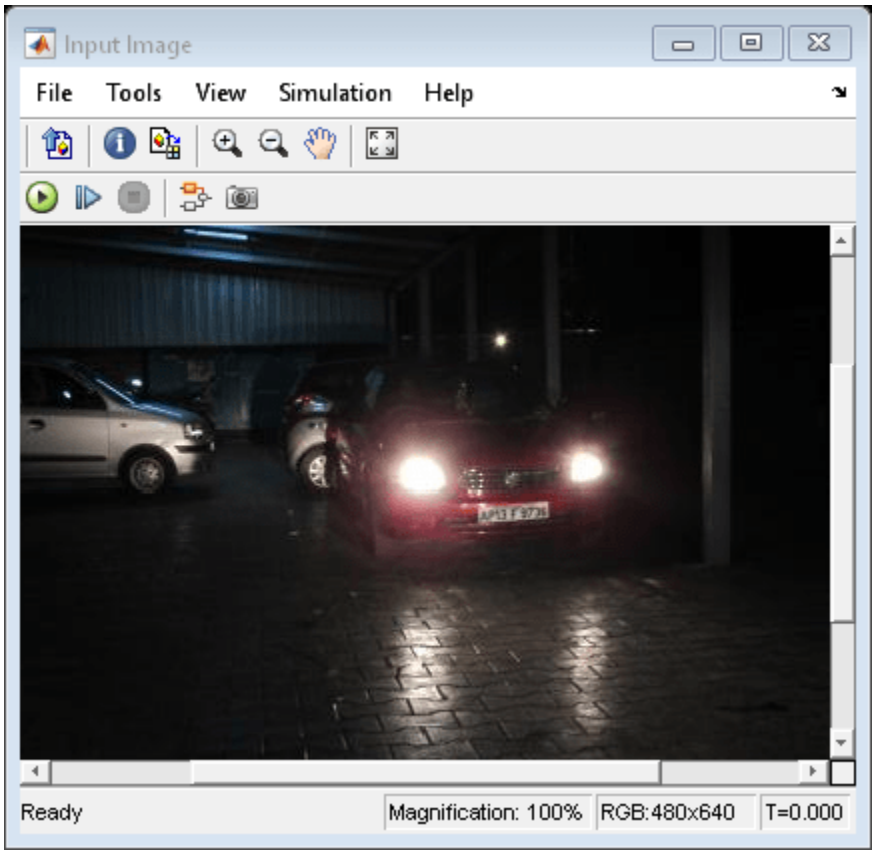
The model reads an RGB input image from a file using the Image From File block, and then converts it to an intensity image using the Color Space Conversion block. The Find Local Maxima block with **Neighborhood size** parameter of `[75 75]`, finds two local maxima in the intensity image. The Draw Markers block then draws the locations of the detected local maxima on the input image.

Simulate and Display Results

Run the model. The model displays the input image and the output image using Video Viewer blocks. Notice that the Find Local Maxima block indicates that the two brightest pixels correspond to the headlights of the front car in the input image.

```
sim(modelname);
```


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
RGB:480x640
T=0.000


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
RGB:480x640
T=0.000

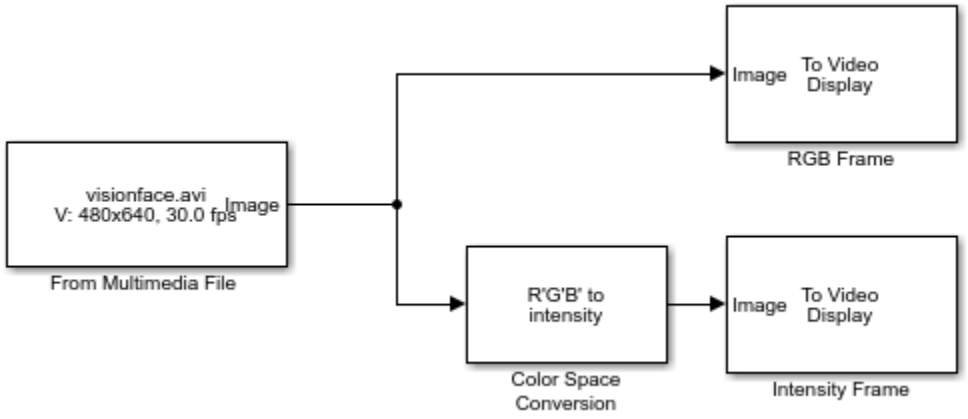
Read, Convert, and View Video from File

This example shows you how to read RGB video frames from a multimedia file, convert them to intensity frames, and display them using the To Video Display block.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkToVideoDisplay.slx';
open_system(modelname)
```



The model reads an input video file using a From Multimedia File block with these parameters:

- **File name** — `visionface.avi`
- **Output color format** — `RGB`

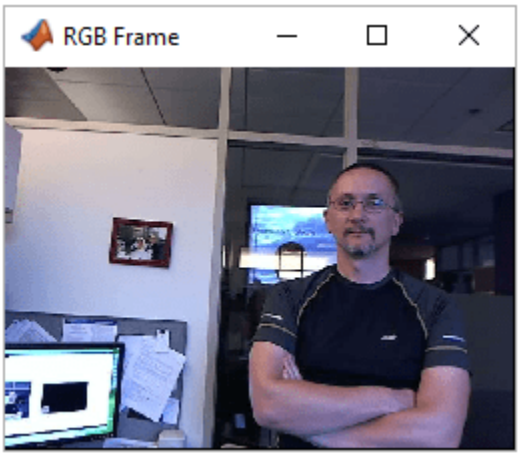
To convert the RGB video frames to intensity frames, the model uses the Color Space Conversion block with the **Conversion** parameter set to `R'G'B' to intensity`. This block converts each RGB frame of the video to an intensity frame.

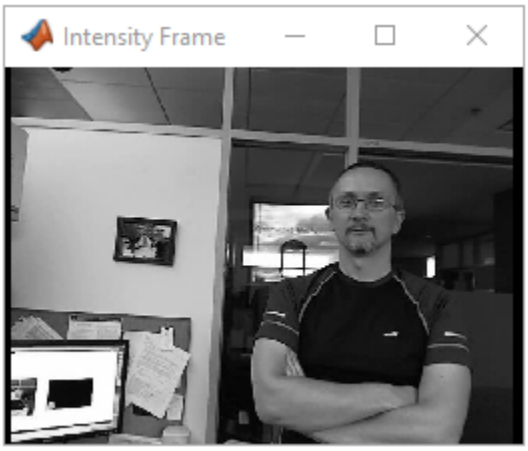
The model displays the original RGB video frames and the converted intensity video frames using To Video Display blocks with default parameters.

Simulate Model

Run the model. The model opens two display windows: one for the RGB frames and the other for the converted intensity frames. In the following figure, the top image shows the last RGB frame of the input video file, and the bottom image shows the corresponding converted intensity frame.

```
sim(modelname);
```


RGB Frame


Intensity Frame

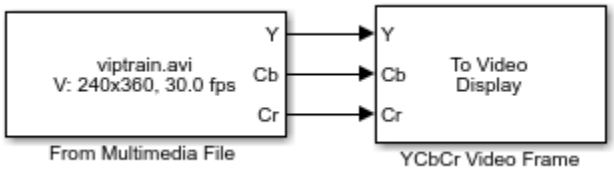
Read and Display YCbCr Video from File

This example shows you how to read video frames from a multimedia file as their luma and chrominance components, and then display them.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkToVideoDisplayYCbCr.slx';
open_system(modelname)
```



The model reads an input video file using a From Multimedia File block with these parameters:

- **File name** — `viptrain.avi`
- **Output color format** — `YCbCr 4:2:2`

The model displays the video frames using a To Video Display block with the **Input Color Format** parameter set to `YCbCr 4:2:2`. The **Y** port of each block represents the luma component and the **Cb** and **Cr** ports represent the blue-difference and red-difference chrominance components, respectively, of the input video frames.

Simulate the Model

Run the model. The model displays the video frames in a viewer window.

```
sim(modelname);
```


YCbCr Video Frame

Display Frame Rate of Input Video

This example shows you how to display the frame rate of input video data.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkFrameRateDisplay.slx';
open_system(modelname)
```

The model reads an input video file using a From Multimedia File block with these parameters:

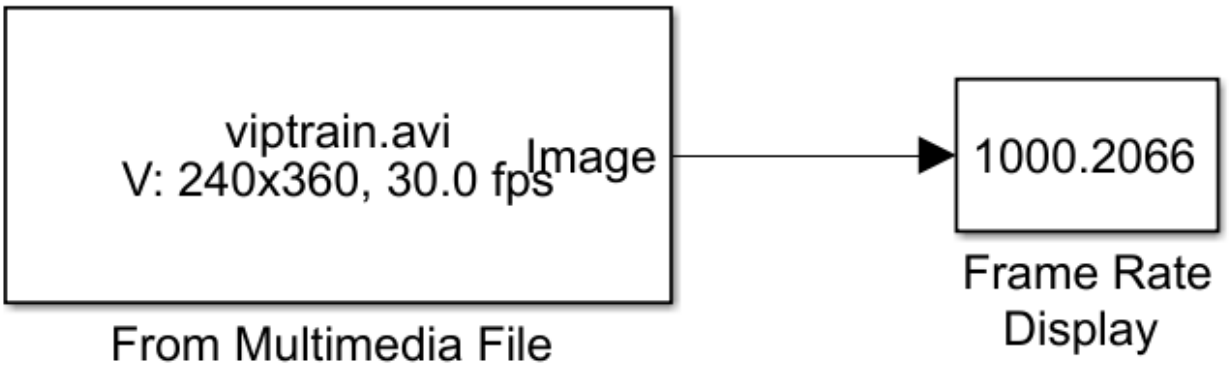
- **File name** — `viptrain.avi`
- **Output color format** — `RGB`

The model calculates the video frame rate using a Frame Rate Display block with the **Update rate** parameter set to `10`.

Simulate the Model

Run the model. The Frame Rate Display block displays the frame rate of the input video on the block mask. You can change the **Update rate** parameter in the Frame Rate Display block to control how often the block calculates the frame rate and updates the display on the block mask.

```
sim(modelname);
```



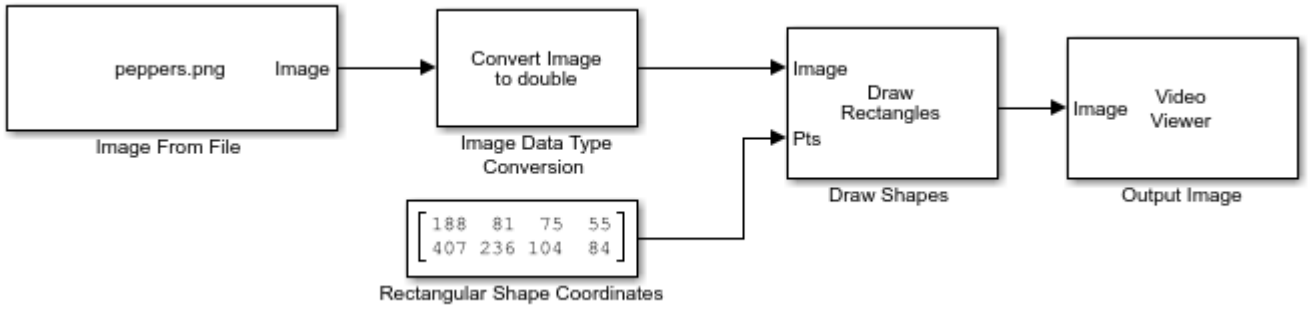
Draw Rectangles on Image

This example shows you how to draw rectangles on an image by overwriting the pixel values of the input image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkDrawShapesRectangles.slx';
open_system(modelname)
```



This model reads an input image using a Image From File block with the **File name** parameter set to `peppers.png`.

The model then changes the data type of the input image to `double`, using an Image Data Type Conversion block with the **Output data type** parameter set to `double`.

The model draws filled, translucent rectangles at the locations specified by the `Rectangular Shape Coordinates` Constant (Simulink) block by using a Draw Shapes block with these parameters:

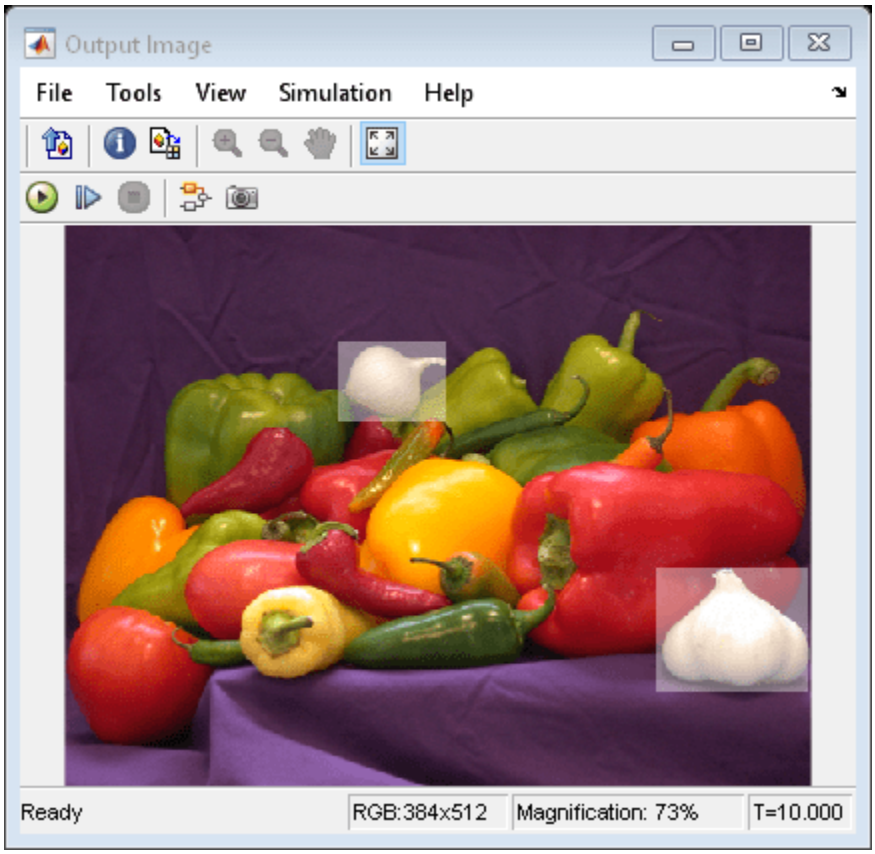
- **Shape** — `Rectangles`
- **Fill shapes** — `on`
- **Fill color** — `White`
- **Opacity factor (between 0 and 1)** — `0.4`

The Draw Shapes block overwrites the value of each pixel in the specified rectangular areas of the original image.

Simulate Model

Run the model. The model displays the shape-embedded output image by using a Video Viewer block..

```
sim(modelname);
```


Output Image
File
Tools
View
Simulation
Help
Ready
RGB:384x512
Magnification: 73%
T=10.000

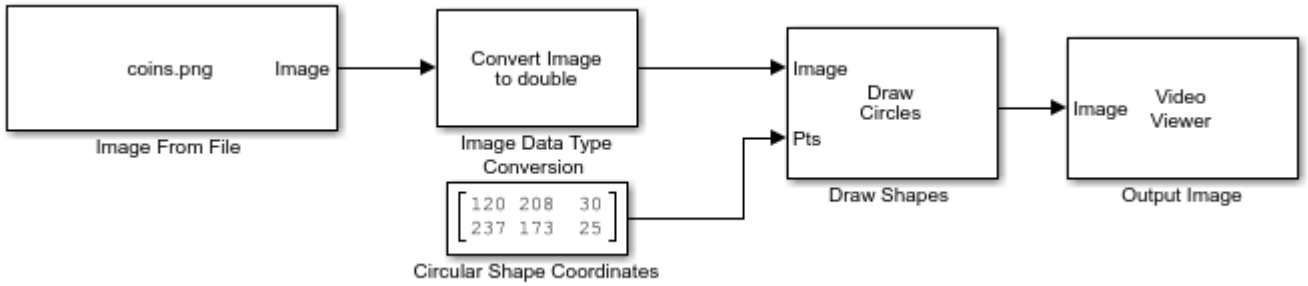
Draw Circles on Image

This example shows you how to draw circles on an image by overwriting the pixel values of the input image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkDrawShapesCircles.slx';
open_system(modelname)
```



This model reads the input image using a Image From File block with the **File name** parameter set to `coins.png`.

The model then changes the data type of the input image to `double`, using an Image Data Type Conversion block with the **Output data type** parameter set to `double`.

The model draws filled, translucent circles at the locations specified by the `Circular Shape Coordinates` Constant (Simulink) block by using a Draw Shapes block with these parameters:

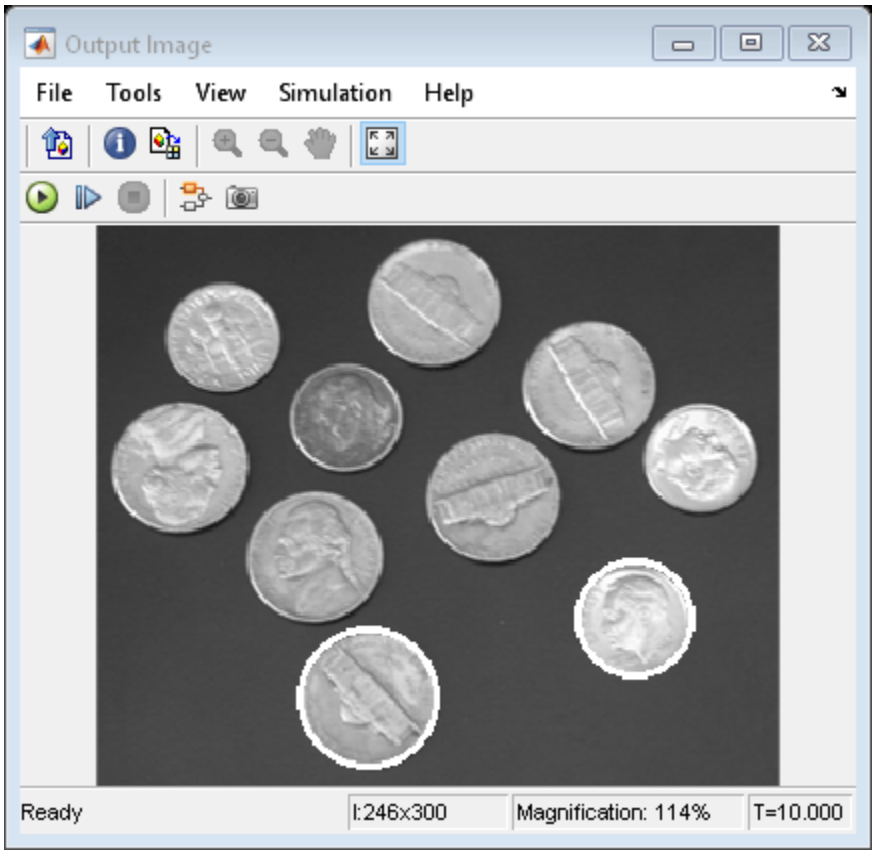
- **Shape** — `Circles`
- **Line width** — `3`
- **Border color** — `White`

The Draw Shapes block overwrites the value of each pixel along the circumferences of the specified areas of the original image.

Simulate the Model

Run the model. The model displays the output image, with white circles over two coins, by using a Video Viewer block.

```
sim(modelname);
```


Output Image
File
Tools
View
Simulation
Help
Ready
I:246x300
Magnification: 114%
T=10.000

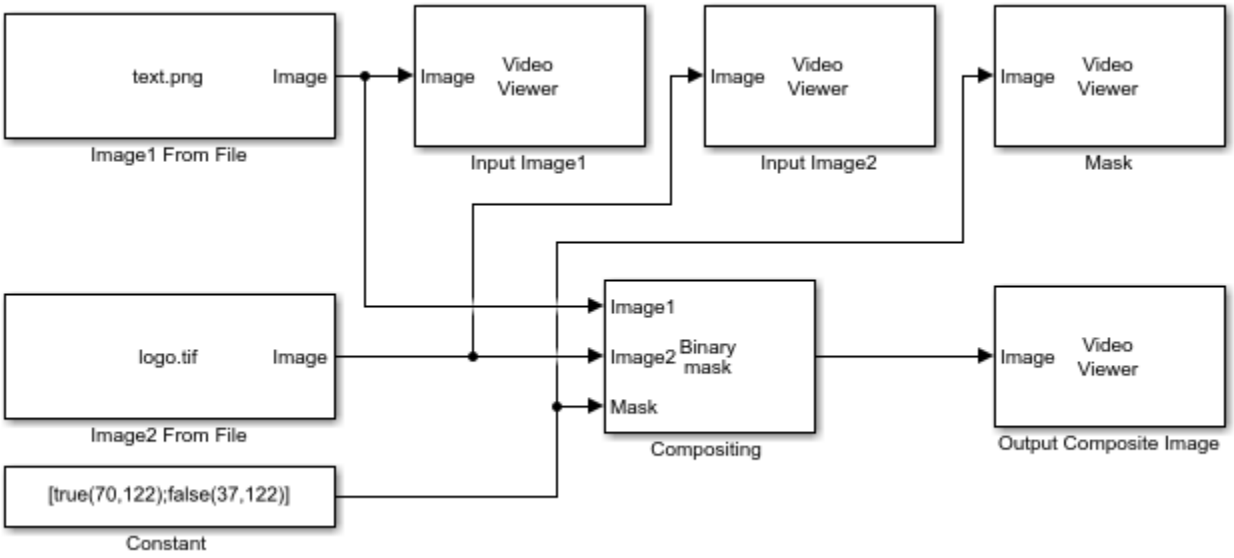
Overlay Images Using Binary Mask

This example shows you how to overlay one image on another image using a binary mask.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkCompositingMask.slx';
open_system(modelname)
```



This model reads two input binary images using Image From File blocks with their **File name** parameters set to `text.png` and `logo.tif`, respectively.

To overlay one image over another image and perform masking, the model uses these parameters for the Compositing block:

- **Operation** — `Binary mask`
- **Mask source** — `Input port`
- **Location source** — `Specify via dialog`
- **Location [x y]** — `[19 89]`

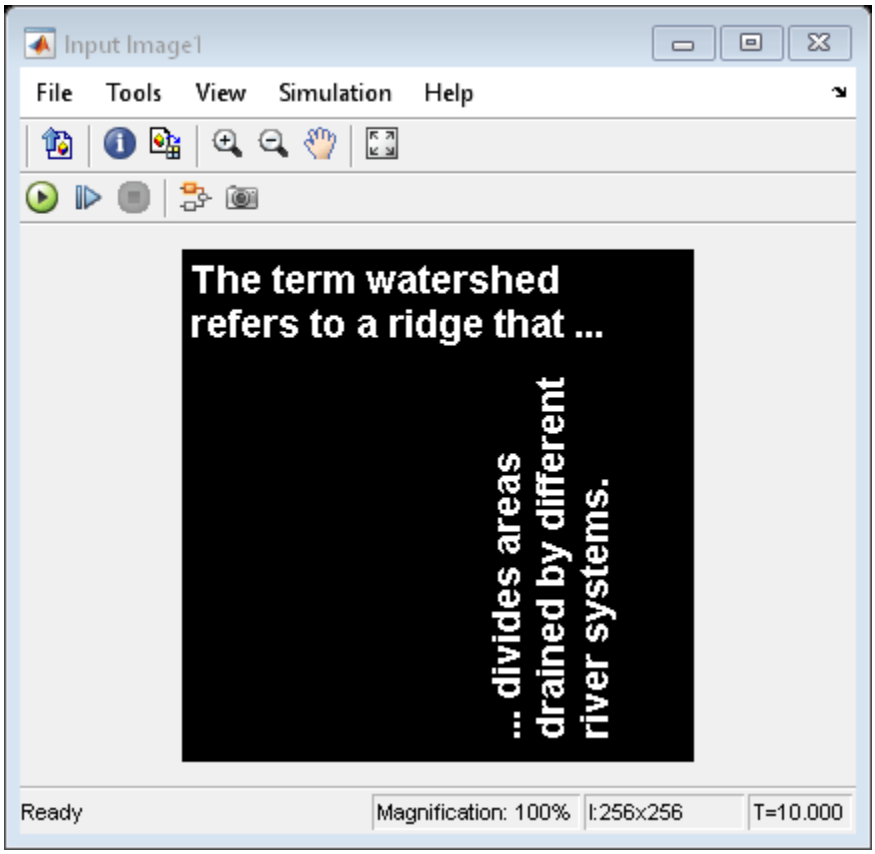
This model overlays the input **Image2** on **Image1**. **Location [x y]** parameter specify the row and column coordinates in **Image1** on which the upper left corner pixel of **Image2** has to be positioned.

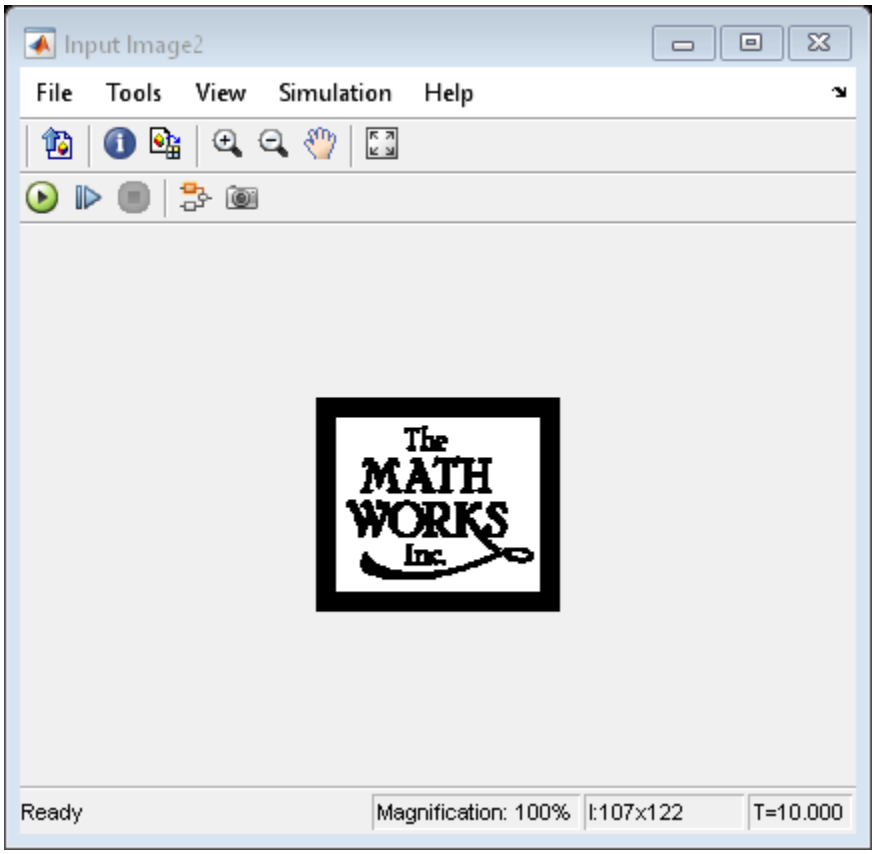
The Constant block specifies a 2-D binary mask, the same size as the image input to **Image2**, to the **Mask** port of the Compositing block.

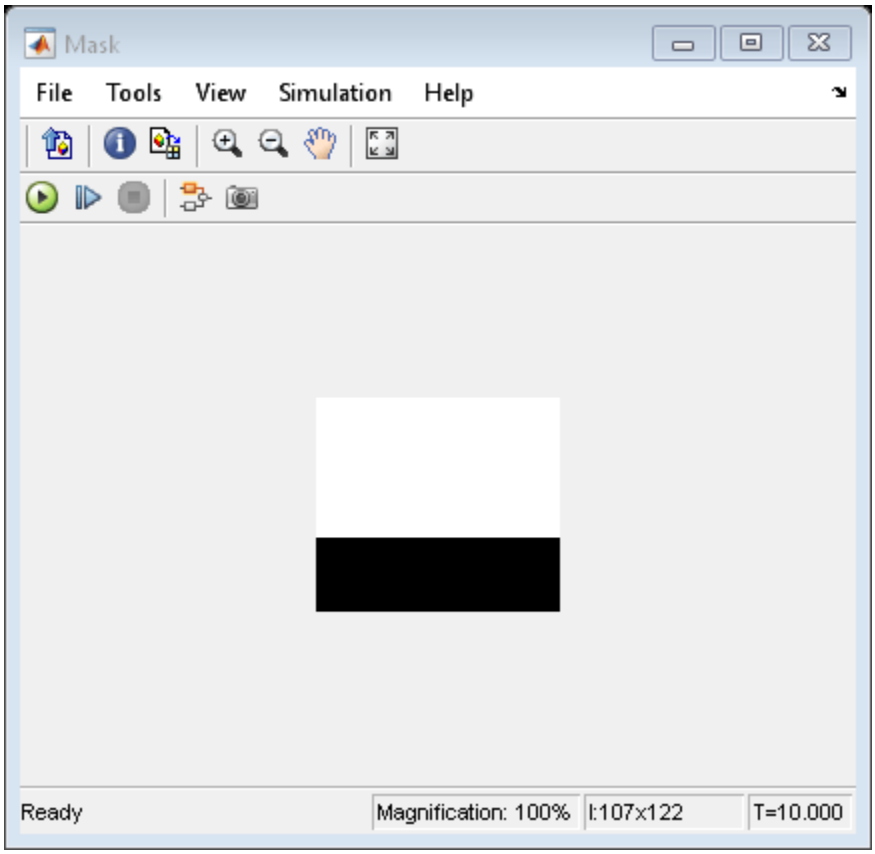
Simulate the Model

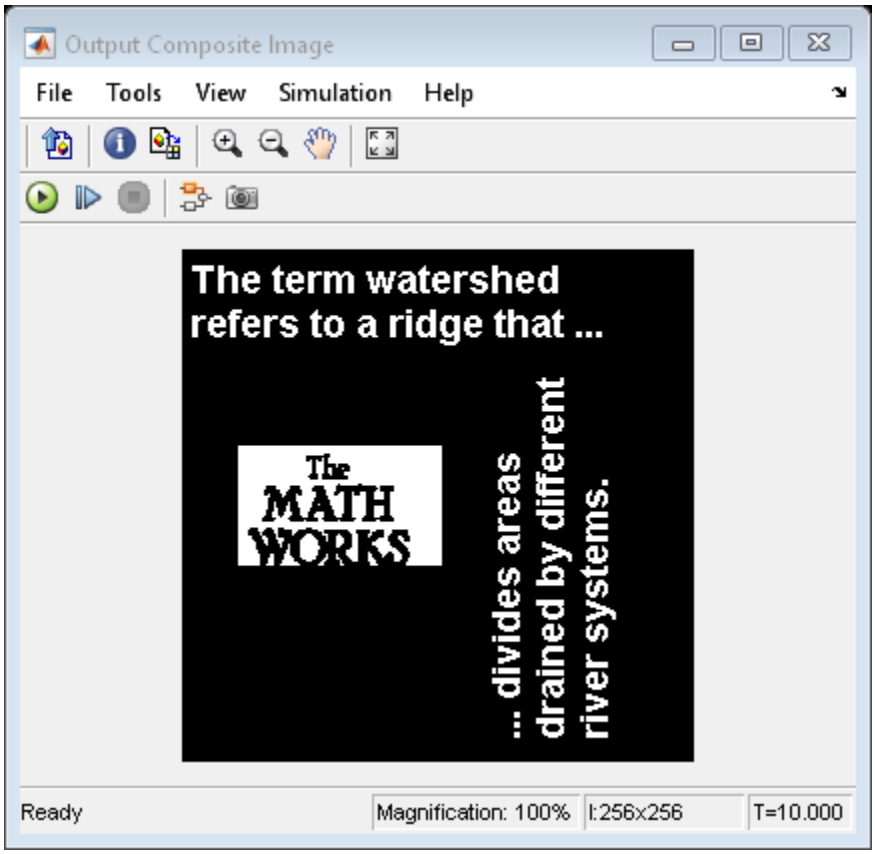
Run the model. The Video Viewer block display the input images, the mask, and the masked output image.

```
sim(modelname);
```


Input Image1
File
Tools
View
Simulation
Help
The term watershed refers to a ridge that ...
... divides areas drained by different river systems.
Ready
Magnification: 100%
I:256x256
T=10.000


Input Image2
File
Tools
View
Simulation
Help
The
MATH
WORKS
Inc.
Ready
Magnification: 100%
I:107x122
T=10.000


Mask
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:107x122
T=10.000


Output Composite Image
File Tools View Simulation Help
The term watershed refers to a ridge that ...
The MATH WORKS
... divides areas drained by different river systems.
Ready
Magnification: 100%
I:256x256
T=10.000

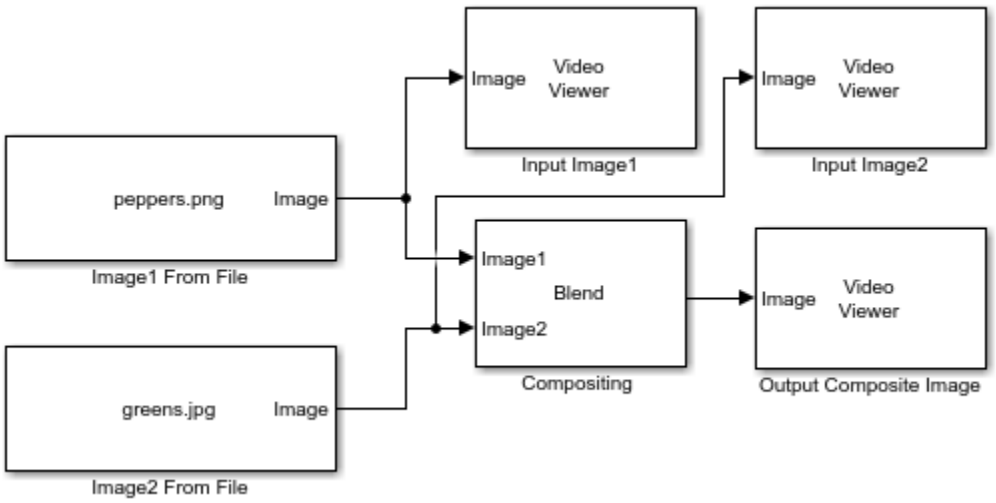
Linearly Combine Two Images

This example shows you how to linearly combine two images using the blend operation.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkCompositingBlend.slx';
open_system(modelname)
```



This model reads two input images using Image From File blocks with their **File name** parameters set to `peppers.png` and `greens.jpg`, respectively.

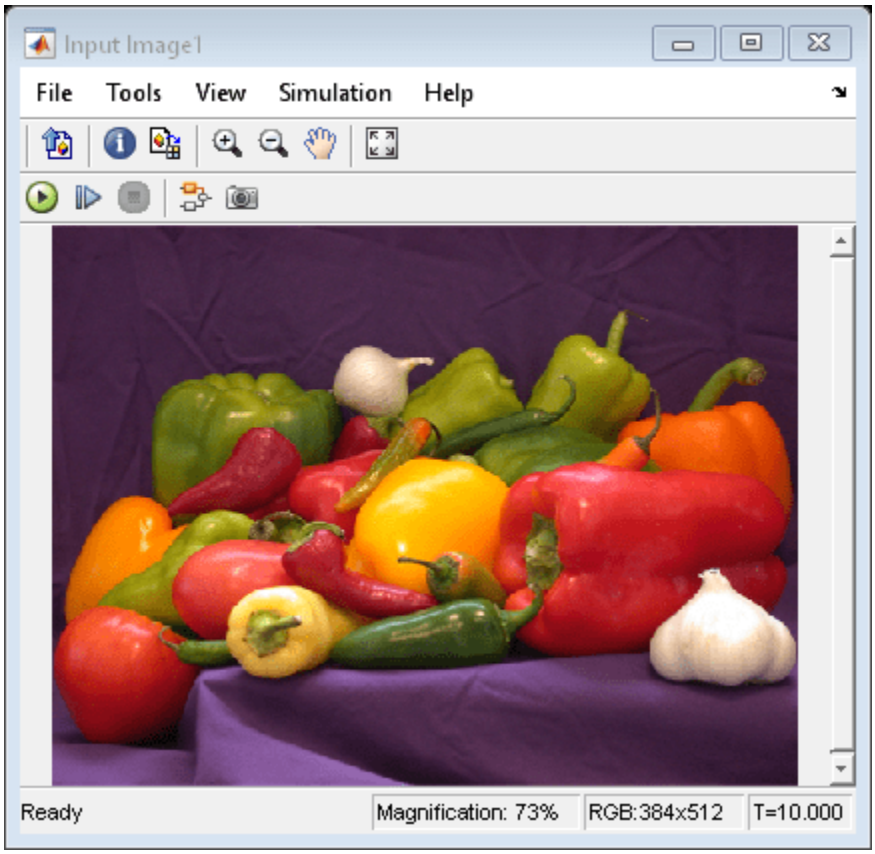
To linearly combine two images using blend operation, the model uses these following parameters for the Compositing block:

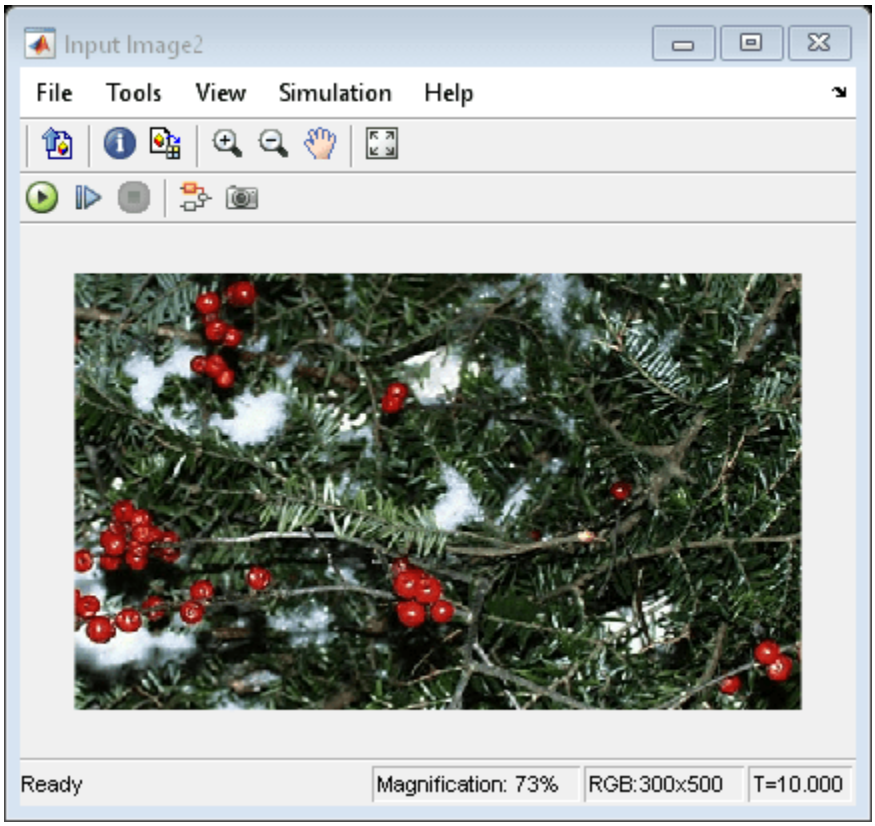
- **Operation** — `Blend`
- **Opacity factor(s) source** — `Specify via dialog`
- **Opacity factor(s)** — `0.5`
- **Location source** — `Specify via dialog`
- **Location [x y]** — `[7 50]`

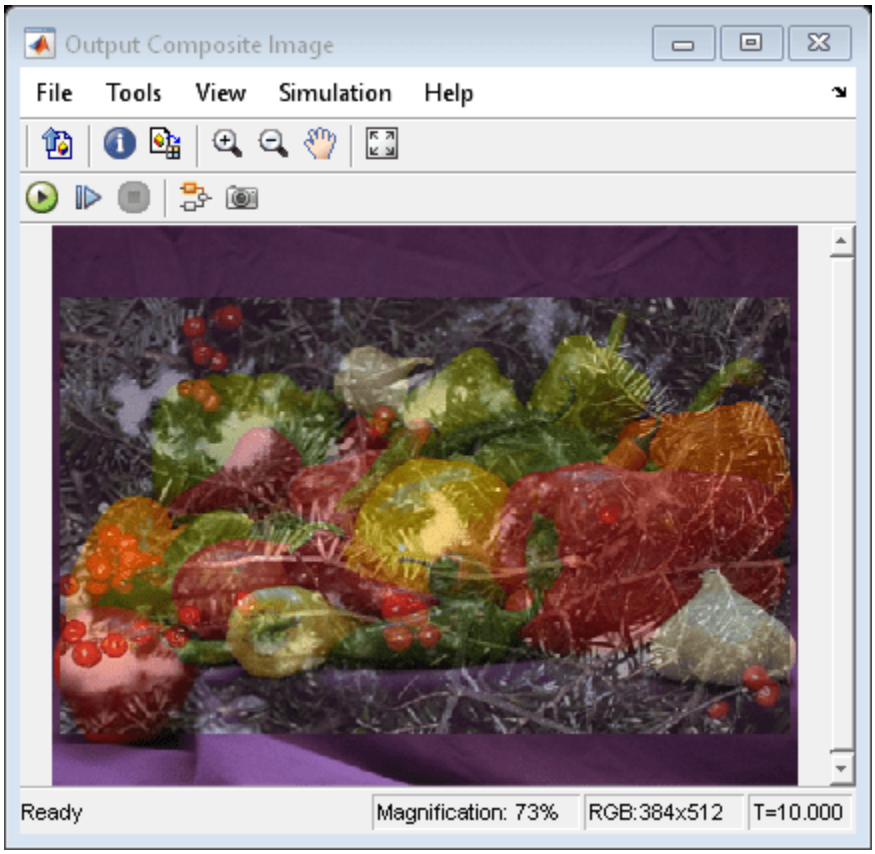
Simulate the Model

Run the model. The Video Viewer blocks display the input images and the blended output image. You can change the **Opacity factor(s)** parameter to any value in the range [0, 1], and see how this affects the blended output image.

```
sim(modelname);
```


Input Image1
File
Tools
View
Simulation
Help
Ready
Magnification: 73%
RGB:384x512
T=10.000


Input Image2
File
Tools
View
Simulation
Help
Ready
Magnification: 73%
RGB:300x500
T=10.000


Output Composite Image
File
Tools
View
Simulation
Help
Ready
Magnification: 73%
RGB:384x512
T=10.000

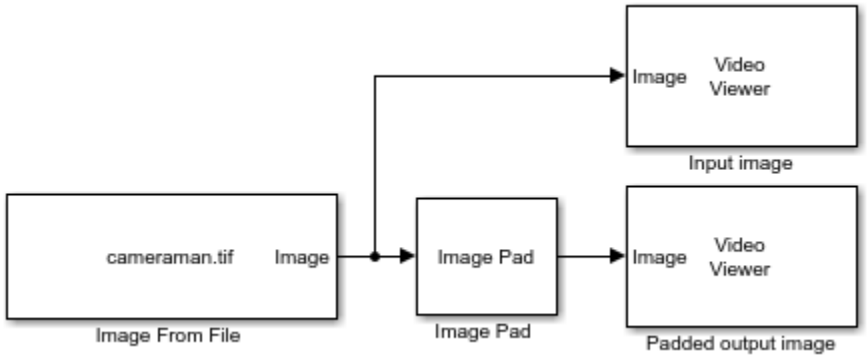
Pad Zeros to Image

This example shows you how to add rows and columns of zeros to an image by using a padding operation.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkImagePad.slx';
open_system(modelname)
```



This model reads an input image using an Image From File block with the **File name** parameter set to `cameraman.tif`.

The model pads zeros to all four sides of the input image by using an Image Pad block with these parameters:

- **Method** — `Constant`
- **Pad value source** — `Specify via dialog`
- **Pad value** — `0`
- **Specify** — `Pad size`
- **Add columns to** — `Both left and right`
- **Number of added columns** — `[10,10]`
- **Add rows to** — `Both top and bottom`
- **Number of added rows** — `[10,10]`

Simulate the Model

Run the model. The model displays the input image and the zero-padded output image by using a Video Viewer block. The padded zeros create the black border of the output image.

```
sim(modelname);
```


Input image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=10.000


Padded output image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:276x276
T=10.000

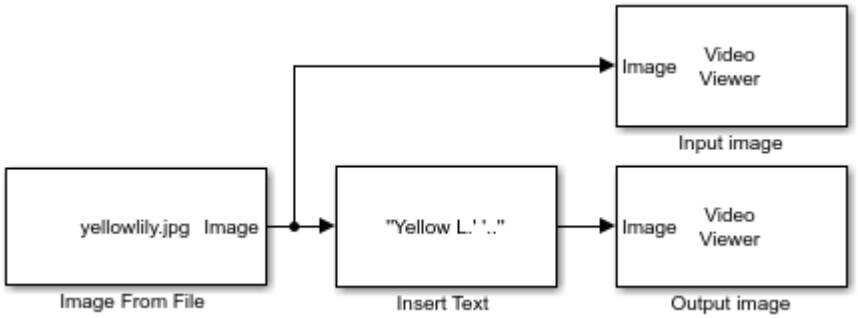
Insert Text into Image

This example shows you how to insert colored text into an input image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkInsertText.slx';
open_system(modelname)
```



This model reads an input image using an Image From File block with the **File name** parameter set to `yellowlily.jpg`.

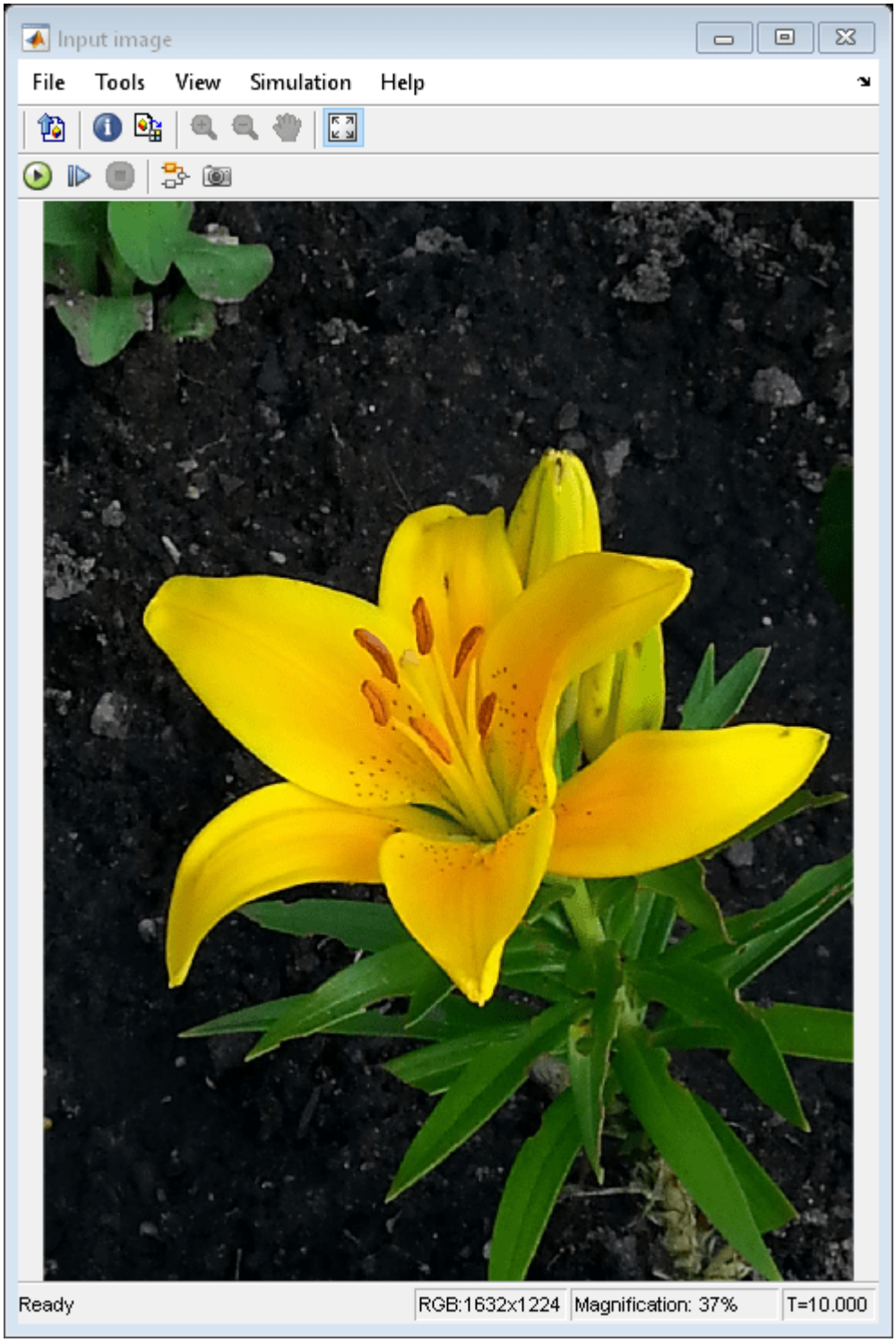
The model inserts text into the input image by using an Insert Text block with these parameters:

- **Text** — `'Yellow Lilly'`
- **Color value source** — `Specify via dialog`
- **Color value** — `[255,255,0]`
- **Location source** — `Specify via dialog`
- **Location [x y]** — `[600,50]`
- **Opacity** — `1`
- **Font face** — `LucidaSansDemiBold`
- **Font size (points)** — `100`

Simulate the Model

Run the model. The model displays both the original input image and the output image with the inserted bold, yellow text by using Video Viewer blocks.

```
sim(modelname);
```


Input image
File
Tools
View
Simulation
Help
Ready
RGB:1632x1224
Magnification: 37%
T=10.000


Output image
File
Tools
View
Simulation
Help
Yellow Lilly
Ready
RGB:1632x1224
Magnification: 37%
T=10.000

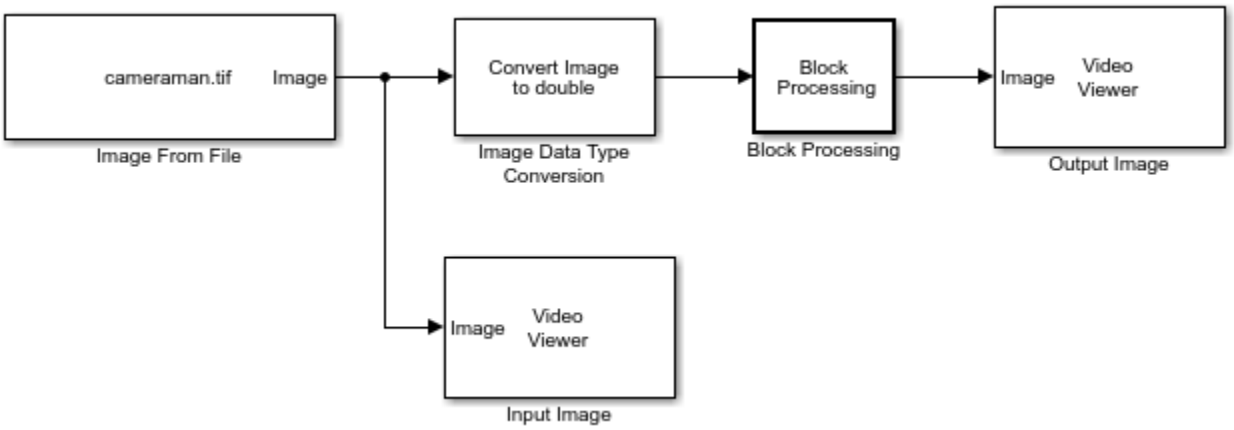
Compress Image Using 2-D DCT

This example shows how to compress an image using a 2-D discrete cosine transform (DCT). The example computes the 2-D DCT of 8-by-8 nonoverlapping blocks of the input image, discards (sets to zero) all but 10 of the 64 DCT coefficients in each block, and then reconstructs the image using the 2-D inverse discrete cosine transform (IDCT) of each block.

Example Model

Open the Simulink® model.

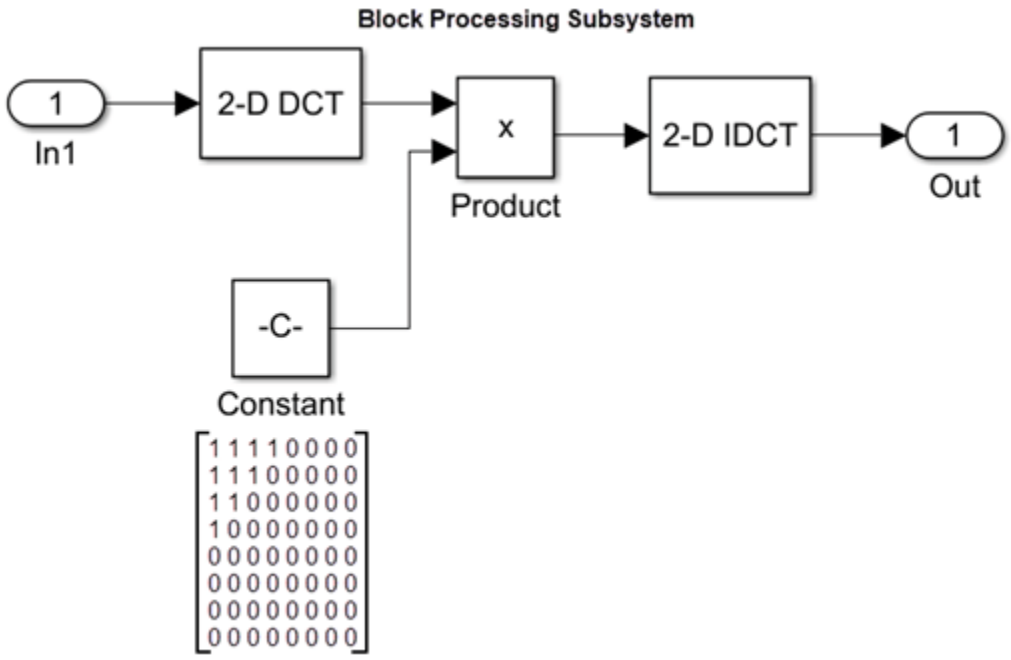
```
modelname = 'ex_blk2ddct.slx';
open_system(modelname)
```



The model reads an input image using the Image From File block, and converts the data type of the image to `double` using the Image Data Type Conversion block.

Process Image Block-by-Block

The Block Processing block performs the operations defined in this subsystem on each block of the input image.

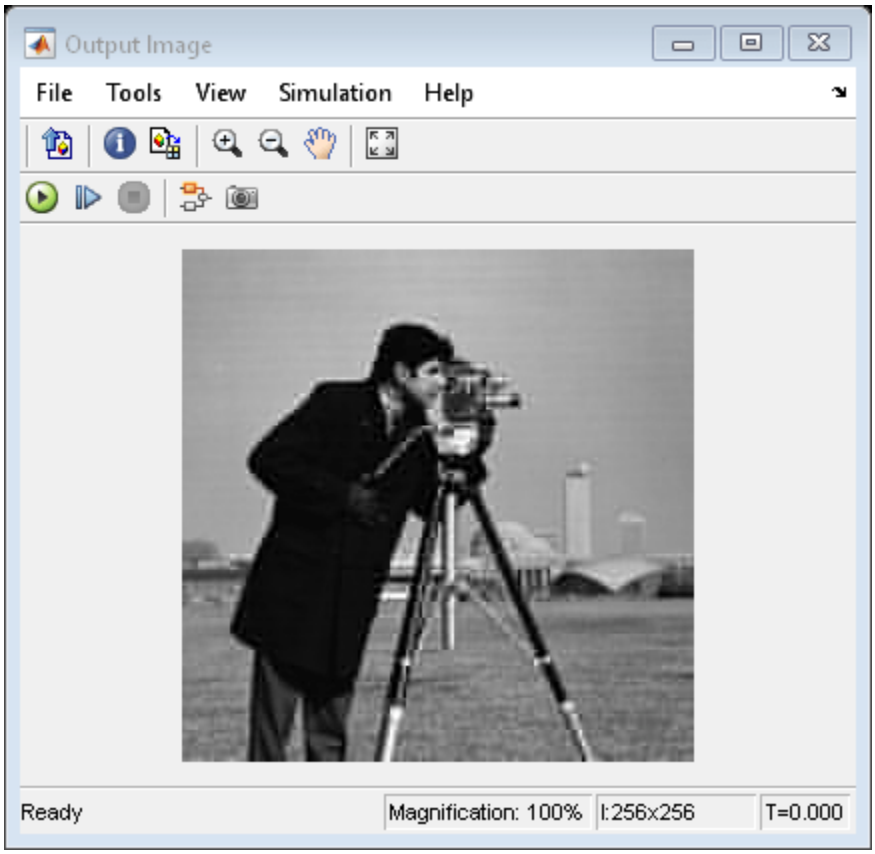


The subsystem first computes the 2-D DCT of a block of the input image. The Product, Matrix Multiply (Simulink) block then multiplies the DCT coefficients by the elements of the mask, defined using the Constant (Simulink) block, to discard all but 10 of the 64 DCT coefficients. The 2-D IDCT block then reconstructs the processed image.

Simulate and Display Results

Simulate the model. The Video Viewer blocks display the input and output images. Notice that, despite some loss of quality, the output image is clearly recognizable even after discarding nearly 85% of the DCT coefficients of the input image.

```
sim(modelname);
```


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=0.000


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:256x256
T=0.000

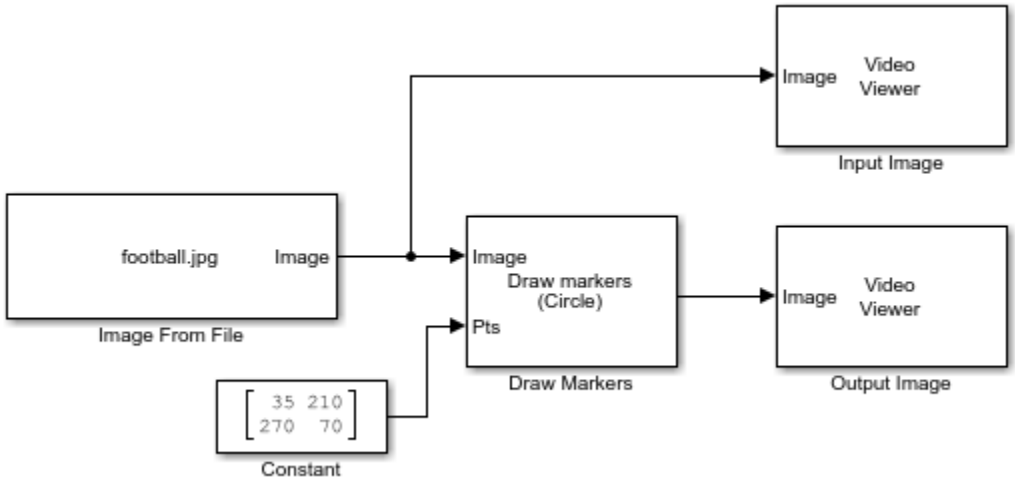
Draw Markers on Image

This example shows you how to draw markers of different colors on an image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkDrawMarkers.slx';
open_system(modelname)
```



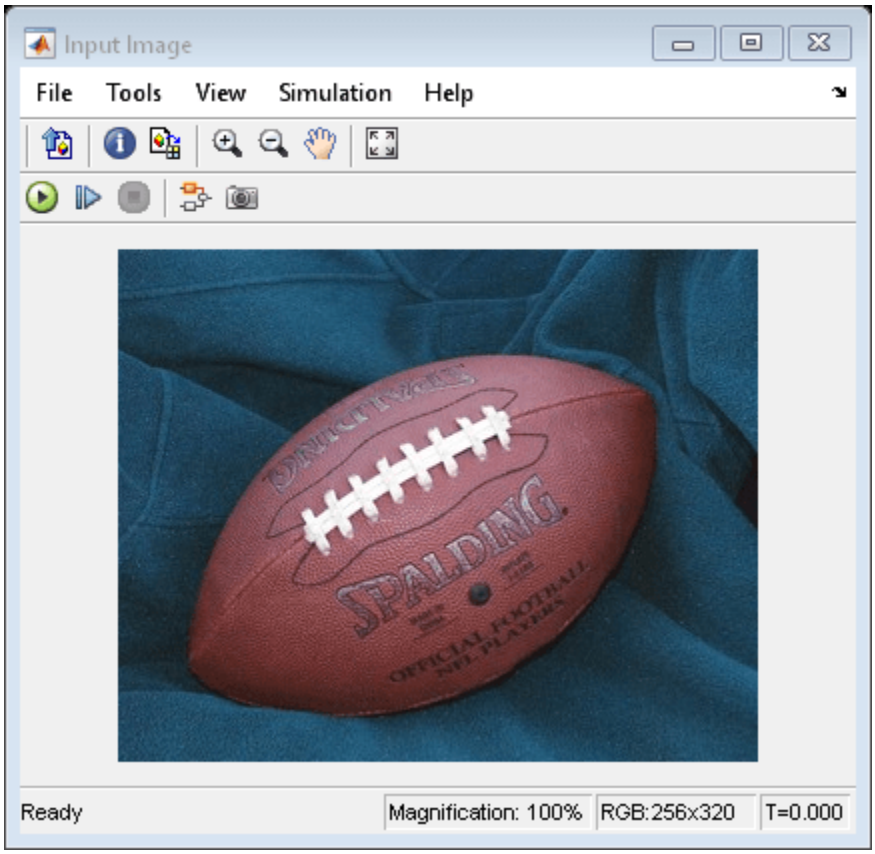
The model reads an input image using the Image From File block. The Constant (Simulink) block, which has a **Constant value** parameter of `[35 210;270 70]`, specifies the pixel coordinates at which to draw two markers. The Draw Markers block then draws the markers on the input image. These markers are filled circles colored green and yellow, respectively. The Draw Markers block uses these nondefault parameters:

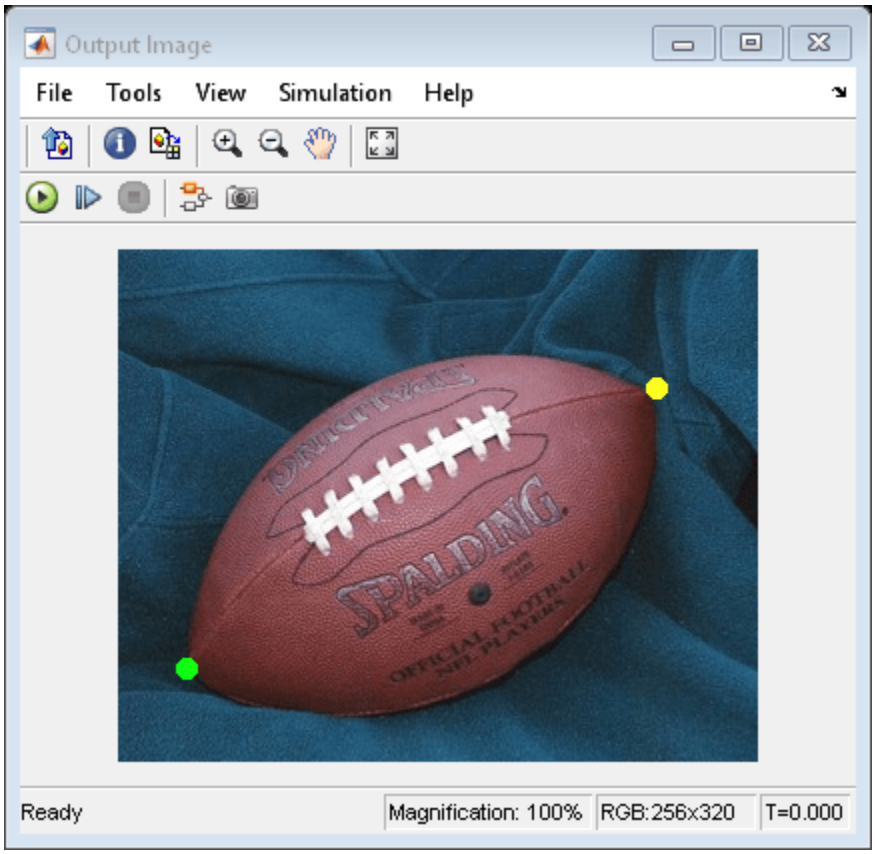
- **Marker size** — `5`
- **Filled** — `on`
- **Fill color source** — `Specify via dialog`
- **Fill color** — `User-specified value`
- **Color value(s)** — `[0 255 0;255 255 0]`

Simulate and Display Results

Run the model and visualize the two markers in the output image. The model uses Video Viewer blocks to display the input and output images.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
SPALDING
Ready
Magnification: 100%
RGB:256x320
T=0.000


Output Image
File
Tools
View
Simulation
Help
SPALDING
Ready
Magnification: 100%
RGB:256x320
T=0.000

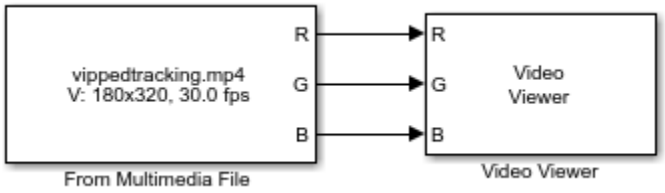
Read and Display RGB Video from File

This example shows you how to read video frames from a multimedia file as their separate red, green, and blue components, and then display the video frames.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkVideoViewer.slx';
open_system(modelname)
```



The model reads an input video file using a From Multimedia File block with these parameters:

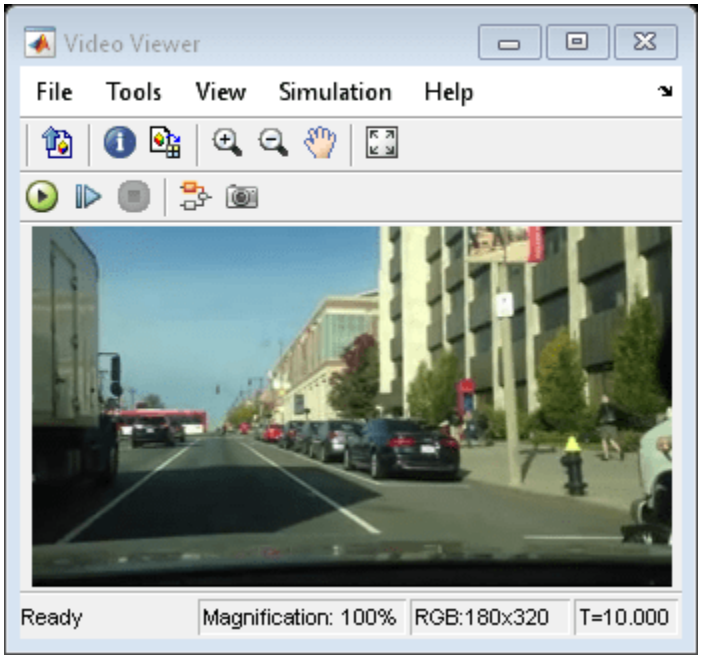
- **File name** — `vippedtracking.mp4`
- **Output color format** — `RGB`
- **Image signal** — `Separate color signals`

The model displays the video frames using a Video Viewer block with the **Separate Color Signals** parameter enabled. The **R**, **G**, and **B** ports of each block represent the red, green and blue color channels, respectively, of the input video frames.

Simulate the Model

Run the model. The model displays the video frames in a viewer window.

```
sim(modelname);
```


Video Viewer
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
RGB:180x320
T=10.000

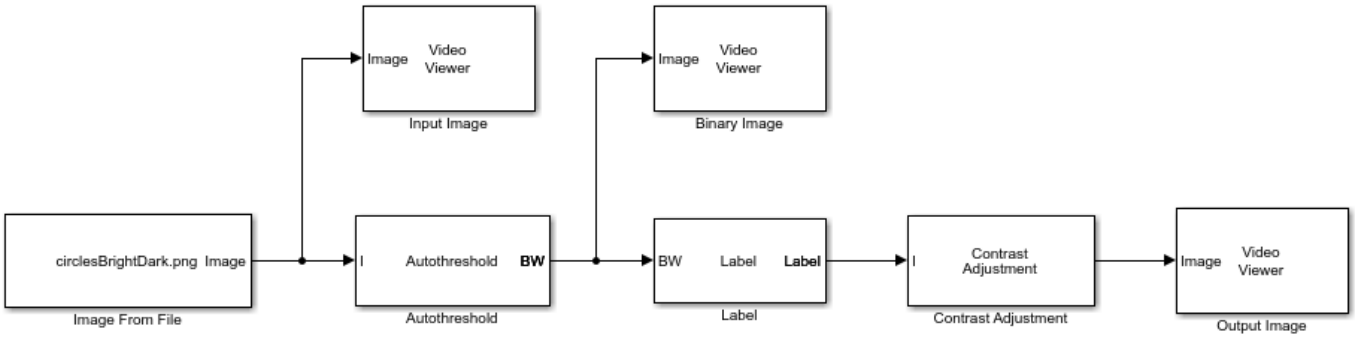
Label Objects in Binary Image

This example shows you how to label objects in a binary image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkLabel.slx';
open_system(modelname)
```



The model reads an input intensity image using the Image From File block. The Autothreshold block converts the intensity image into a binary image using a **Threshold scaling factor** parameter value of 3. To label the objects in the binary image, the model uses the Label block with these nondefault parameters:

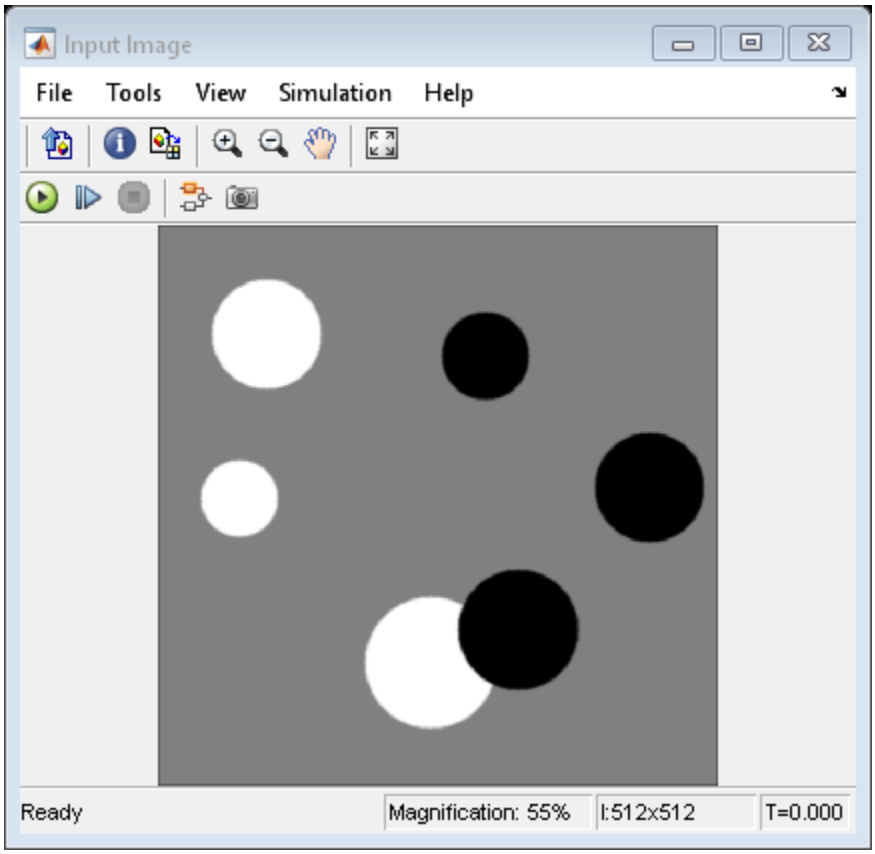
- **Output** — `Label matrix`
- **Output data type** — `uint8`

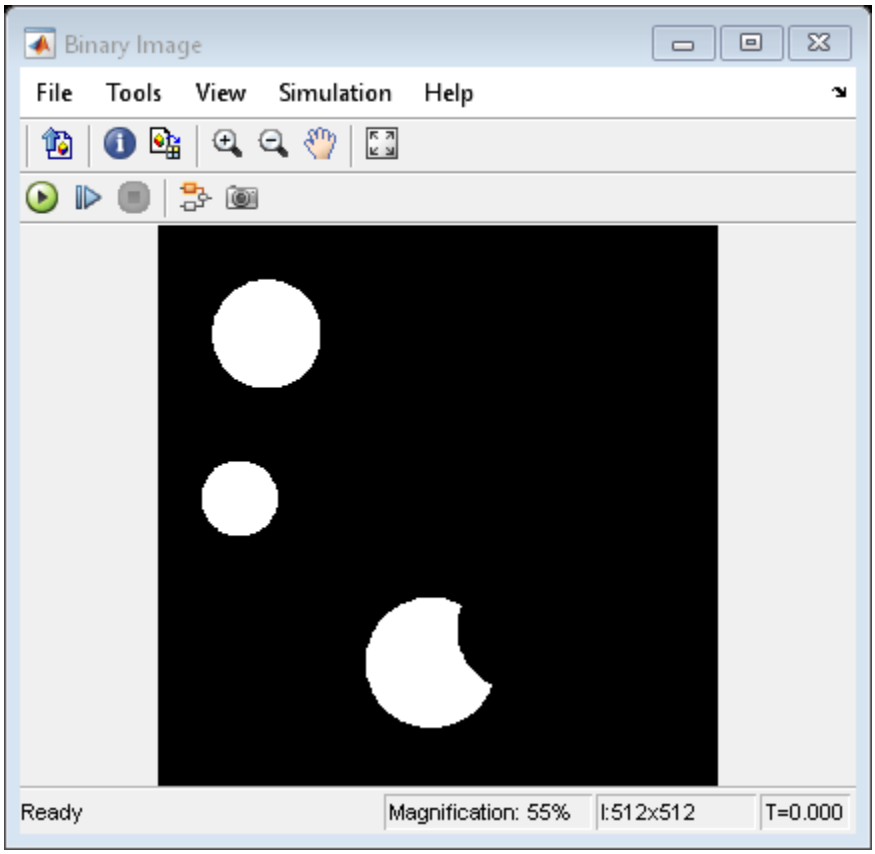
To visualize the labeled objects, the Contrast Adjustment block adjusts the range of intensity values of the output label matrix of the Label block.

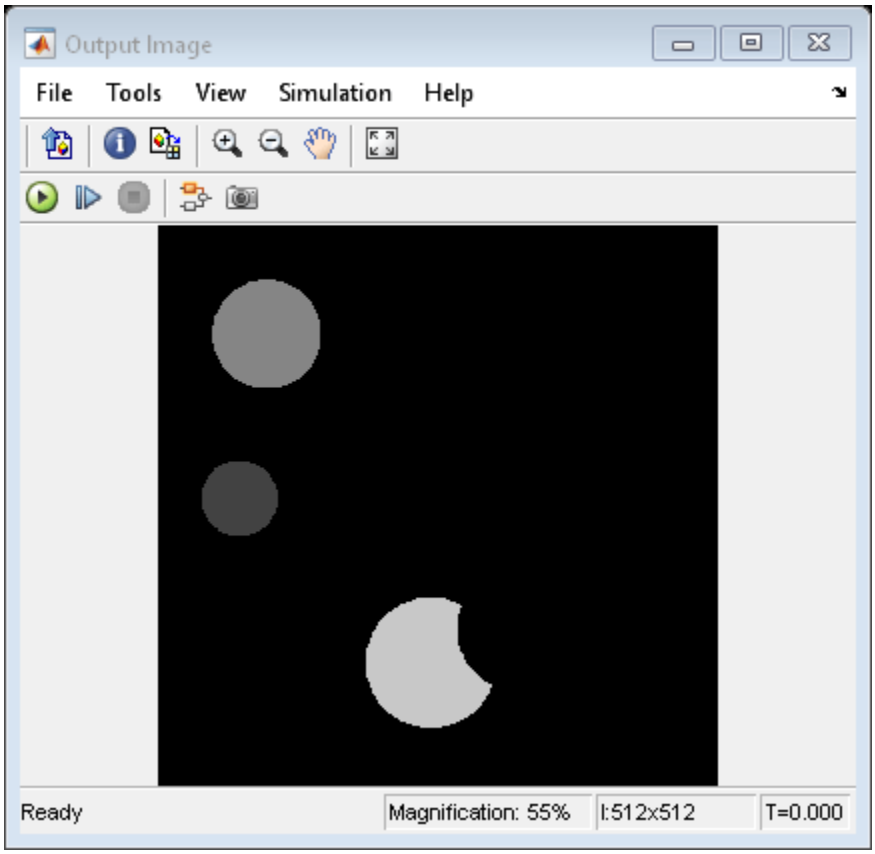
Simulate and Display Results

Run the model. The model displays the input image, the binary image, and the output image using Video Viewer blocks. Observe how the output image shows three labeled objects using different shades of gray.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 55%
I:512x512
T=0.000


Binary Image
File
Tools
View
Simulation
Help
Ready
Magnification: 55%
I:512x512
T=0.000


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 55%
I:512x512
T=0.000

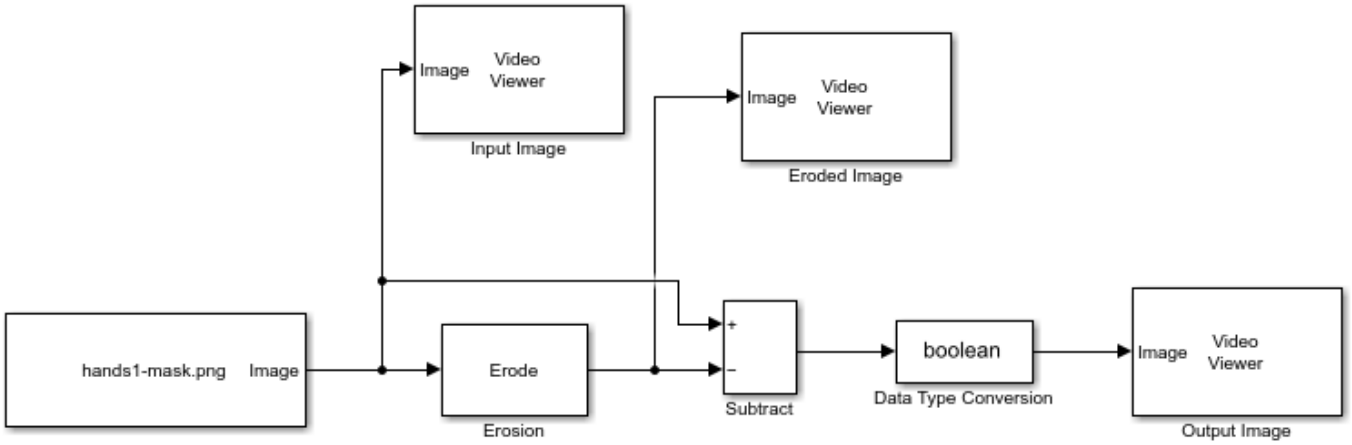
Boundary Extraction of Binary Image

This example shows how to extract a boundary from a binary image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkErosion.slx';
open_system(modelname);
```

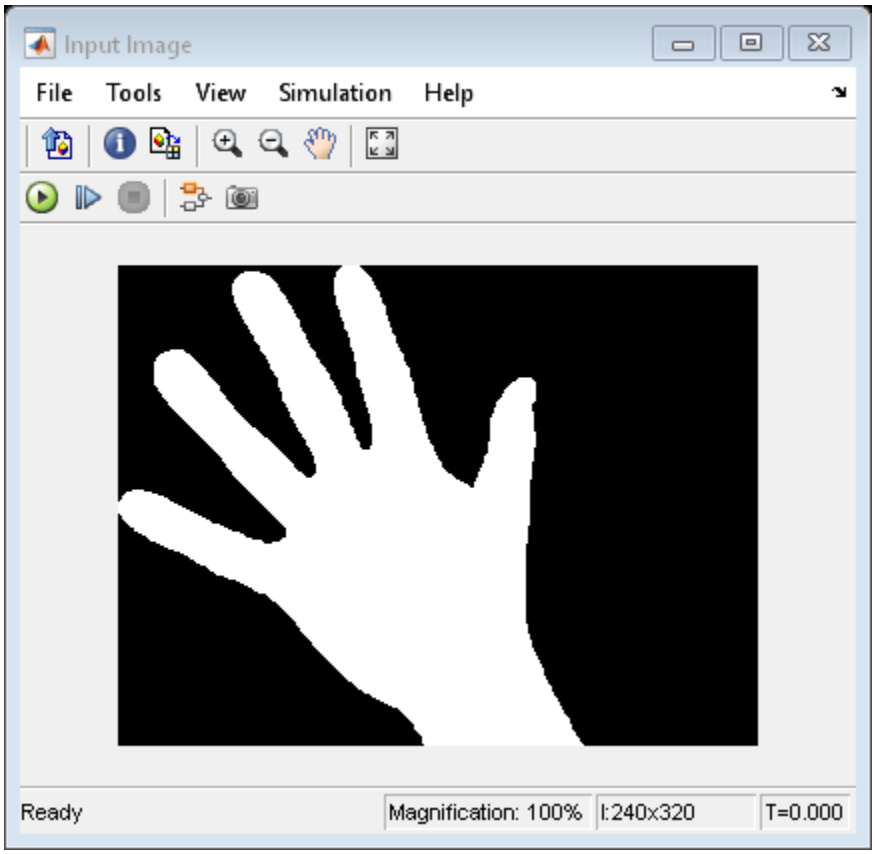


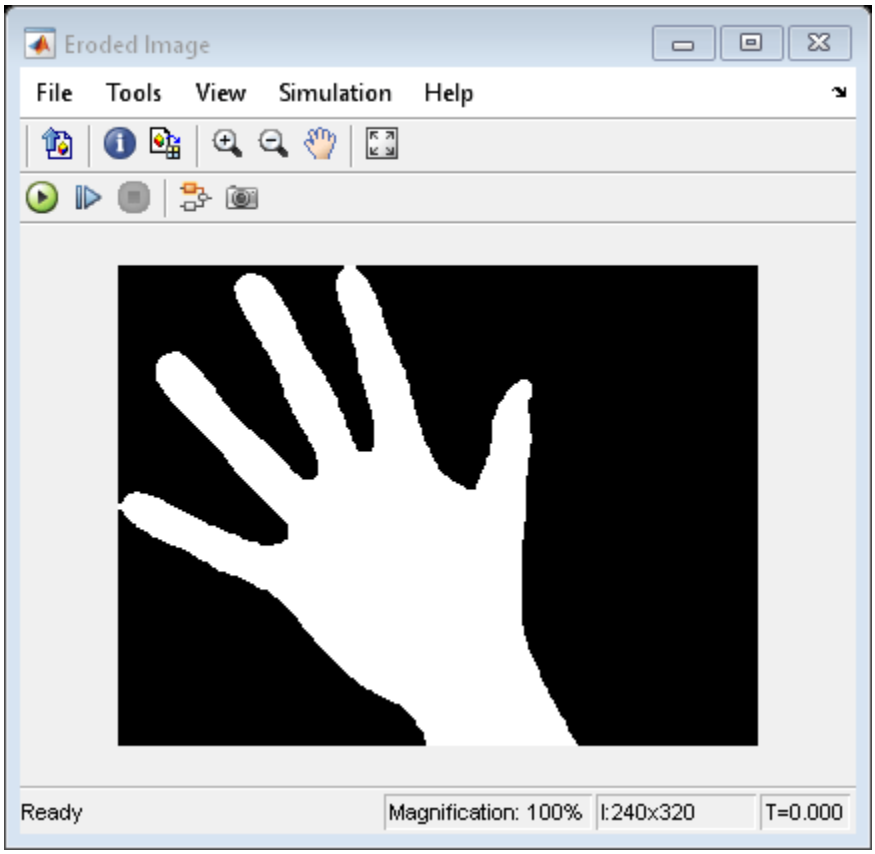
The model reads an input binary image using the Image From File block. The Erosion block erodes the input image using the default 4 pixel wide, square structuring element. To extract the boundary of the input image, the model uses the Subtract block to subtract the eroded image from the input image. The Data Type Conversion (Simulink) block converts the data type of the resulting image to `Boolean`, to match it with the data type of the input image.

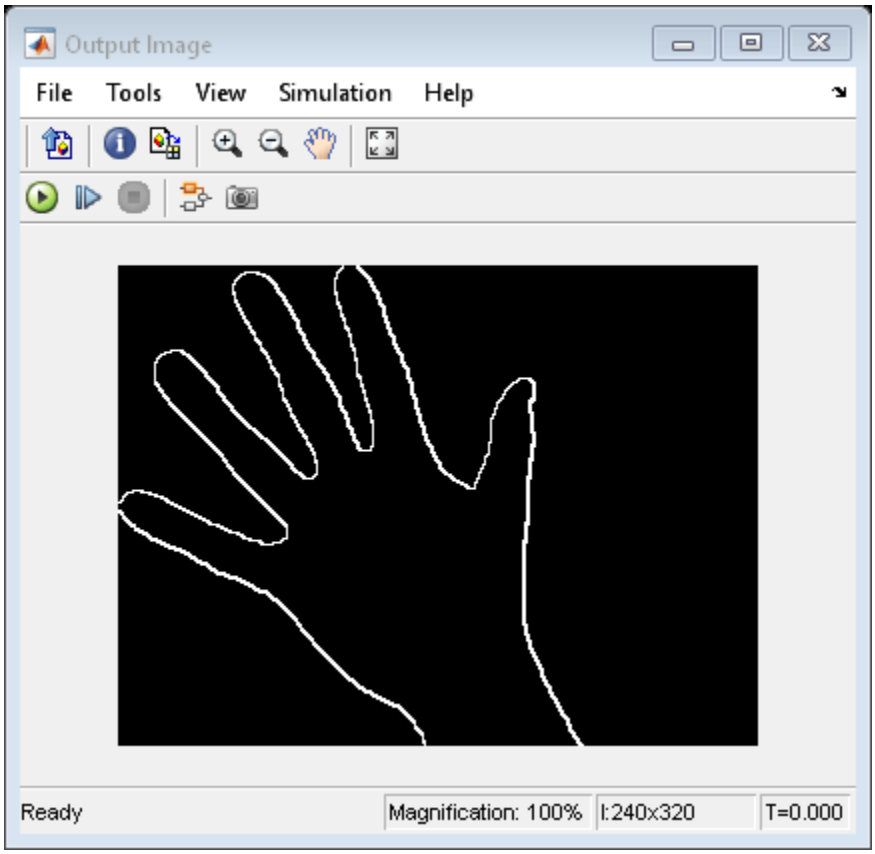
Simulate and Display Results

Run the model and visualize the extracted boundary of the input binary image. The model displays the input image, eroded image, and output image using Video Viewer blocks.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:240x320
T=0.000


Eroded Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:240x320
T=0.000


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:240x320
T=0.000

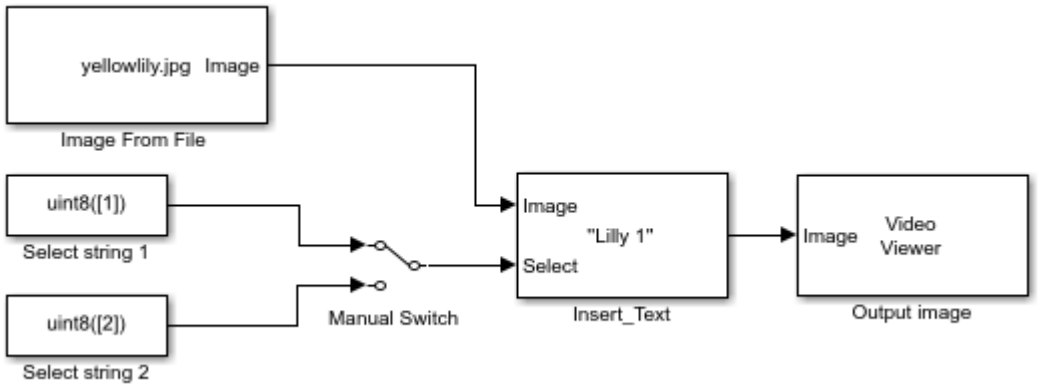
Select String to Insert into Image

This example shows you how to select a text string and insert it into an image when the Insert Text block contains multiple text strings.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkInsertTextSelect.slx';
open_system(modelname)
```



This model reads an input image using an Image From File block with the **File name** parameter set to `yellowlily.jpg`.

The block selects and inserts a text string into the input image by using an Insert Text block with these parameters:

- **Text** — `{'Lilly 1','Lilly 2'}`
- **Color value source** — `Specify via dialog`
- **Color value** — `[255,255,0]`
- **Location source** — `Specify via dialog`
- **Location [x y]** — `[600,50]`
- **Opacity** — `1`
- **Font face** — `LucidaSansDemiBold`
- **Font size (points)** — `100`

A Manual Switch (Simulink) block enables you to select which input string to display.

Simulate Model

Run the model. The model displays the output image with inserted bold, yellow text by using a Video Viewer block. Double-click the Manual Switch (Simulink) block to select the other string and display an output image with that string inserted.

```
sim(modelname);
```


Output image
File
Tools
View
Simulation
Help
Lilly 1
Ready
RGB:1632x1224
Magnification: 38%
T=10.000


Output image
File Tools View Simulation Help
Lilly 2
Ready
RGB:1632x1224
Magnification: 38%
T=10.000

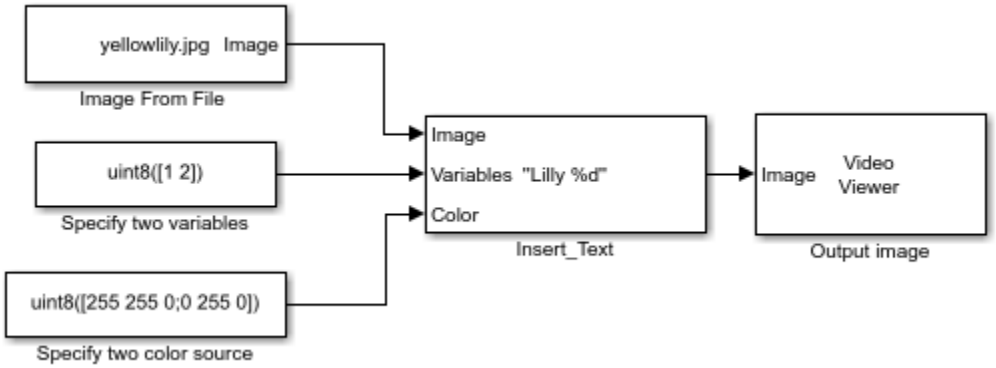
Insert Two Strings into Image at Different Locations

This example shows you how to insert two strings of different colors into an image at different location within the image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blkInsertTextLocation.slx';
open_system(modelname)
```



This model reads an input image using an Image From File block with the **File name** parameter set to `yellowlily.jpg`.

The model inserts two strings into the input image at different locations by using the Insert Text block with these parameters:

- **Text** — `'Lilly %d'`
- **Color value source** — `Input port`
- **Location source** — `Specify via dialog`
- **Location [x y]** — `[100 50;800,50]`
- **Opacity** — `1`
- **Font face** — `LucidaSansDemiBold`
- **Font size (points)** — `100`

The `Specify two variables` Constant (Simulink) block changes the text of the inserted strings, and the `Specify two color source` Constant (Simulink) block changes their color.

Simulate Model

Run the model. The model uses a Video Viewer block to display the output image with an inserted pair of bold text strings in different colors and at different locations.

```
sim(modelname);
```


Output image
File
Tools
View
Simulation
Help
Lilly 1
Lilly 2
Ready
Magnification: 37%
RGB:1632x1224
T=10.000

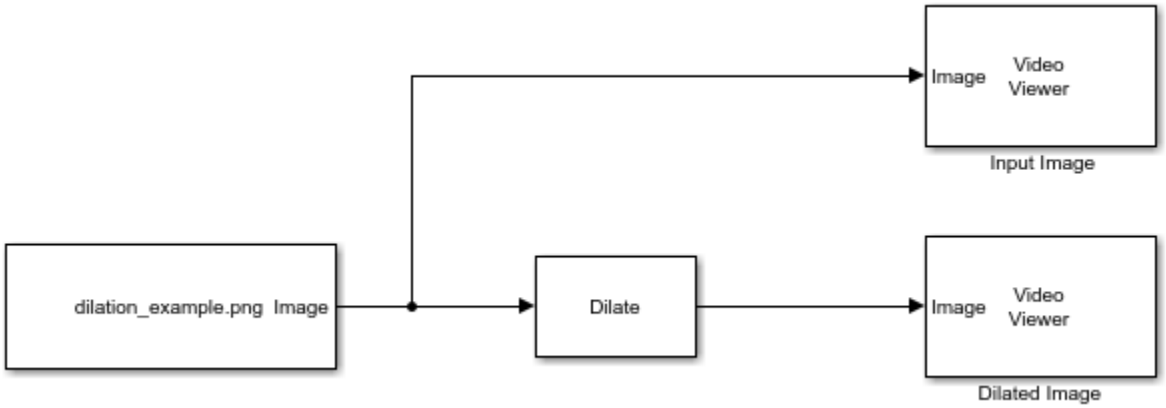
Dilation of Binary Image

This example shows how to increase the size of a binary image object and fill holes in it.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blockDilation.slx';
open_system(modelname);
```

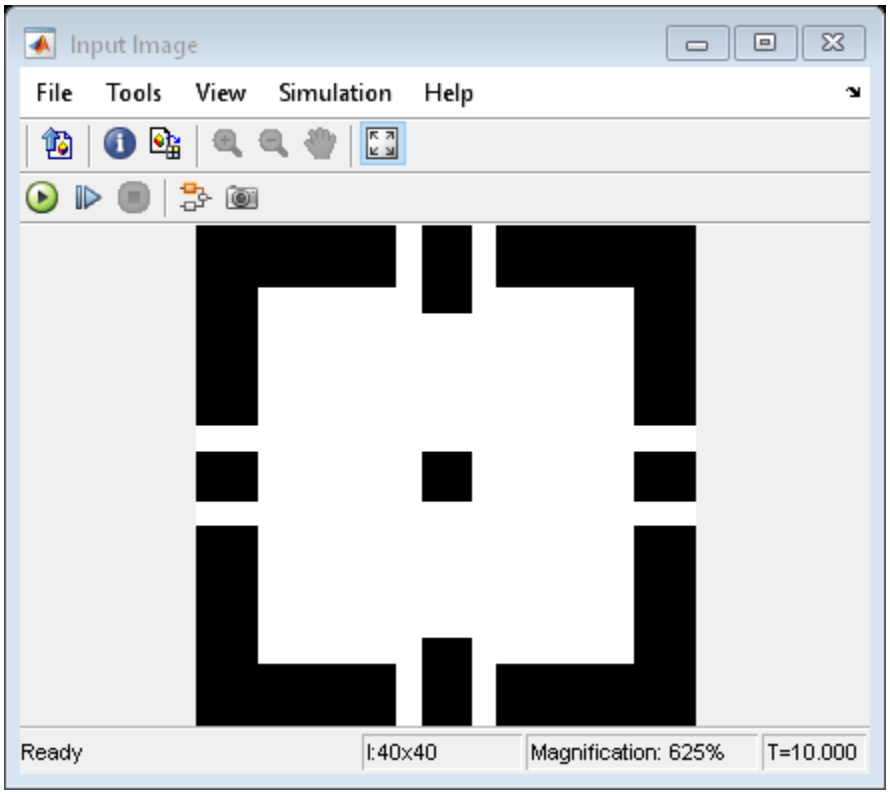


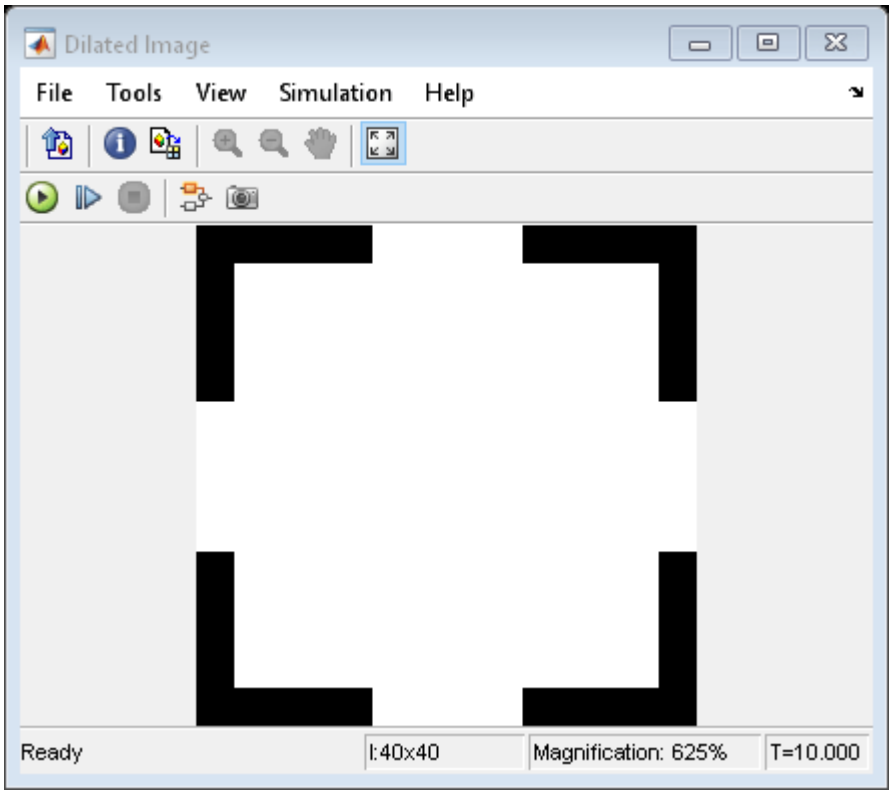
The model reads an input binary image using the Image From File block. The Dilation block dilates the input image using a 5-pixel wide square structuring element. The model displays the resulting dilated image using the Video Viewer block.

Simulate and Display Results

Run the model to visualize the dilated image. The Dilation block fills the holes in the white object and also increases its size.

```
sim(modelname);
```


Input Image
File Tools View Simulation Help
Ready
I:40x40
Magnification: 625%
T=10.000


Dilated Image
File Tools View Simulation Help
Ready
I:40x40
Magnification: 625%
T=10.000

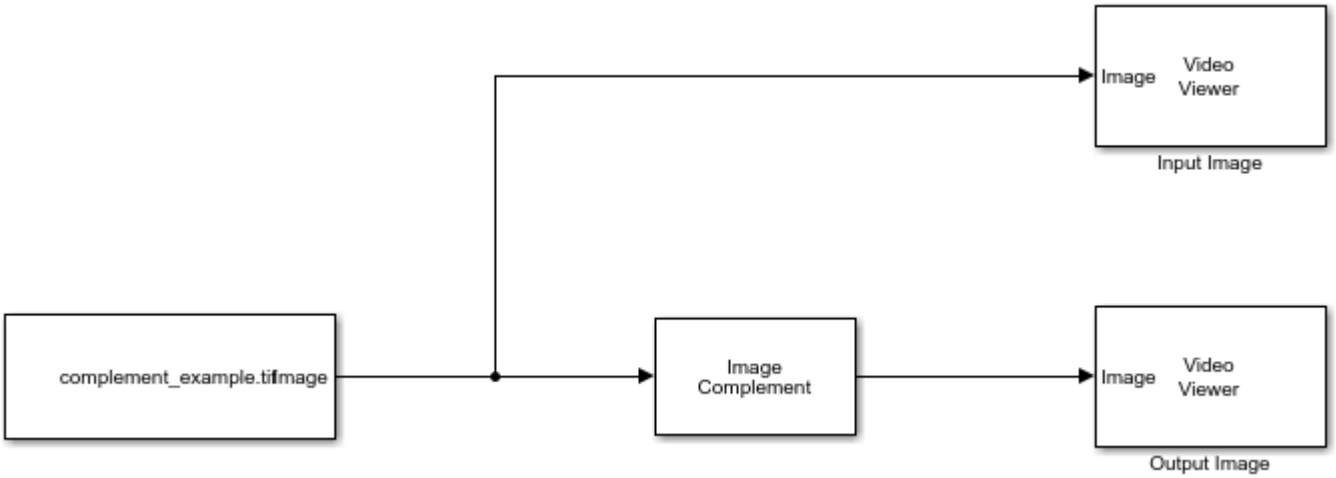
Find Complement of Intensity Image

This example shows how to obtain the complement of an intensity image.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blockImageComplement.slx';
open_system(modelname);
```

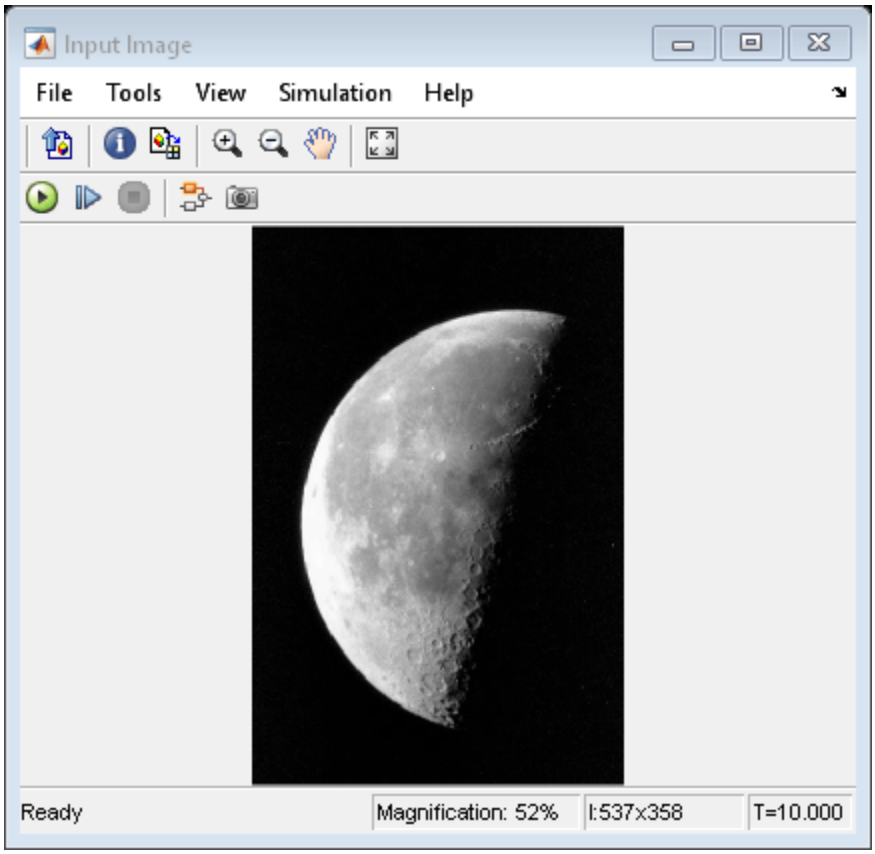


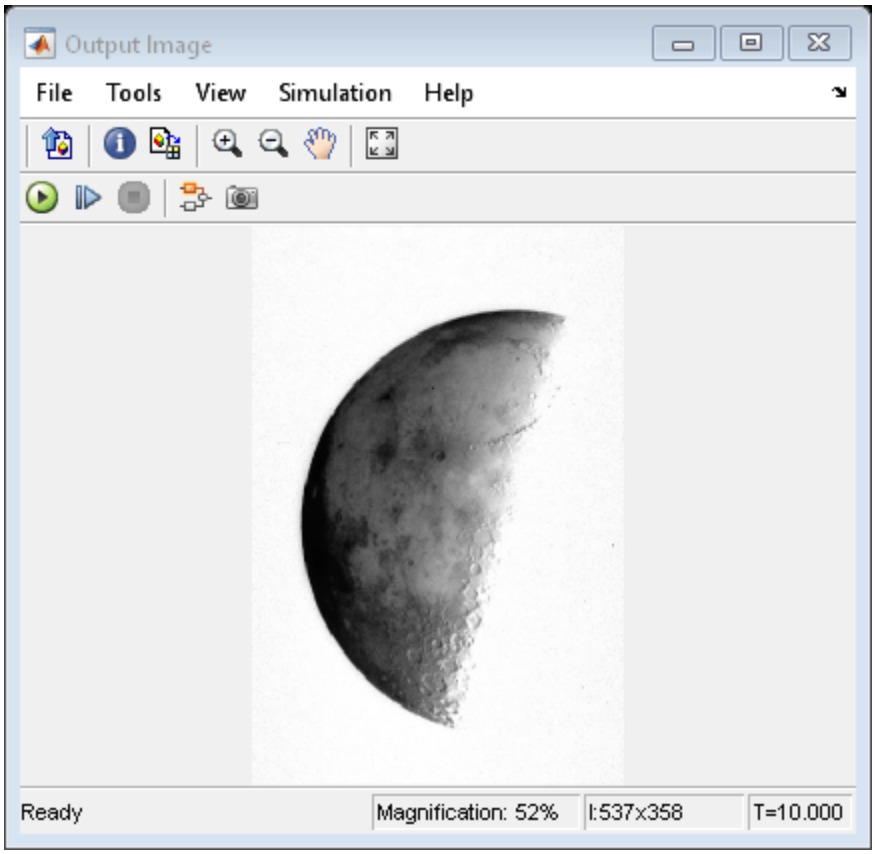
The model reads an input intensity image using the Image From File block. The Image Complement block computes the complement of this image. The model displays the resulted output image using the Video Viewer block.

Simulate and Display Results

Run the model to visualize the output image. It can clearly be seen that the dark areas became lighter and the light areas became darker in the output image.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
Magnification: 52%
I:537x358
T=10.000


Output Image
File
Tools
View
Simulation
Help
Ready
Magnification: 52%
I:537x358
T=10.000

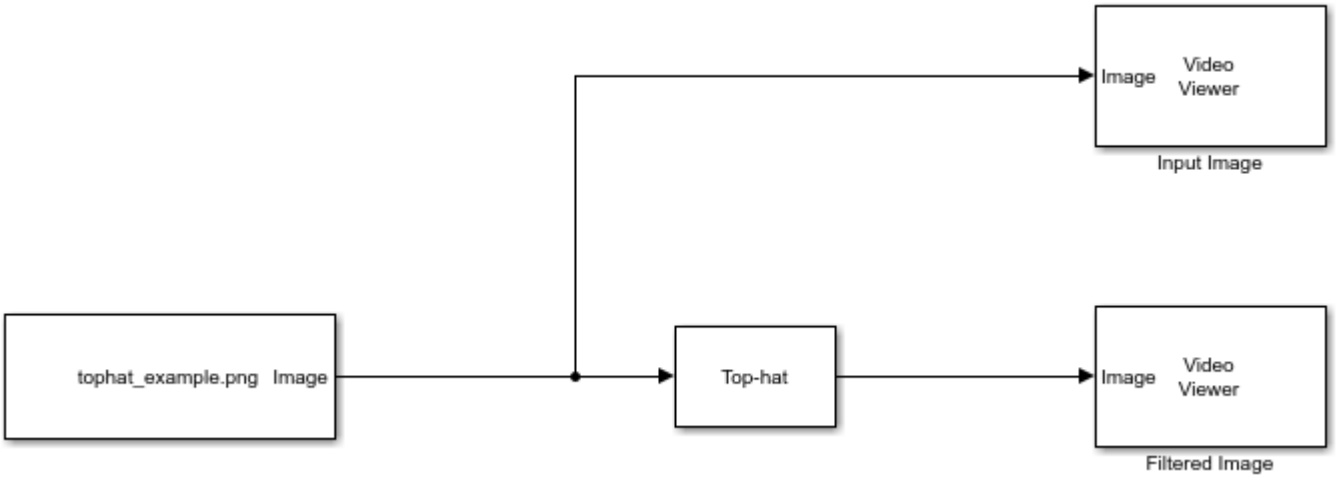
Perform Top-Hat Filtering of Binary Image

This example shows how to perform top-hat filtering on a binary image object.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blockTophat.slx';
open_system(modelname);
```

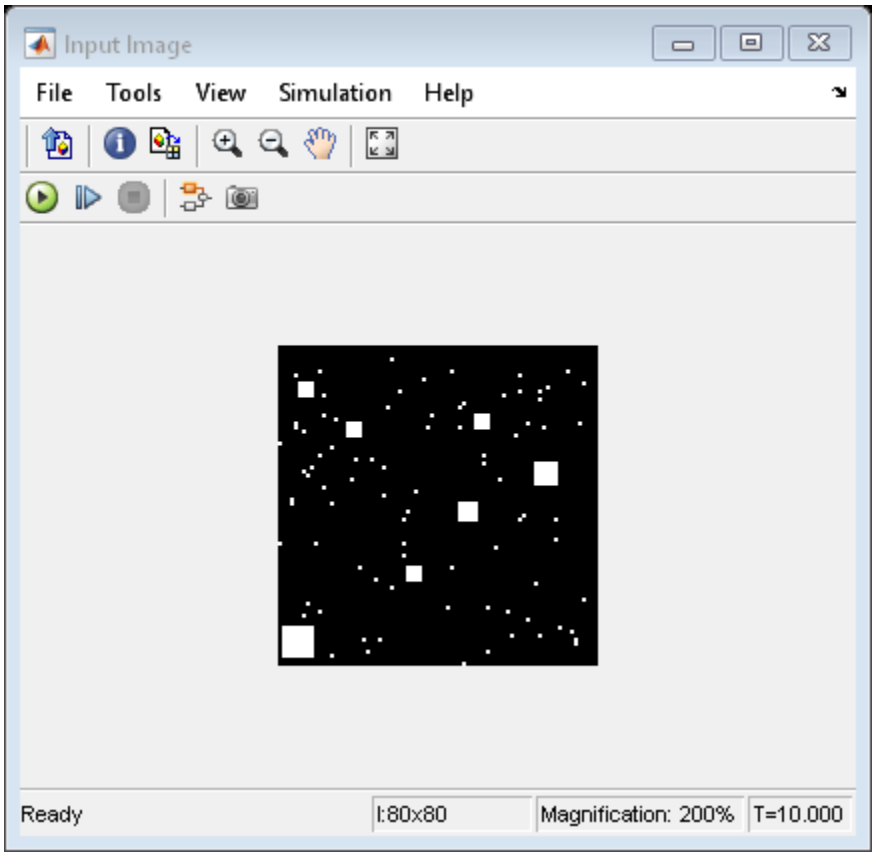


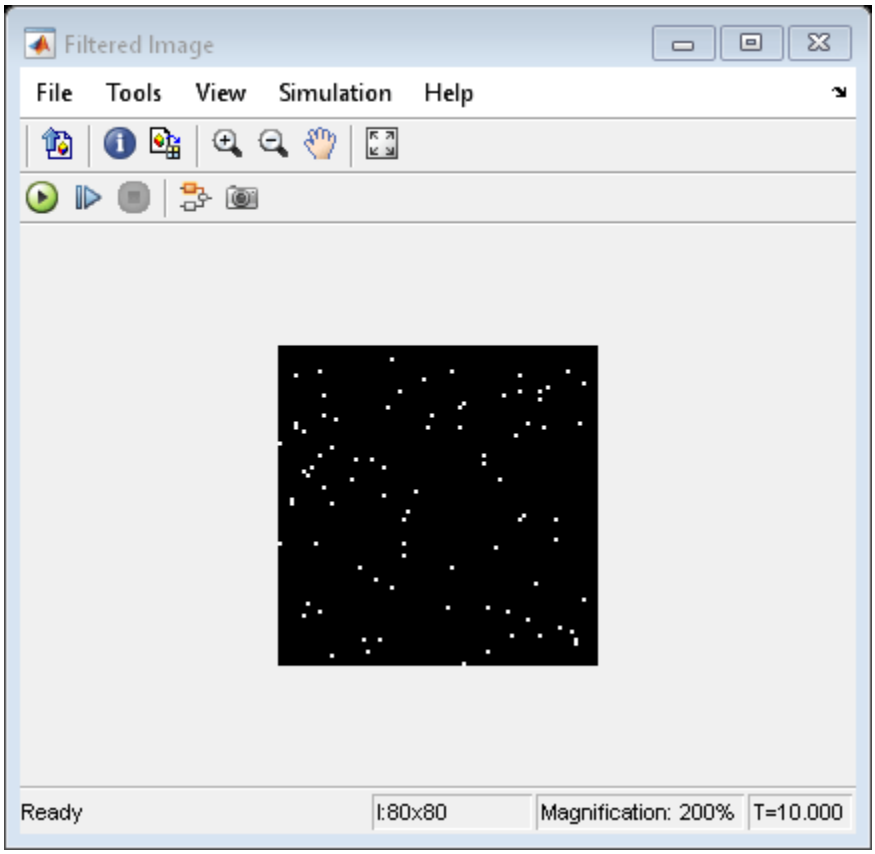
The model reads an input binary image using the Image From File block. The Top-hat block performs top-hat filtering on the input image using a square-shaped structuring element of width 4. The Input image type parameter of the block is set to 'Binary'. The model displays the resulted filtered image using the Video Viewer block.

Simulate and Display Results

Run the model to visualize the filtered image. The Top-hat block first performs the opening operation on the input image and then subtracts the result of this operation from the input image. Here, the block removes the white objects that are larger than the structuring element and retains the smaller white objects, as can be seen.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
I:80x80
Magnification: 200%
T=10.000


Filtered Image
File
Tools
View
Simulation
Help
Ready
I:80x80
Magnification: 200%
T=10.000

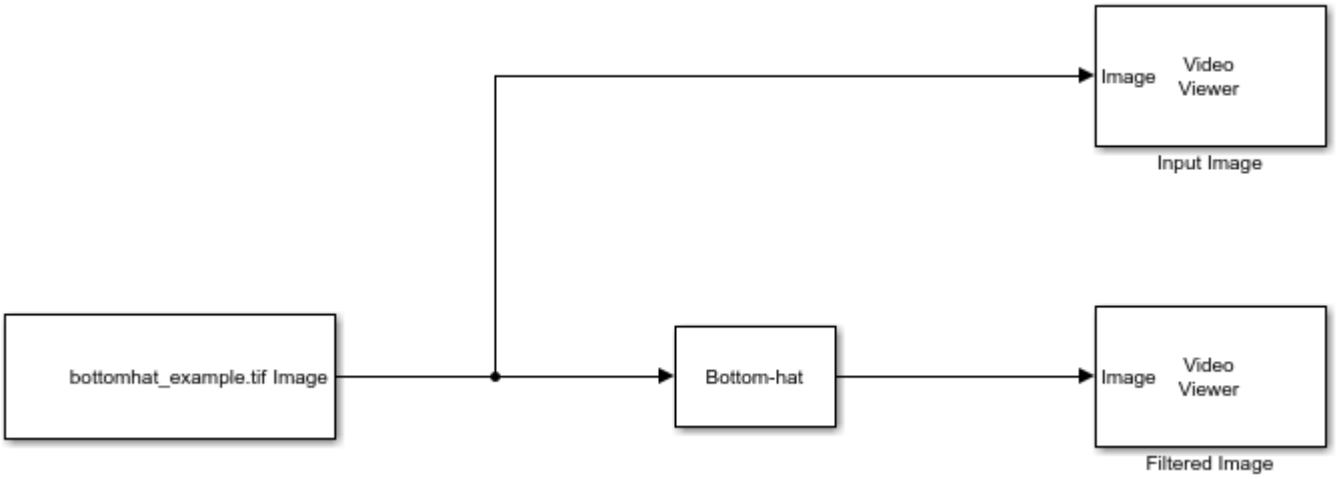
Perform Bottom-hat Filtering of Binary Image

This example shows how to perform bottom-hat filtering on a binary image object.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blockBottomhat.slx';
open_system(modelname);
```

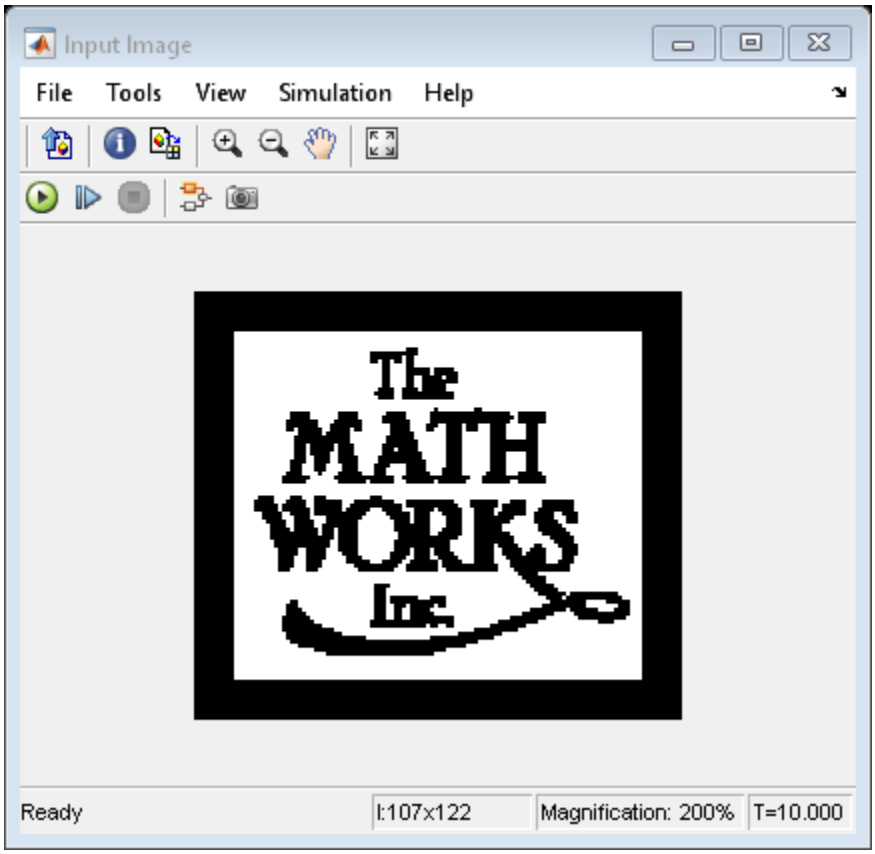


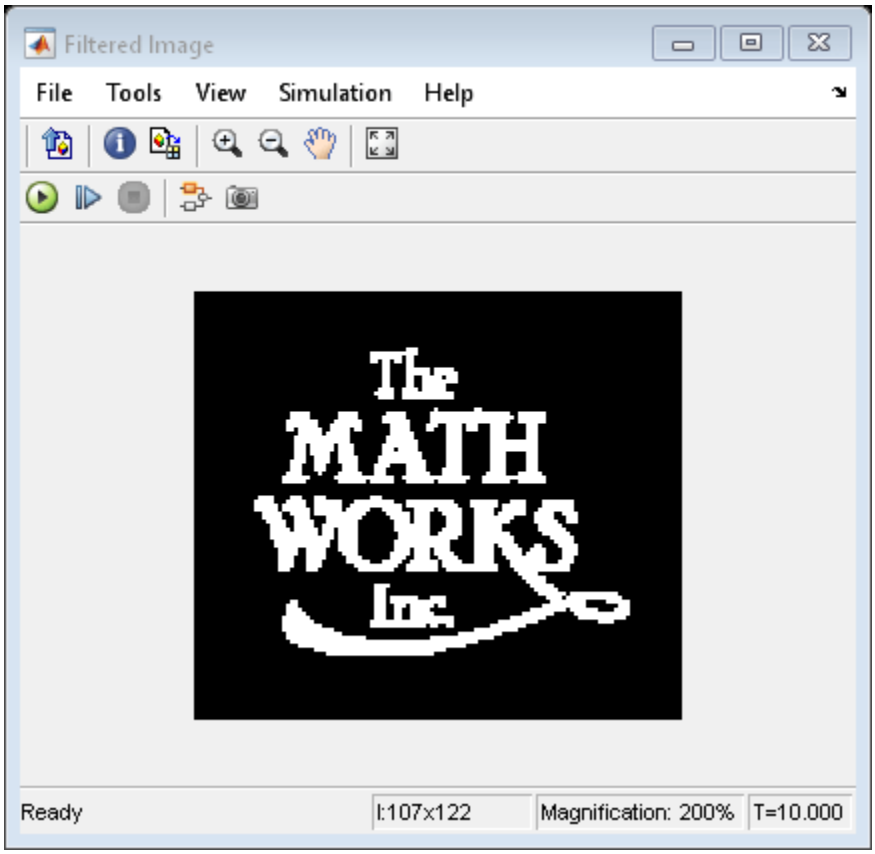
The model reads an input binary image using the Image From File block. The Bottom-hat block performs bottom-hat filtering on the input image using a disk-shaped structuring element of radius 5. The Input image type parameter of the block is set to 'Binary'. The model displays the resulted filtered image using the Video Viewer block.

Simulate and Display Results

Run the model to visualize the filtered image. The Bottom-hat block first performs the closing operation on the input image and then subtracts the input image from the result of this operation. Hence, the logo which was originally the image background, is now the white image object, as can be seen.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
The
MATH
WORKS
Inc.
Ready
I:107x122
Magnification: 200%
T=10.000


Filtered Image
File
Tools
View
Simulation
Help
The
MATH
WORKS
Inc.
Ready
I:107x122
Magnification: 200%
T=10.000

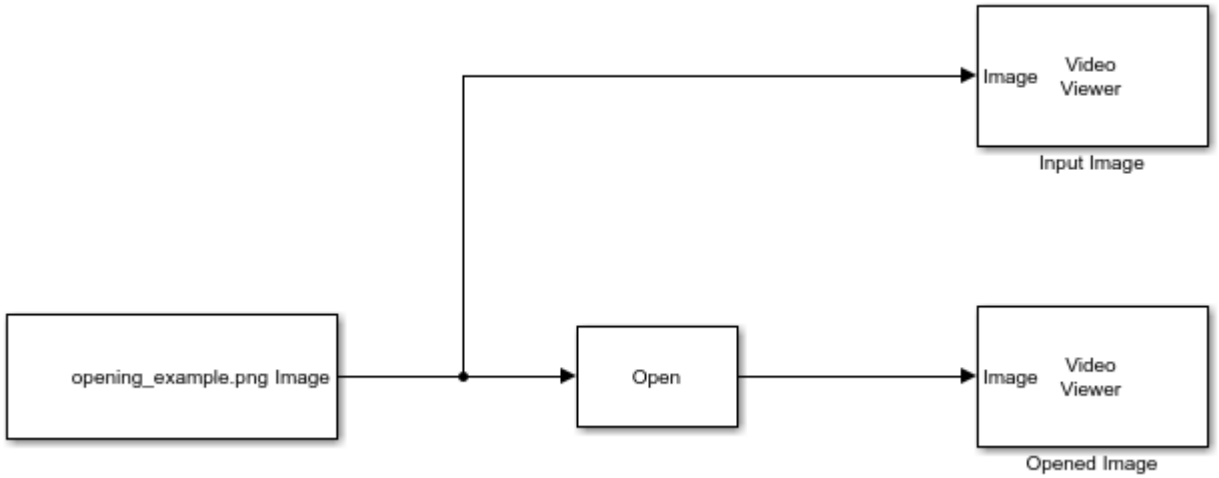
Perform Opening of Binary Image

This example shows how to break narrow bridges between two main sections of a binary image object and eliminate thin protruding elements.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blockOpening.slx';
open_system(modelname);
```

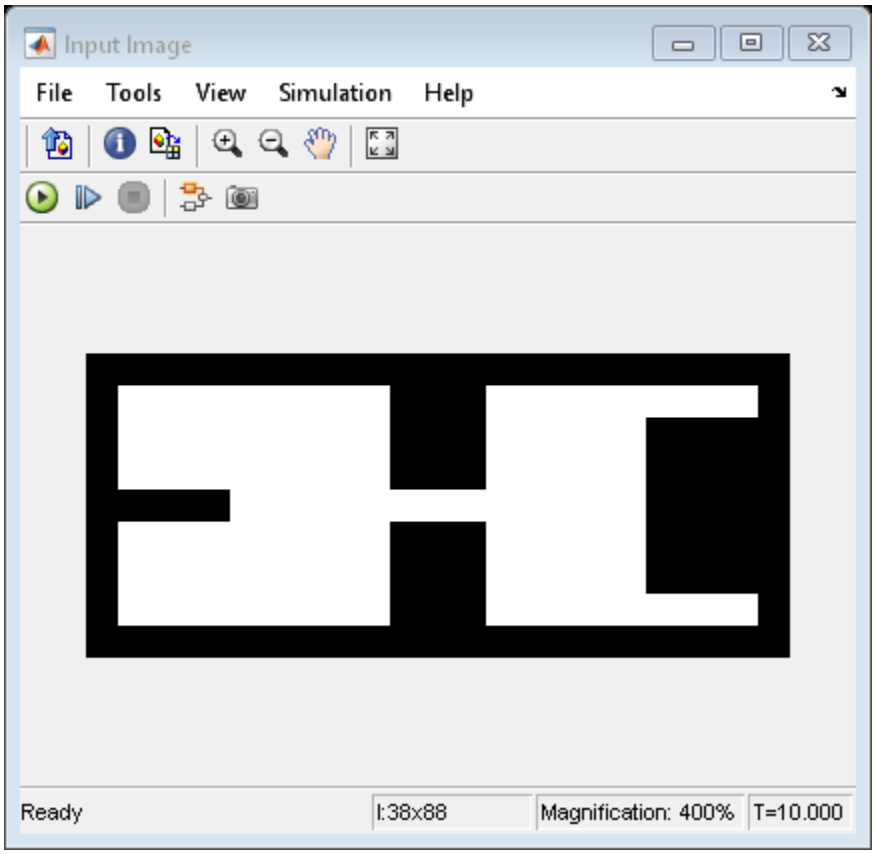


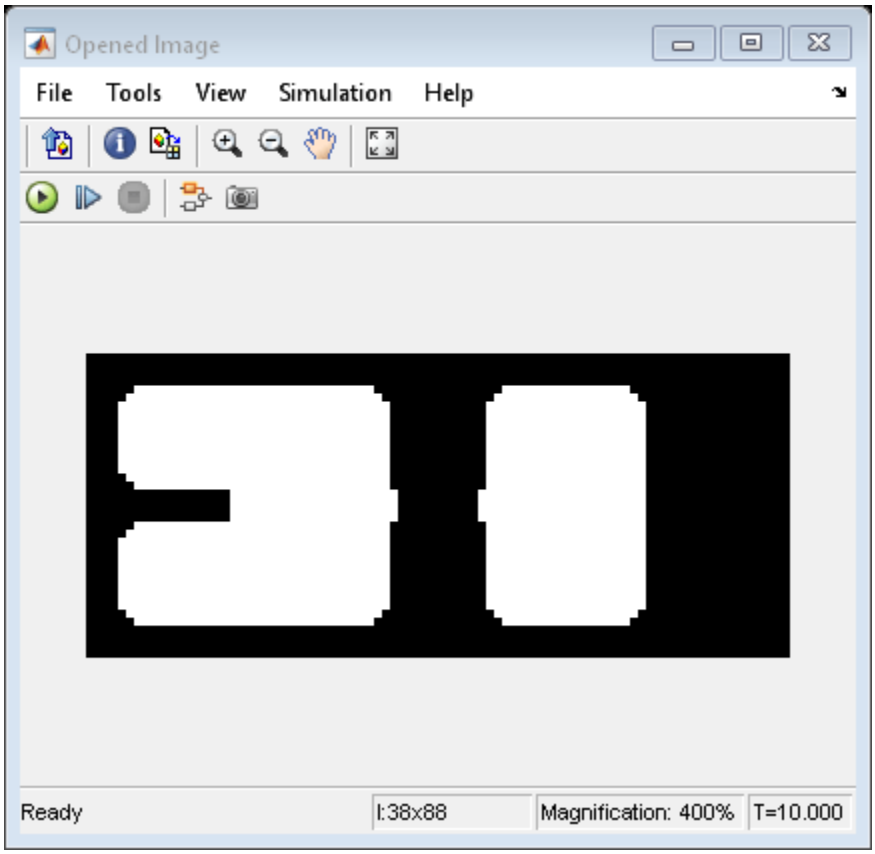
The model reads an input binary image using the Image From File block. The Opening block performs the opening operation on the input image using a disk-shaped structuring element with a radius of 4. The model displays the resulted opened image using the Video Viewer block.

Simulate and Display Results

Run the model to visualize the opened image. The Opening block eliminates the narrow connecting bridge and the thin protrusions of the white object, as can be seen.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
I:38x88
Magnification: 400%
T=10.000


Opened Image
File
Tools
View
Simulation
Help
Ready
I:38x88
Magnification: 400%
T=10.000

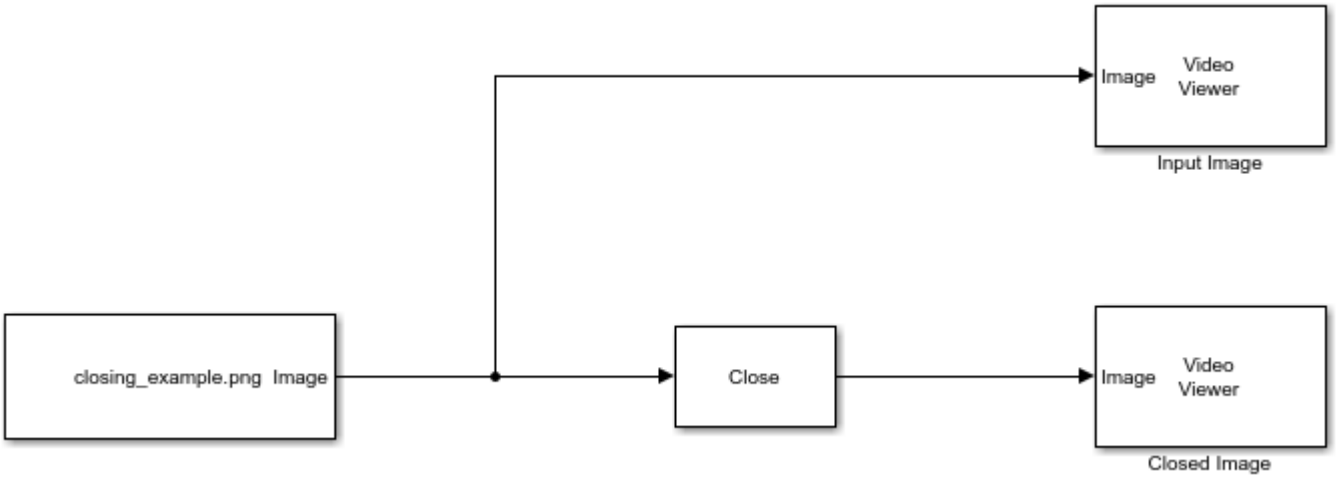
Perform Closing of Binary Image

This example shows how to eliminate small holes in a binary image by using closing operation.

Example Model

Open the Simulink® model.

```
modelname = 'ex_blockClosing.slx';
open_system(modelname);
```

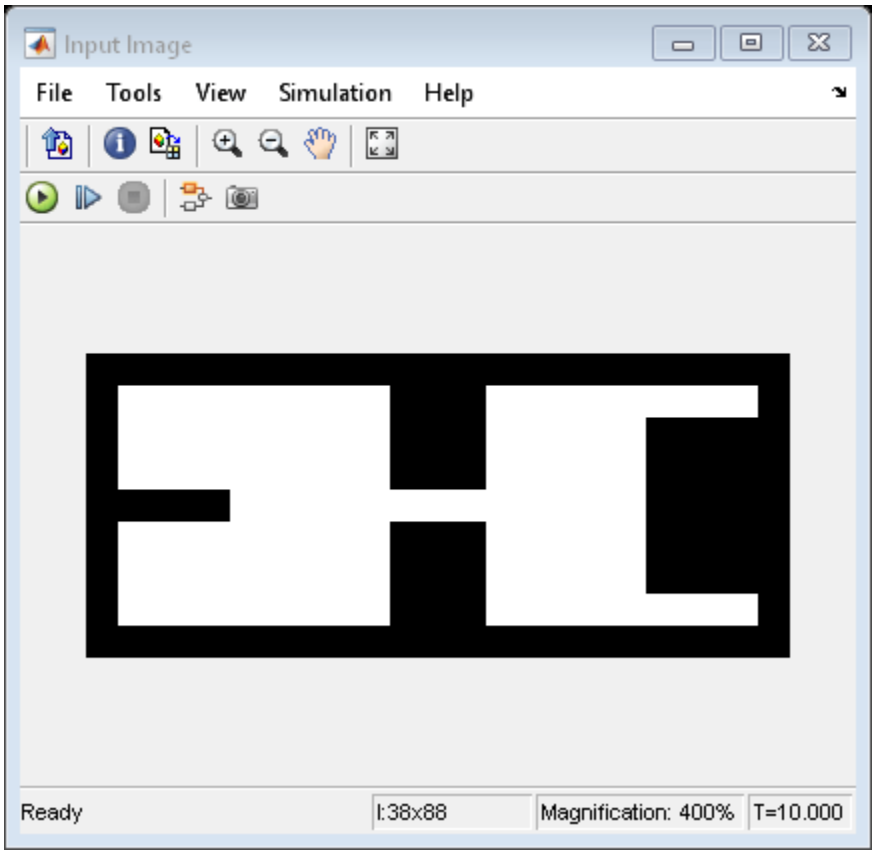


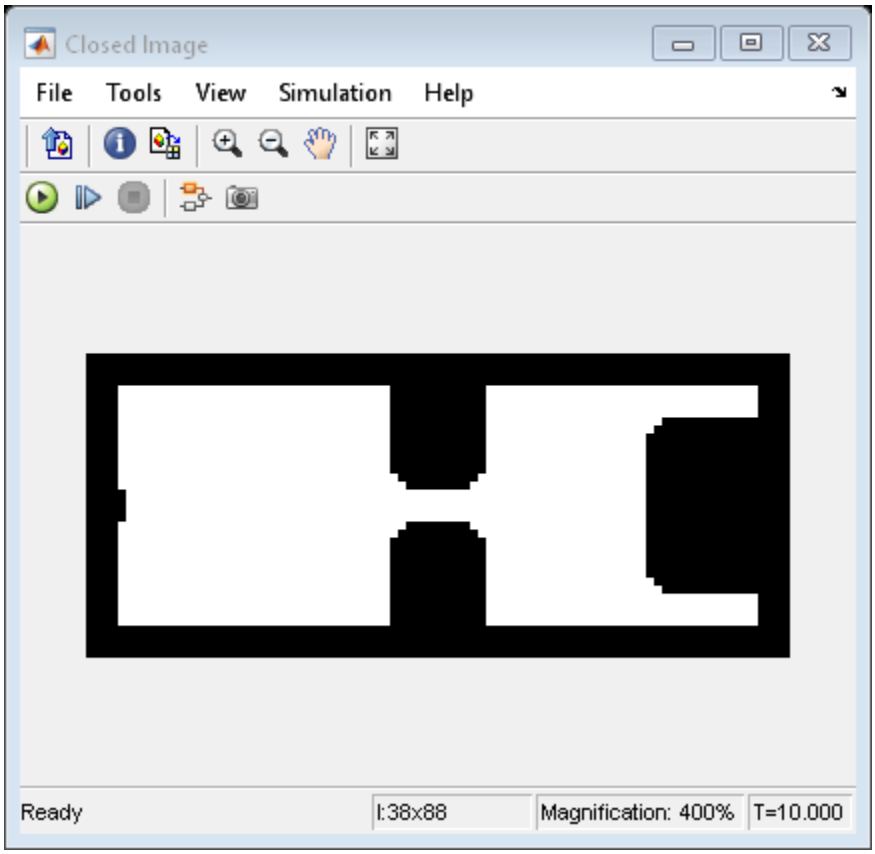
The model reads an input binary image by using the `Image From File` block. The `Closing` block performs the closing operation on the input image. The model uses a disk-shaped structuring element with a radius of 4. The model displays the output image by using the `Video Viewer` block.

Simulate and Display Results

Run the model and display the result. You can see that the closing operation reduced the size of the leftmost intrusion on the boundary of the white object and smoothed the sections of the contour.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
I:38x88
Magnification: 400%
T=10.000


Closed Image
File
Tools
View
Simulation
Help
Ready
I:38x88
Magnification: 400%
T=10.000

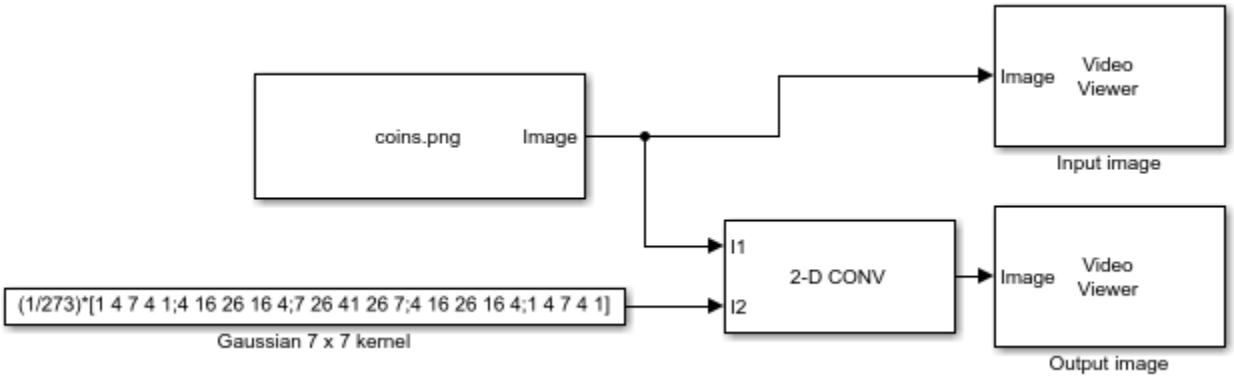
Blur Image Using Gaussian Kernel

This example shows you how to perform 2-D convolution to blur an image using the Gaussian kernel.

Example Model

Open the Simulink® model.

```
modelname = "ex_blk2DConvolution.slx";
open_system(modelname);
```



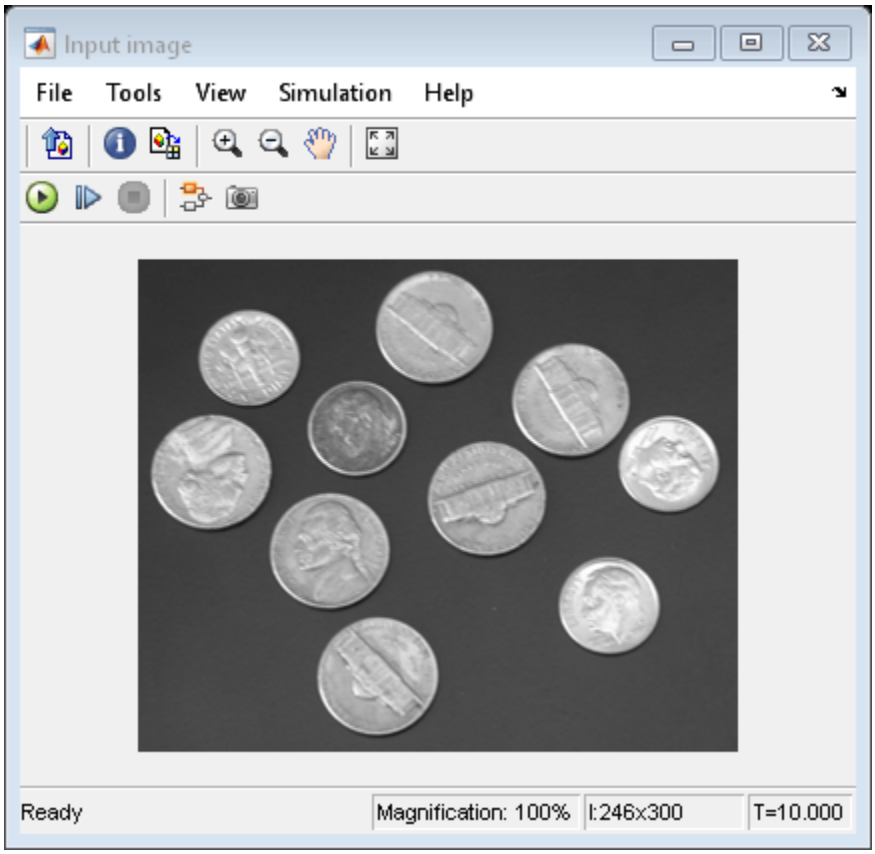
This model reads a PNG image using the Image From File block, which outputs it as a matrix of data type `double`.

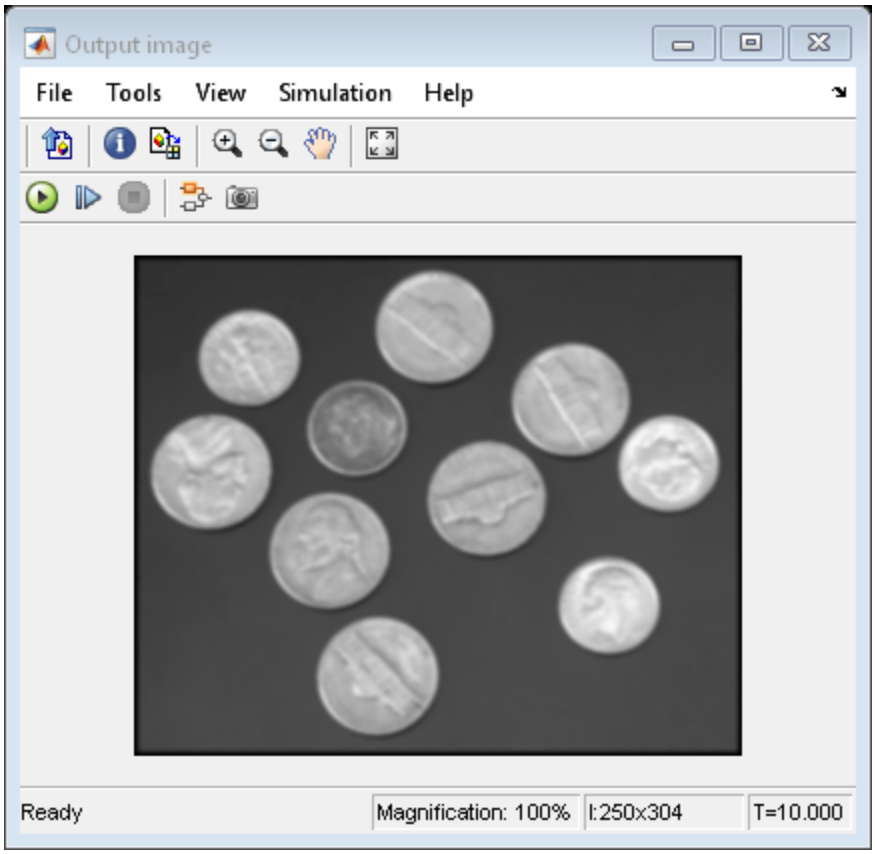
The model then blurs the image by using a 2-D Convolution block to convolve the input image with a 7-by-7 representation of the Gaussian kernel.

Simulate the Model

Run the model. The model displays the input image and the blurred output image using Video Viewer blocks.

```
sim(modelname);
```


Input image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:246x300
T=10.000


Output image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:250x304
T=10.000

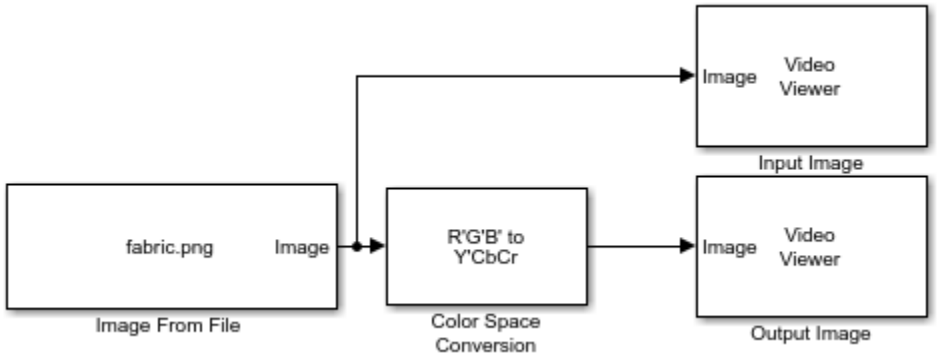
Convert Image Color Space from RGB to YCbCr

This example shows you how to convert an image color space from RGB to YCbCr.

Example Model

Open the Simulink® model.

```
modelname = "ex_blkconvertcolorspace.slx";
open_system(modelname);
```



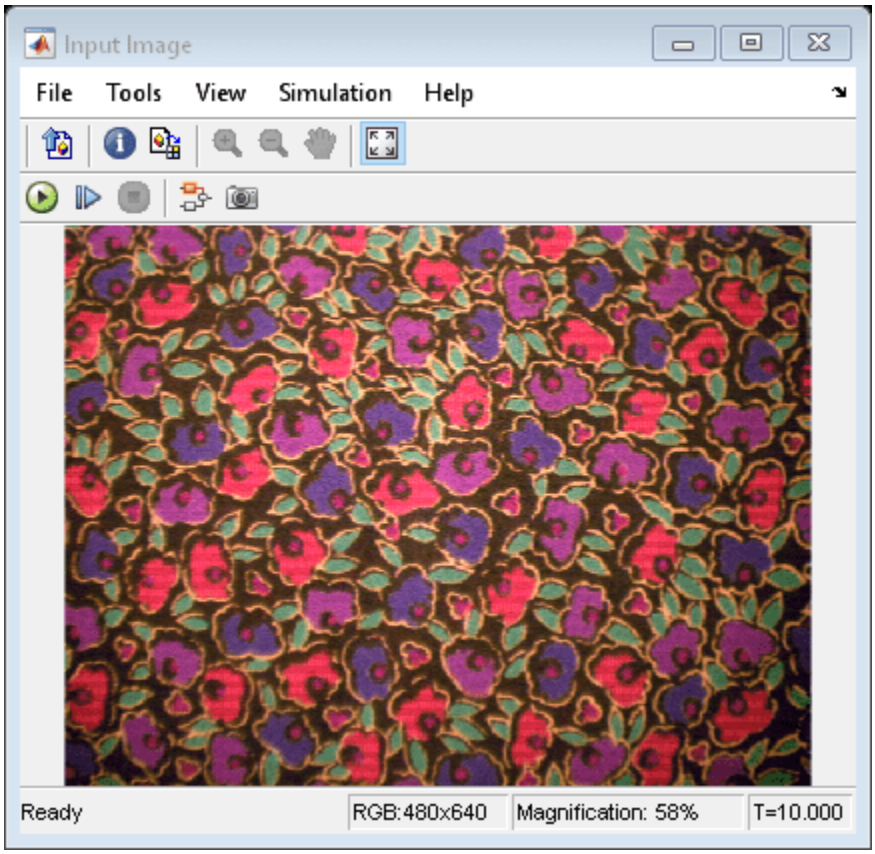
This model reads an input image using the Image From File block, then converts the input image from the RGB color space to the YCbCr color space by using a Color Space Conversion block with these parameter values:

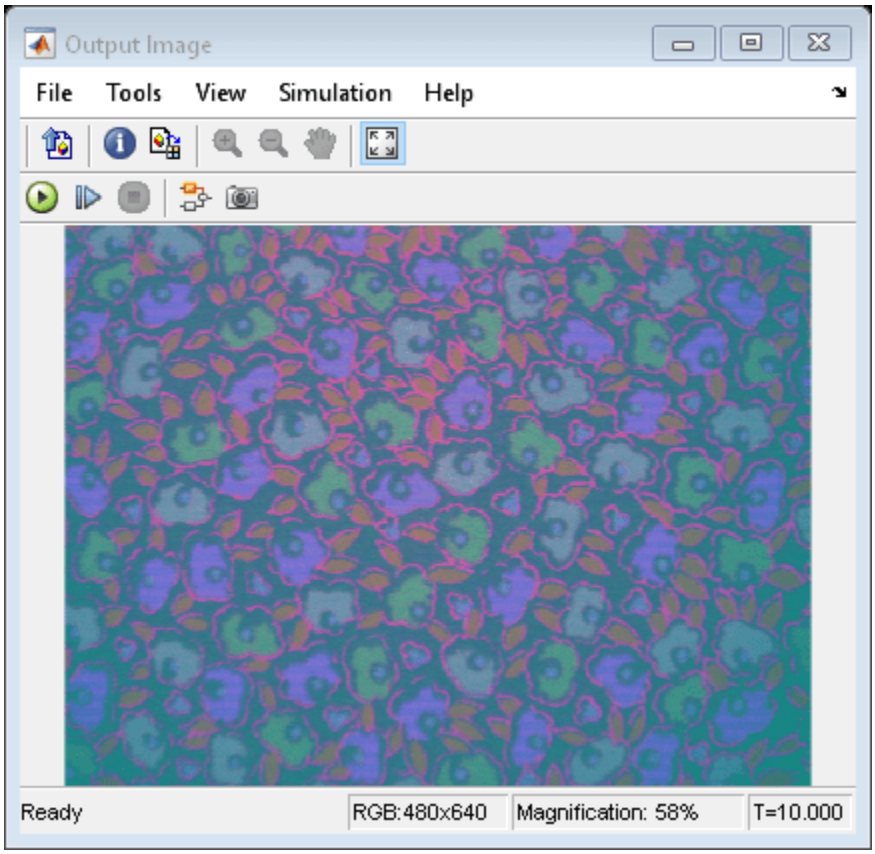
- **Conversion** — `R'G'B' to Y'CbCr`
- **Use conversion specified by** — `Rec. 601 (SDTV)`
- **Image signal** — `One multidimensional signal`

Simulate the Model

Run the model. The model displays the RGB color space input image and the converted YCbCr color space output image using Video Viewer blocks.

```
sim(modelname);
```


Input Image
File
Tools
View
Simulation
Help
Ready
RGB:480x640
Magnification: 58%
T=10.000


Output Image
File
Tools
View
Simulation
Help
Ready
RGB:480x640
Magnification: 58%
T=10.000

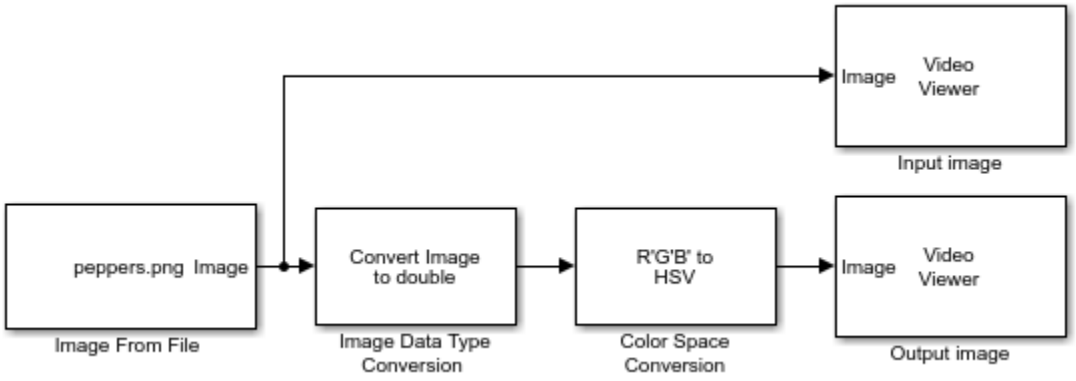
Convert Data Type and Color Space of Image from RGB to HSV

This example shows you how to convert the data type and color space of an input image from the RGB to the HSV color space.

Example Model

Open the Simulink® model.

```
modelname = "ex_blkconvertdatatypeandcolorspace.slx";
open_system(modelname);
```



This model reads an input image using an Image From File block with the **File name** parameter set to `peppers.png`. The input image has a data type of `uint8`.

To convert the image data type to double, the model uses an Image Data Type Conversion block with **Output data type** set to `double`.

To convert the input image from the RGB to the HSV color space, the model uses a Color Space Conversion block with these parameters:

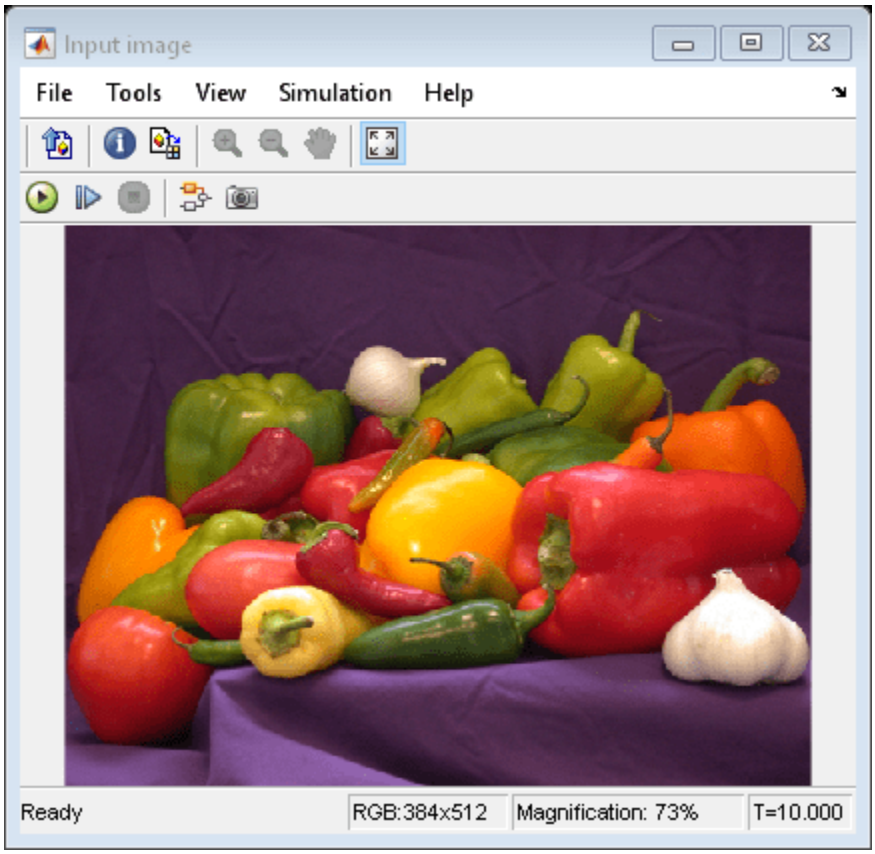
- **Conversion** — `R'G'B' to HSV`
- **Image signal** — `One multidimensional signal`

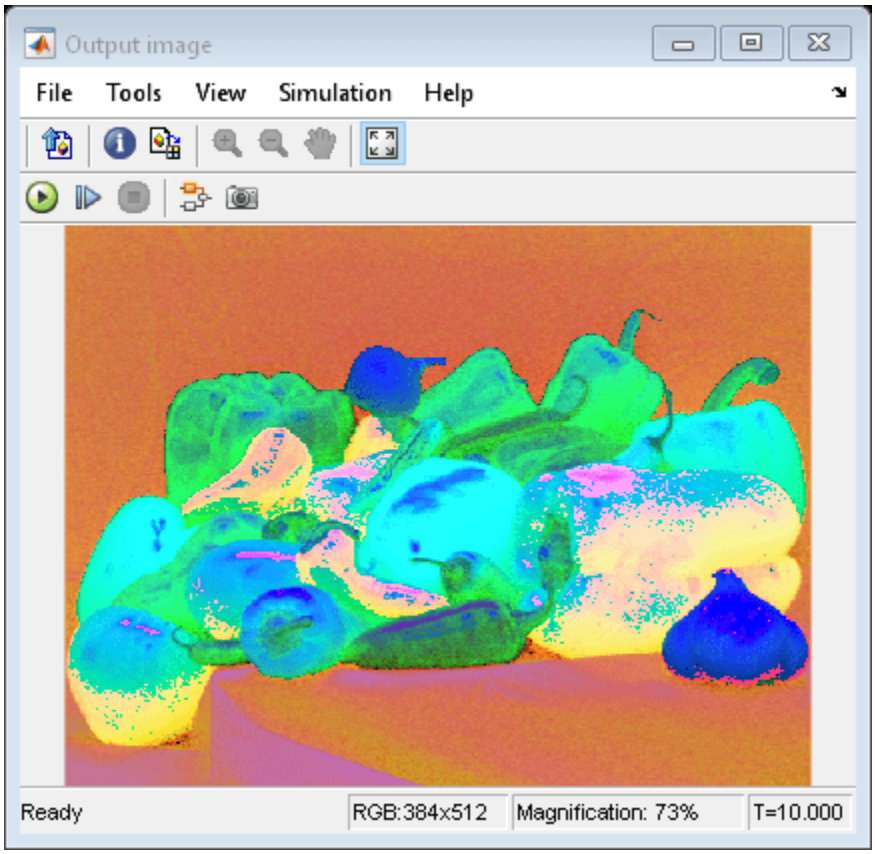
The Video Viewer blocks display the input RGB image and the output HSV image.

Simulate the Model

Run the model. The Video Viewer blocks display the input image and the output image converted to the HSV color space.

```
sim(modelname);
```


Input image
File
Tools
View
Simulation
Help
Ready
RGB:384x512
Magnification: 73%
T=10.000


Output image
File
Tools
View
Simulation
Help
Ready
RGB:384x512
Magnification: 73%
T=10.000

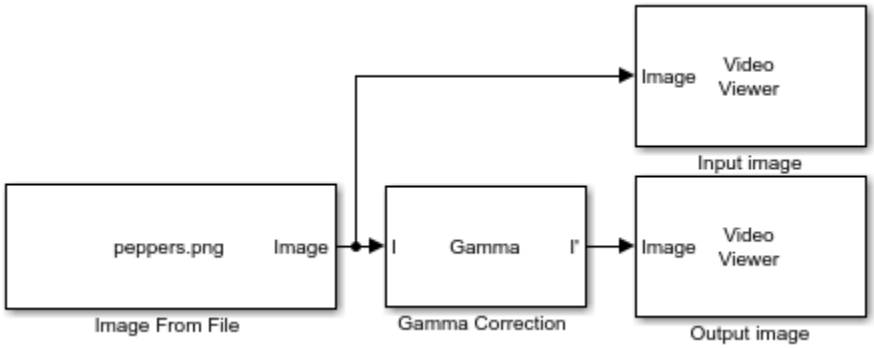
Perform Gamma Correction of Image

This example shows you how to perform gamma correction of an image.

Example Model

Open the Simulink® model.

```
modelname = "ex_blkgammacorrection.slx";
open_system(modelname);
```



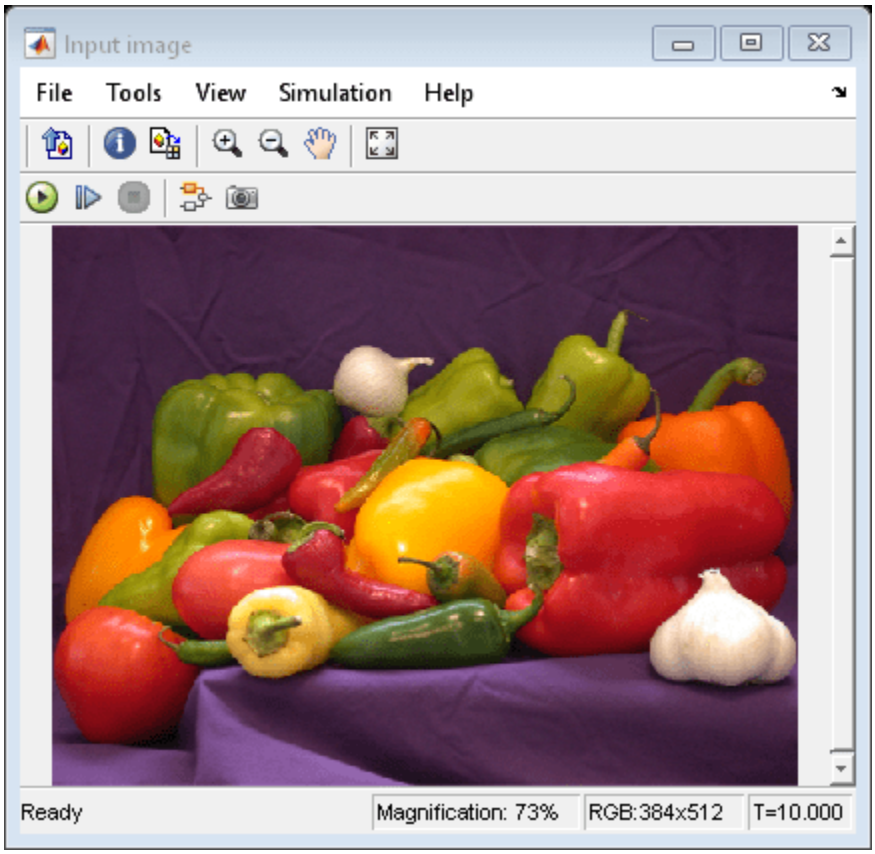
This model reads an input image using an Image From File block with the **File name** parameter set to `peppers.png`.

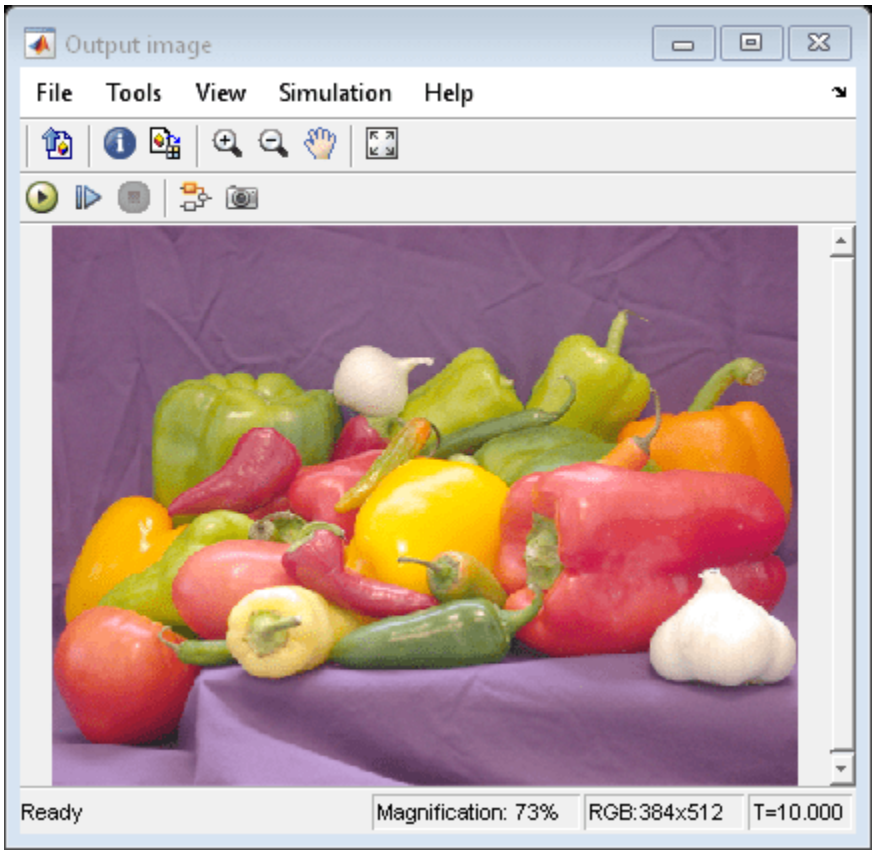
To perform gamma correction of an image, the model uses a Gamma Correction block with default parameters.

Simulate the Model

Run the model. The Video Viewer blocks display the input image and the gamma-corrected output image.

```
sim(modelname);
```


Input image
File
Tools
View
Simulation
Help
Ready
Magnification: 73%
RGB:384x512
T=10.000


Output image
File
Tools
View
Simulation
Help
Ready
Magnification: 73%
RGB:384x512
T=10.000

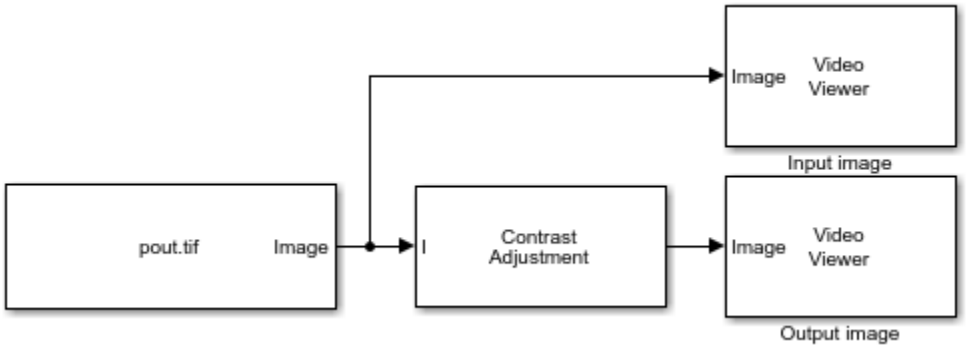
Adjust Contrast of Image

This example shows you how to adjust the contrast of an image by linearly scaling pixel values.

Example Model

Open the Simulink® model.

```
modelname = "ex_blkcontrastadjustment.slx";
open_system(modelname);
```



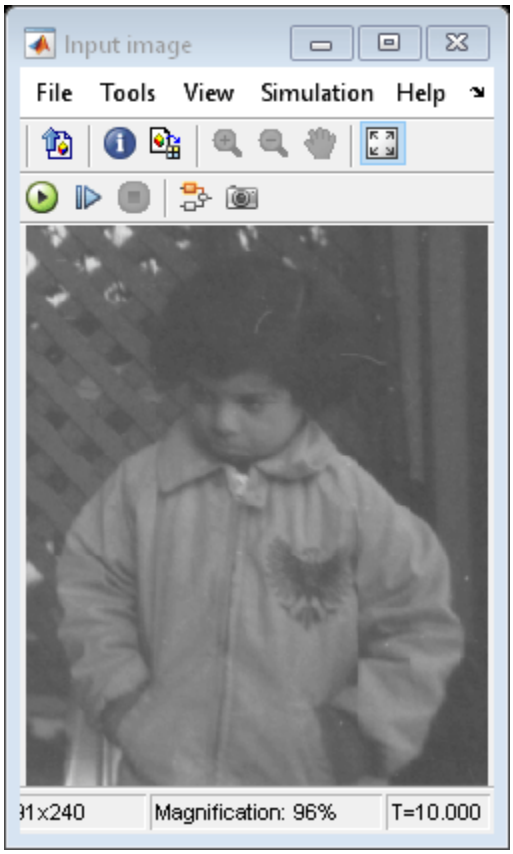
This model reads an input image using an Image From File block with the **File name** parameter set to `pout.tif`.

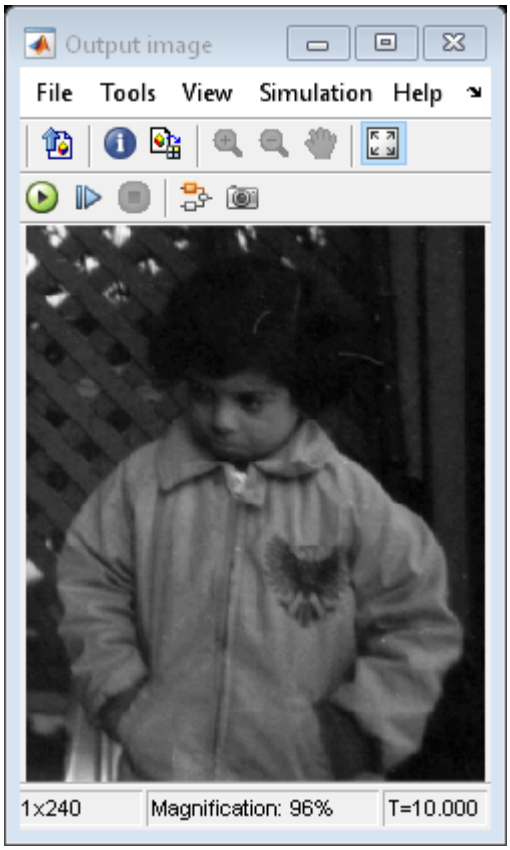
To adjust the contrast of the image by linearly scaling pixel values, the model uses a Contrast Adjustment block.

Simulate the Model

Run the model. The Video Viewer blocks displays the input image and the contrast-adjusted output image.

```
sim(modelname);
```


Input image
File Tools View Simulation Help
91x240
Magnification: 96%
T=10.000


Output image
File Tools View Simulation Help
1x240
Magnification: 96%
T=10.000

Remove Impulse Noise from Image

This example shows you how to remove impulse noise from an image by using a median filter.

Create Noisy Image

Read an image into the MATLAB® workspace.

```
I = imread("coins.png");
```

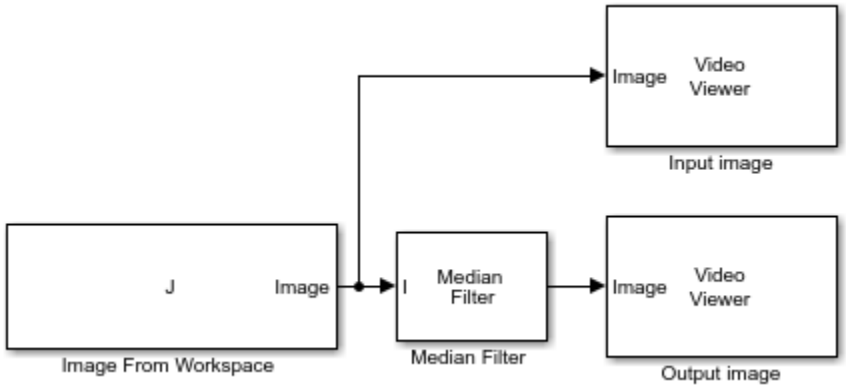
Add salt-and-pepper noise.

```
J = imnoise(I,"salt & pepper",0.03);
```

Explore Model

Open the simulink® model. The model reads the image stored in the variable `J` from the MATLAB workspace by using an Image From Workspace block.

```
modelname = "ex_blkmedianfilter.slx";
open_system(modelname);
```



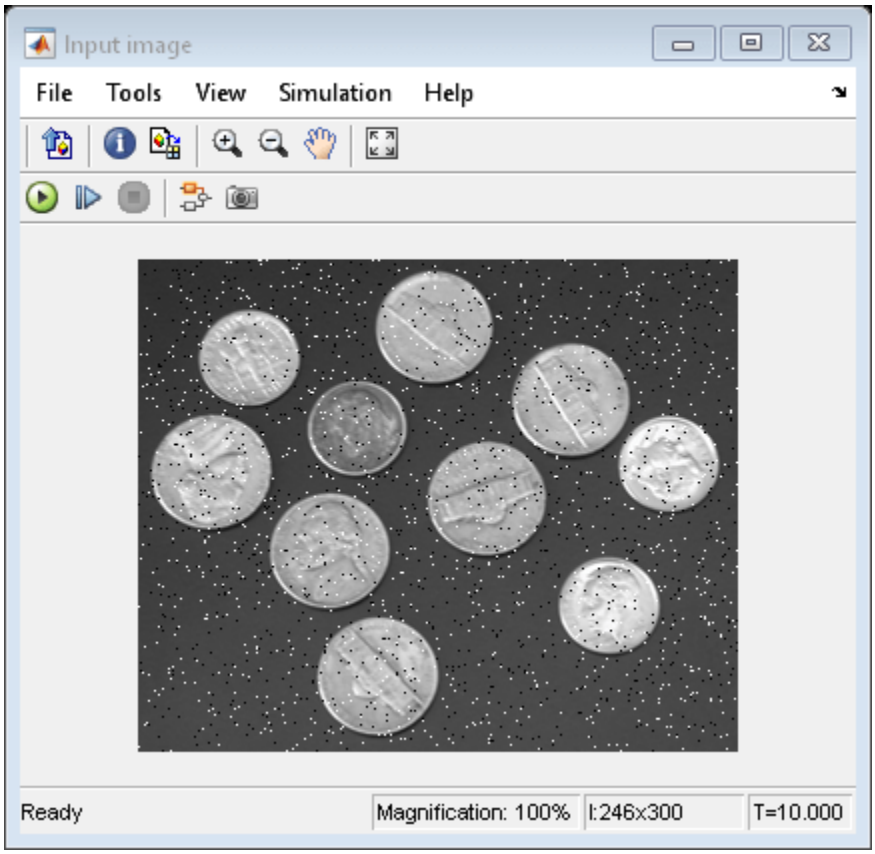
To remove impulse noise, the model uses a Median Filter block with the default parameters.

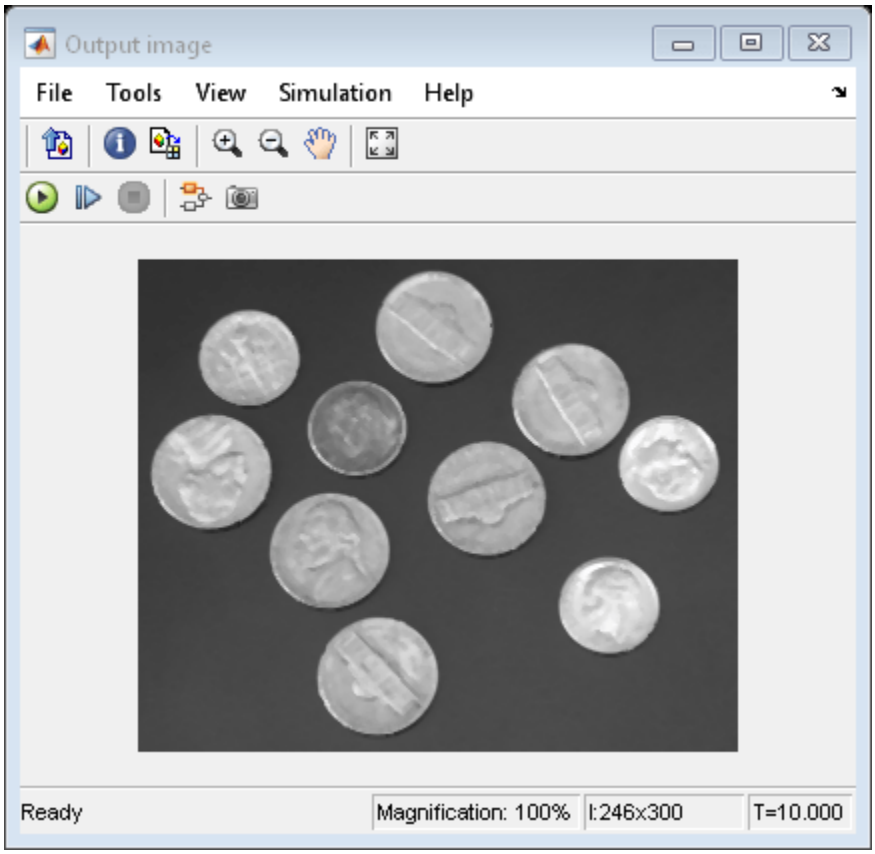
The Video Viewer blocks display the noisy image and the median-filtered output image.

Simulate the Model

Run the model. The Video Viewer blocks display the noisy image and the median-filtered output image.

```
sim(modelname);
```


Input image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:246x300
T=10.000


Output image
File
Tools
View
Simulation
Help
Ready
Magnification: 100%
I:246x300
T=10.000

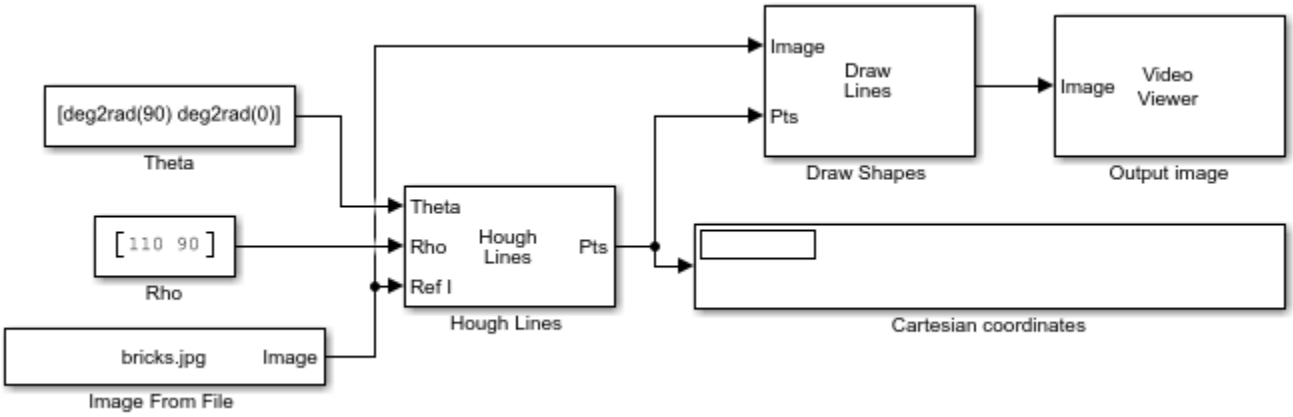
Draw Hough Lines on Image

This example shows you how to find the Cartesian coordinates of lines using rho and theta pairs, and draw those lines on an image.

Example Model

Open the Simulink® model.

```
modelname = "ex_blkhoughlines.slx";
open_system(modelname);
```



This model reads an input image using an Image From File block with the **File name** parameter set to `bricks.jpg`.

To find the Cartesian coordinates of lines by using rho and theta pairs, the model uses a Hough Lines block with the **Sine value computation method** set to `Trigonometric function`.

To draw lines on the image, the model uses a Draw Shapes block with these parameters:

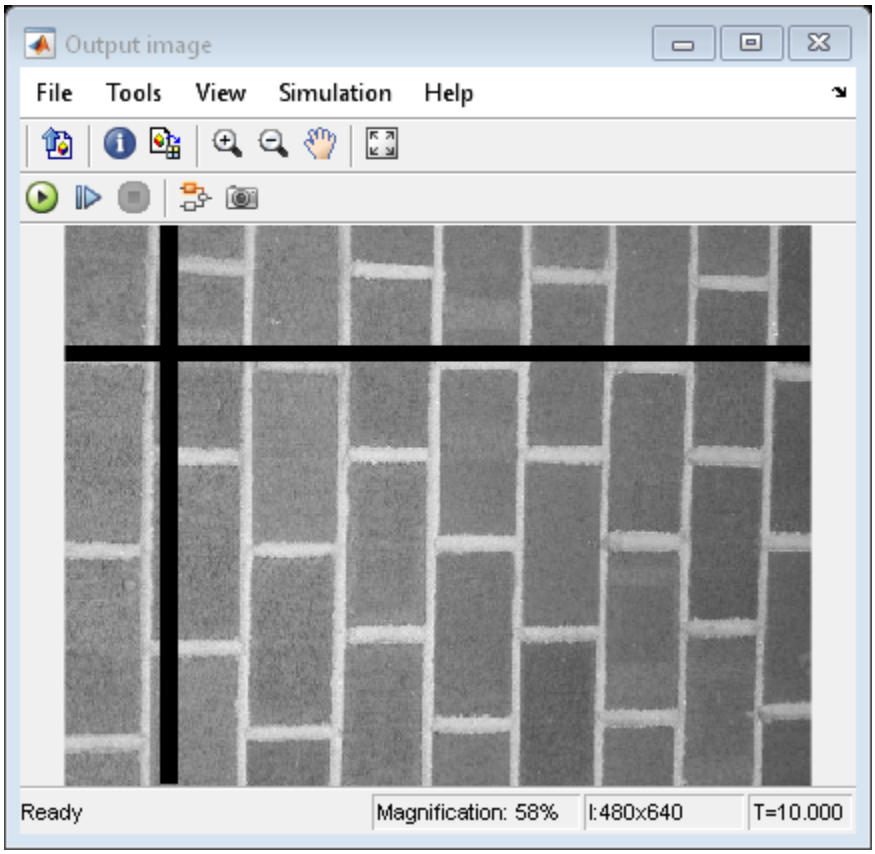
- **Shape** — `Lines`
- **Line width** — 15

The Video Viewer block displays the line-embedded output image.

Simulate the Model

Run the model. The Video Viewer block displays the cartesian coordinates and the line-embedded output image.

```
sim(modelname);
```


Output image
File
Tools
View
Simulation
Help
Ready
Magnification: 58%
I:480x640
T=10.000

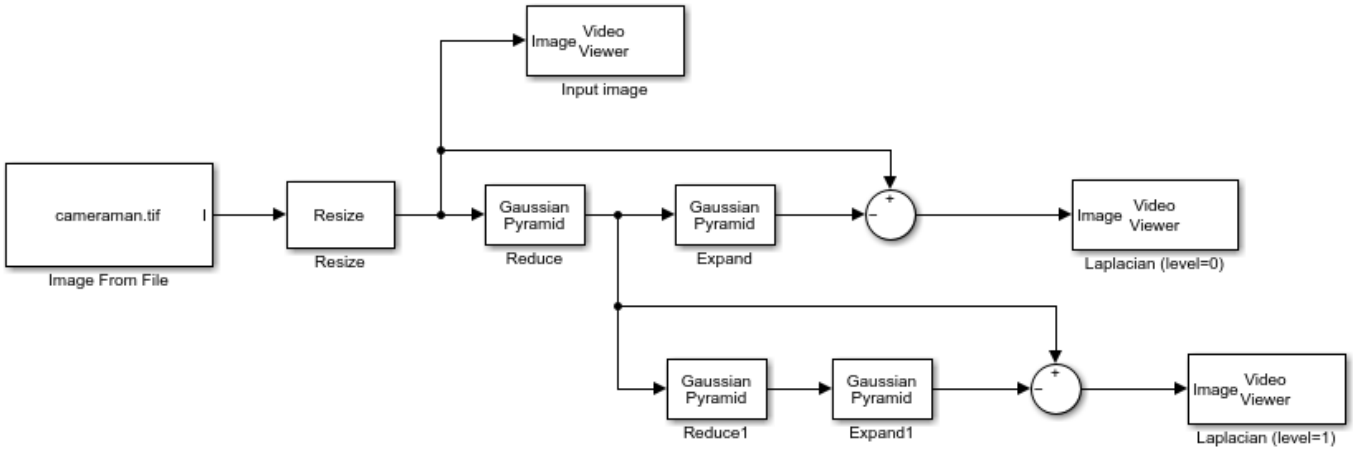
Construct Laplacian Pyramid Image

This example shows you how to construct a Laplacian pyramid image.

Example Model

Open the Simulink® model.

```
modelname = "ex_blklaplasianpyramid.slx";
open_system(modelname);
```



This model reads an input image using an Image From File block with the **File name** parameter set to `cameraman.tif` and **Output data type** parameter set to `single`.

To construct a Laplacian pyramid the Resize block resizes the input image to 253-by-253 pixels, and performs reduce and expand opertions using Gaussian Pyramid blocks. At the initial level, the Laplacian pyramid reduces the resized input image, expands the reduced image, and then subtracts the output from the resized input image. At the next level, the Laplacian pyramid performs a reduce operation on the already reduced image, expands this second-order reduction, and then subtracts it from the first-order reduced image.

The Video Viewer blocks display the input image and output images from each level of the Laplacian pyramid.

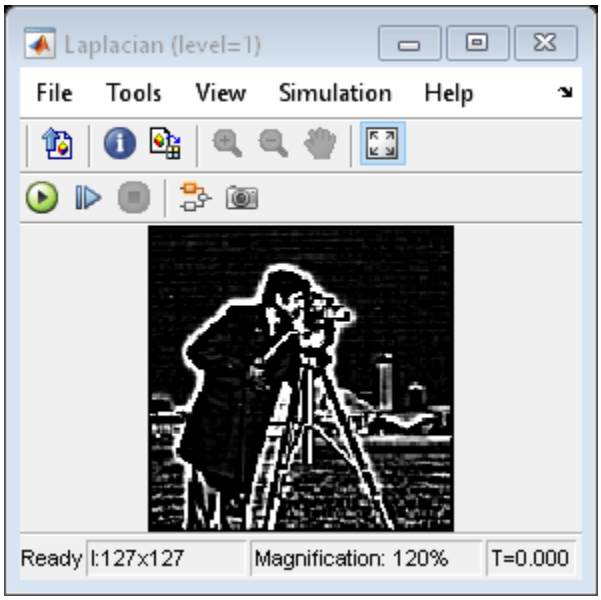
Simulate the Model

Run the model. The Video Viewer blocks display the input image and the Laplacian pyramid output images.

```
sim(modelname);
```


Input image
File Tools View Simulation Help
:253x253
Magnification: 91%
T=0.000


Laplacian (level=0)
File Tools View Simulation Help
Read
I:253x253
Magnification: 91%
T=0.000


Laplacian (level=1)
File
Tools
View
Simulation
Help
Ready
I:127x127
Magnification: 120%
T=0.000