In [3]:
```
import cv2
import glob
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import fftconvolve,convolve2d
import math
```

SSD or Normalized correlation

In [2]:
```
vidcap = cv2.VideoCapture('20220417_151055.mp4')
success,image = vidcap.read()
count = 0
while success:
    success,image = vidcap.read()
    if count%30==0 :
      cv2.imwrite("data/frame%d.jpg" % count, image)     # save frame as JPEG file
      print('Read a new frame: ', success)
    count += 1
```

```
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
Read a new frame:  True
```

In [3]:
```
def ssd(A,B):
    squares = (A[:,:,:3] - B[:,:,:3]) ** 2
    return math.sqrt(np.sum(squares))
```

In [4]:
```
def norm_data(data):
    mean_data=np.mean(data)
    std_data=np.std(data, ddof=1)
    return (data-mean_data)/(std_data)


def ncc(data0, data1):
    return (1.0/(data0.size-1)) * np.sum(norm_data(data0)*norm_data(data1))
```

In [5]:
```
import cv2

imdir = 'data/'
ext = ['png', 'jpg', 'gif']    # Add image formats here

files = []
[files.extend(glob.glob(imdir + '*.' + e)) for e in ext]
```

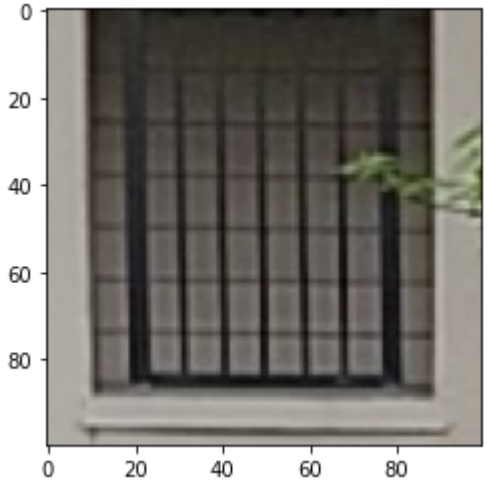
```
images = [cv2.imread(file) for file in files]
```

In [6]:
```
plt.imshow(cv2.cvtColor(images[0], cv2.COLOR_BGR2RGB))
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x237fab99df0>
```

In [7]:
```
cropped_image = images[0][0:100,520:620]
plt.imshow(cv2.cvtColor(cropped_image, cv2.COLOR_BGR2RGB))

cv2.imwrite("Cropped Image.jpg", cropped_image)
```

Out[7]:
```
True
```

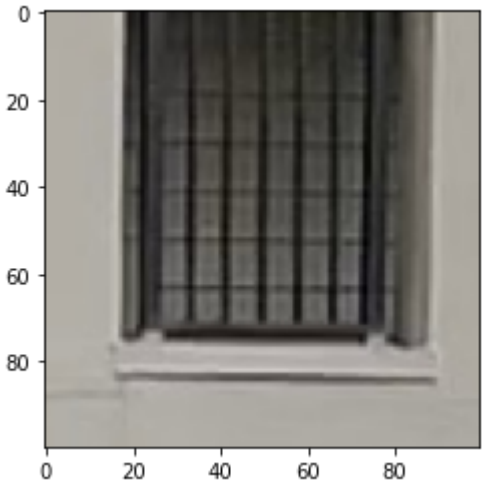


In [8]:
```
d=dict()
d_norm=dict()
for i in range(0,620,20):
    for j in range(0,1180,20):
        d[str(i)+":"+str(i+100),str(j)+":"+str(j+100)]=ssd(cropped_image,images[12][i:i
        #d_norm[str(i)+":"+str(i+100),str(j)+":"+str(j+100)]=ncc(norm_data(cropped_imag
```

In [9]:
```
a=min(d.items(), key=lambda x: x[1])
y1,y2=map(int,a[0][0].split(':'))
x1,x2=map(int,a[0][1].split(':'))
```

In [10]:
```
plt.imshow(cv2.cvtColor(images[12][y1:y2,x1:x2], cv2.COLOR_BGR2RGB))
```

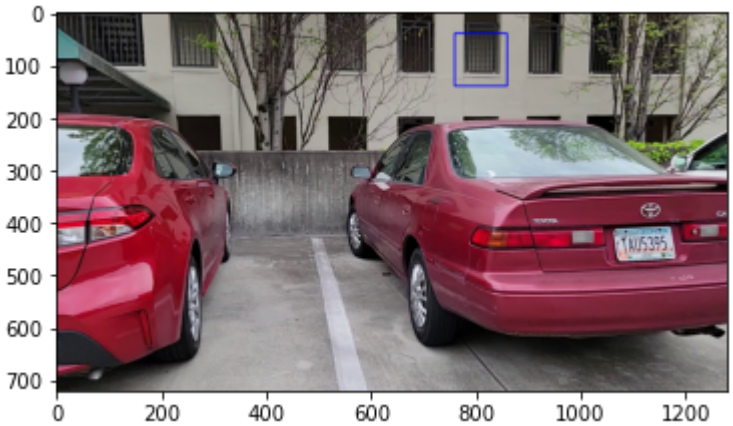
Out[10]:
```
<matplotlib.image.AxesImage at 0x237f8b59400>
```



In [11]:
```
color = (255, 0, 0)

# Line thickness of 2 px
thickness = 2

# Using cv2.rectangle() method
# Draw a rectangle with blue line borders of thickness of 2 px
image = cv2.rectangle(images[12], (x1,y1), (x2,y2), color, thickness)
```

In [17]:
```
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
```

Out[17]:
```
<matplotlib.image.AxesImage at 0x237f8b0b460>
```



Motion tracking equation

In [96]:
```
Iref=cv2.imread('data/frame0.jpg',cv2.IMREAD_GRAYSCALE)
Inext=cv2.imread('data/frame30.jpg',cv2.IMREAD_GRAYSCALE)
Iref=np.array(Iref).astype(np.float32)
Inext=np.array(Inext).astype(np.float32)
kernel_x = np.array([[-1., 1.], [-1., 1.]])*.25
kernel_y = np.array([[-1., -1.], [1., 1.]])*.25
```

```
kernel_t = np.array([[1., 1.], [1., 1.]])*.25
Iref = Iref / 255. # normalize pixels
Inext = Inext / 255. # normalize pixels
Ix=cv2.filter2D(Iref,-1,kernel=kernel_x)
Iy=cv2.filter2D(Iref,-1,kernel=kernel_y)
It=cv2.filter2D(Iref,-1,kernel=kernel_t)+cv2.filter2D(Inext,-1,kernel=kernel_x)
Ix,Iy,It=np.array(Ix),np.array(Iy),np.array(It)
```

In [103…

```
u=np.divide(It,np.sqrt(np.square(Ix)+np.square(Iy)))
```

```
C:\Users\anant\AppData\Local\Temp/ipykernel_15740/2072950217.py:1: RuntimeWarning: divid
e by zero encountered in true_divide
  u=np.divide(It,np.sqrt(np.square(Ix)+np.square(Iy)))
C:\Users\anant\AppData\Local\Temp/ipykernel_15740/2072950217.py:1: RuntimeWarning: inval
id value encountered in true_divide
  u=np.divide(It,np.sqrt(np.square(Ix)+np.square(Iy)))
```

In [104…

```
u
```

Out[104…

```
array([[224.52206  , 223.88962  , 223.8893   , ...,   1.5135136,
          1.6142869,   1.6235628],
       [224.52173  , 223.88928  , 223.88962  , ...,   1.5135134,
          1.6142869,   1.6235628],
       [158.53748  , 158.09027  , 139.04636  , ...,   2.8125768,
          2.7348177,   4.3707867],
       ...,
       [ 42.29572  ,  42.29572  ,  90.0854   , ...,  25.744085 ,
         17.03911  ,  54.465305 ],
       [ 31.839685 ,  32.141964 ,  47.82018  , ...,  35.329903 ,
         23.784195 ,  14.581035 ],
       [ 27.80274  ,  28.153563 ,  70.99516  , ...,  41.042233 ,
         39.726547 ,  25.000002 ]], dtype=float32)
```

Dense optical Flow

In [1]:

```
import cv2 as cv
import numpy as np


# The video feed is read in as
# a VideoCapture object
cap = cv.VideoCapture("20220417_151055.mp4")
ret, first_frame = cap.read()
prev_gray = cv.cvtColor(first_frame, cv.COLOR_BGR2GRAY)
mask = np.zeros_like(first_frame)
mask[..., 1] = 255

while(cap.isOpened()):

    ret, frame = cap.read()

    # Opens a new window and displays the input
    # frame
    cv.imshow("input", frame)

    # Converts each frame to grayscale - we previously
```

```
    # only converted the first frame to grayscale
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    # Calculates dense optical flow by Farneback method
    flow = cv.calcOpticalFlowFarneback(prev_gray, gray,None,0.5, 3, 15, 3, 5, 1.2, 0)

    # Computes the magnitude and angle of the 2D vectors
    magnitude, angle = cv.cartToPolar(flow[..., 0], flow[..., 1])
    # Sets image hue according to the optical flow
    # direction
    mask[..., 0] = angle * 180 / np.pi / 2

    # Sets image value according to the optical flow
    # magnitude (normalized)
    mask[..., 2] = cv.normalize(magnitude, None, 0, 255, cv.NORM_MINMAX)

    # Converts HSV to RGB (BGR) color representation
    rgb = cv.cvtColor(mask, cv.COLOR_HSV2BGR)

    # Opens a new window and displays the output frame
    cv.imshow("dense optical flow", rgb)

    # Updates previous frame
    prev_gray = gray

    # Frames are read by intervals of 1 millisecond. The
    # programs breaks out of the while loop when the
    # user presses the 'q' key
    if cv.waitKey(1) & 0xFF == ord('q'):
        break

# The following frees up resources and
# closes all windows
cap.release()
cv.destroyAllWindows()
```

LUCAS KANADE Algorithm

In [4]:

```
#'20220417_151055.mp4'
cap = cv2.VideoCapture(0)
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )
# Parameters for lucas kanade optical flow
lk_params = dict( winSize  = (15, 15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03
# Create some random colors
color = np.random.randint(0, 255, (100, 3))
# Take first frame and find corners in it
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)
# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)
while(1):
    ret, frame = cap.read()
```

```
    if not ret:
        print('No frames grabbed!')
        break
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # calculate optical flow
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)
    # Select good points
    if p1 is not None:
        good_new = p1[st==1]
        good_old = p0[st==1]
    # draw the tracks
    for i, (new, old) in enumerate(zip(good_new, good_old)):
        a, b = new.ravel()
        c, d = old.ravel()
        mask = cv2.line(mask, (int(a), int(b)), (int(c), int(d)), color[i].tolist(), 2)
        frame = cv2.circle(frame, (int(a), int(b)), 5, color[i].tolist(), -1)
    img = cv2.add(frame, mask)
    cv2.imshow('frame', img)
    if cv2.waitKey(1)==ord('q'):
        break
    # Now update the previous frame and previous points
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1, 1, 2)
cv2.destroyAllWindows()
```

Disparity based depth estimation

In [67]:

```
ul,vl=721,108 # from matlab ginput()
ur,vr=20,110
b=546.1 # distance between left and right cameras
f=1403.54736624058 #focallength
z=(b*f)/(ul-ur) #distance of object
print('The distance is '+str(z)+'mm')
```

```
The distance is 1093.4054446561781mm
```

Multiple Face Tracking with lip tracking using HAARCASCADE

In [5]:

```
import os
import time

import imutils
detectorPaths = {

    "face": "face.xml",
    "smile": "smile.xml",
}

print("[INFO] loading haar cascades...")
detectors = dict()

for (name, path) in detectorPaths.items():
    detectors[name] = cv2.CascadeClassifier(path)

print("[INFO] starting video stream...")
vs = cv2.VideoCapture(0)
```

```
while True:
    _,frame = vs.read()
    frame = imutils.resize(frame, width=500)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faceRects = detectors["face"].detectMultiScale(
        gray, scaleFactor=1.05, minNeighbors=5, minSize=(30, 30),
        flags=cv2.CASCADE_SCALE_IMAGE)

    for (fX, fY, fW, fH) in faceRects:
        faceROI = gray[fY:fY + fH, fX:fX + fW]
        smileRects = detectors["smile"].detectMultiScale(
            faceROI, scaleFactor=1.1, minNeighbors=10,
            minSize=(15, 15), flags=cv2.CASCADE_SCALE_IMAGE)
        for (sX, sY, sW, sH) in smileRects:
            ptA = (fX + sX, fY + sY)
            ptB = (fX + sX + sW, fY + sY + sH)
            cv2.rectangle(frame, ptA, ptB, (255, 0, 0), 2)
        cv2.rectangle(frame, (fX, fY), (fX + fW, fY + fH),
                      (0, 255, 0), 2)
    cv2.imshow("Frame", frame)
    if cv2.waitKey(1) == ord("q"):
        break

cv2.destroyAllWindows()
```

```
[INFO] loading haar cascades...
[INFO] starting video stream...
```

In [ ]:

Constraint eqn of optical flow :

- First we consider the 2 images and then we take that time t, and $t+\delta t$
- By considering 2 images and one small window at the same point in Both

i.e we get

α, ψ along with the $(x+\delta x, y+\delta y)$

the optical flow $u, v = \left(\frac{\delta x}{\delta t}, \frac{\delta y}{\delta t}\right)$ and the displacement is given as $(\delta x, \delta y)$

We are assuming that the Brightness

$$I(x+\delta x, y+\delta y, t+\delta t) = I(x, y, t) \longrightarrow \text{eqn } ①$$

We also see that the displacements are very small

We apply the taylor's approximation.

as δx is small

$$f(x+\delta x) = f(x) + \frac{\partial f}{\partial x}\delta x + \textcircled{0} \longrightarrow \text{Small order}$$

then

$$f(x+\delta x, y+\delta y, t+\delta t) \cong f(x, y, t) + \frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y + \frac{\partial f}{\partial t}\delta t$$

$$I(x+\delta x, y+\delta y, t+\delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t$$

We get $= I(x, y, t) + I_x \delta x + I_y \delta y + I_t \delta t$

$\longrightarrow$ eqn ②

Subtracting ① − ②

$$I_x \delta x + I_y \delta y + I_t \delta t = 0$$

divide by δt

$$\Rightarrow I_x \frac{\partial x}{\partial t} + I_y \frac{\partial y}{\partial t} + I_t = 0$$

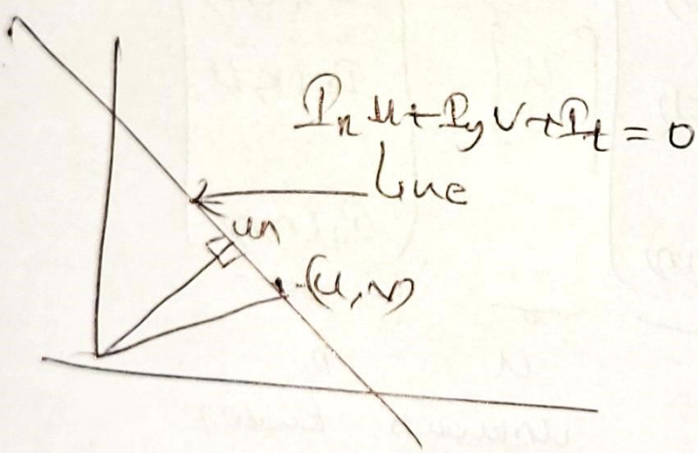
$$\boxed{I_x u + I_y v + I_t = 0}$$

Constraint eqn

u, v is optical flow

$I_x I_y I_t$ can be calculated from 2 frame

u, v lie in the line



Lucas Kanade algorithm :

Here we assume motion field and optical flow (u, v) is constant within a small neighbourhood ω.

- So for all $(K, l) \in \omega$

  derivatives in x, y, t direction $= 0$

$$I_x(K,l)u + I_y(K,l)v + I_t(K,l) = 0$$

Considering for $(K, I) \in \omega$

let window size be $n \times n$

In matrix form

$$\begin{bmatrix} I_x(1,1) & I_y(1,1) \\ I_x(K,l) & I_y(K,l) \\ I_x(n,n) & I_y(n,n) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} I_t(1,1) \\ I_t(K,l) \\ \vdots \\ I_t(n,n) \end{bmatrix}$$

A — known; u — unknown; B — known

considering $Au = B$

$$A^T A u = A^T B$$

matrix

$$\begin{bmatrix} \Sigma_\omega I_x I_x & \Sigma_\omega I_x I_y \\ \Sigma_\omega I_x I_y & \Sigma_\omega I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\Sigma_\omega I_x I_t \\ -\Sigma_\omega I_y I_t \end{bmatrix}$$

$$u = (A^T A)^{-1} A^T B$$

① $A^T A$ must be invertible $\rightarrow \det(A^T A) \neq 0$

② A^TA must be small - conditioned

λ_1 and λ_2 are the eigen values of A^TA

then $\lambda_1 > E$ and $\lambda_2 > E$

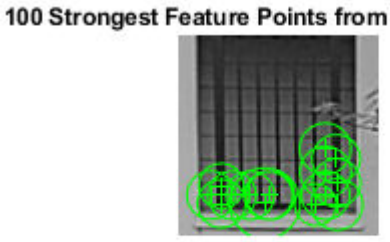
$\lambda_1 \geq \lambda_2$ but not $\lambda_1 \gg \lambda_2$

```
RGB = imread('Cropped.jpg');
boxImage = rgb2gray(RGB);
figure;
imshow(boxImage);
title('Image of a blanket');
```


```
RGB1 = imread('frame0.jpg');
sceneImage = rgb2gray(RGB1);
%sceneImage = imread(I1);
figure;
imshow(sceneImage);
title('Image of a Cluttered Scene');
```

```
boxPoints = detectSURFFeatures(boxImage);
```

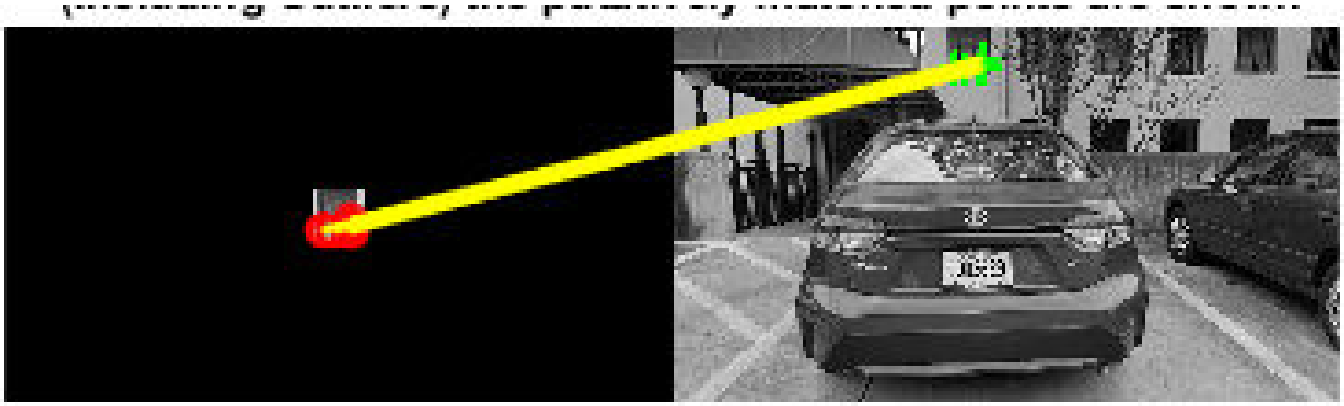
```
scenePoints = detectSURFFeatures(sceneImage);

figure;
imshow(boxImage);
title('100 Strongest Feature Points from Blanket image');
hold on;
plot(selectStrongest(boxPoints, 100));
```



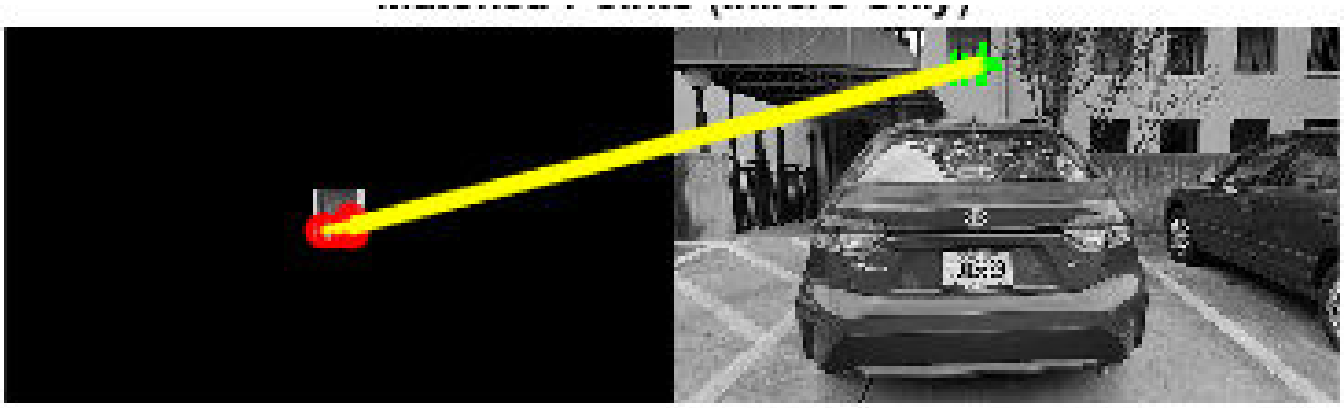

```
figure;
imshow(sceneImage);
title('300 Strongest Feature Points from blanket or clutter Image');
hold on;
plot(selectStrongest(scenePoints, 300));
```

```
[boxFeatures, boxPoints] = extractFeatures(boxImage, boxPoints);
```

```
[sceneFeatures, scenePoints] = extractFeatures(sceneImage, scenePoints);

boxPairs = matchFeatures(boxFeatures, sceneFeatures);

matchedBoxPoints = boxPoints(boxPairs(:, 1), :);
matchedScenePoints = scenePoints(boxPairs(:, 2), :);
figure;
showMatchedFeatures(boxImage, sceneImage, matchedBoxPoints, ...
    matchedScenePoints, 'montage');
title(' (Including Outliers) the putatively matched points are shown');
```



```
[tform, inlierIdx] = ...
    estimateGeometricTransform2D(matchedBoxPoints, matchedScenePoints, 'affine');
inlierBoxPoints   = matchedBoxPoints(inlierIdx, :);
inlierScenePoints = matchedScenePoints(inlierIdx, :);

figure;
showMatchedFeatures(boxImage, sceneImage, inlierBoxPoints, ...
    inlierScenePoints, 'montage');
title('Matched Points (Inliers Only)');
```



```
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

newBoxPolygon = transformPointsForward(tform, boxPolygon);

figure;
imshow(sceneImage);
hold on;
line(newBoxPolygon(:, 1), newBoxPolygon(:, 2), 'Color', 'y');
title('Detected blanket');
```

```
imds = imageDatastore('kitchen','IncludeSubfolders',true,'LabelSource','foldernames');
tbl = countEachLabel(imds)
```

tbl = 5×2 table

	Label	Count
1	Bottle	10
2	Fork	10
3	Peeler	10
4	Scissors	10
5	mug	10

```
figure
montage(imds.Files(1:6:end))
```

```
[trainingSet, validationSet] = splitEachLabel(imds, 0.6, 'randomize');
bag = bagOfFeatures(trainingSet);
```

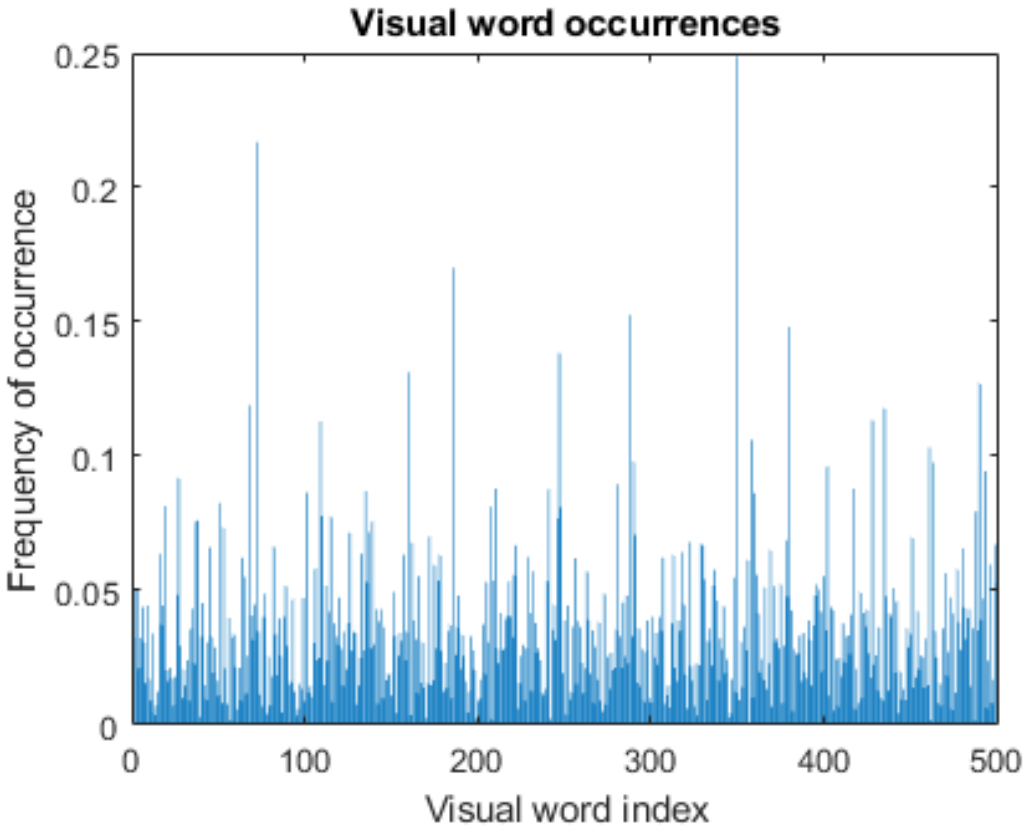
```
Creating Bag-Of-Features.
-------------------------
* Image category 1: Bottle
* Image category 2: Fork
* Image category 3: Peeler
* Image category 4: Scissors
* Image category 5: mug
* Selecting feature point locations using the Grid method.
* Extracting SURF features from the selected feature point locations.
** The GridStep is [8 8] and the BlockWidth is [32 64 96 128].

* Extracting features from 30 images...done. Extracted 5898240 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          : 4718590
* Number of clusters          : 500
* Initializing cluster centers...100.00%.
* Clustering...completed 24/100 iterations (~8.30 seconds/iteration)...converged in 24 iterations.

* Finished creating Bag-Of-Features
```

```
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')
```




```
categoryClassifier = trainImageCategoryClassifier(trainingSet, bag);
```

```
Training an image category classifier for 5 categories.
--------------------------------------------------------
* Category 1: Bottle
* Category 2: Fork
* Category 3: Peeler
* Category 4: Scissors
* Category 5: mug

* Encoding features for 30 images...done.

* Finished training the category classifier. Use evaluate to test the classifier on a test set.
```

```
confMatrix = evaluate(categoryClassifier, trainingSet);
```

```
Evaluating image category classifier for 5 categories.
--------------------------------------------------------

* Category 1: Bottle
* Category 2: Fork
* Category 3: Peeler
* Category 4: Scissors
* Category 5: mug

* Evaluating 30 images...done.

* Finished evaluating all the test sets.

* The confusion matrix for this test set is:


                           PREDICTED
KNOWN       | Bottle   Fork   Peeler   Scissors   mug
---------------------------------------------------------
Bottle      | 1.00     0.00   0.00     0.00       0.00
Fork        | 0.00     1.00   0.00     0.00       0.00
Peeler      | 0.00     0.00   1.00     0.00       0.00
Scissors    | 0.00     0.00   0.00     1.00       0.00
mug         | 0.00     0.00   0.00     0.00       1.00

* Average Accuracy is 1.00.
```

```
confMatrix = evaluate(categoryClassifier, validationSet);
```

```
Evaluating image category classifier for 5 categories.
--------------------------------------------------------

* Category 1: Bottle
* Category 2: Fork
* Category 3: Peeler
* Category 4: Scissors
* Category 5: mug

* Evaluating 20 images...done.

* Finished evaluating all the test sets.

* The confusion matrix for this test set is:


                           PREDICTED
KNOWN       | Bottle   Fork   Peeler   Scissors   mug
---------------------------------------------------------
Bottle      | 1.00     0.00   0.00     0.00       0.00
```

```
Fork        | 0.00     1.00   0.00     0.00       0.00
Peeler      | 0.00     0.00   1.00     0.00       0.00
Scissors    | 0.00     0.00   0.00     1.00       0.00
mug         | 0.00     0.00   0.00     0.00       1.00

* Average Accuracy is 1.00.
```

```
% Compute average accuracy
mean(diag(confMatrix))
```

```
ans = 1
```

```
img = imread(fullfile('kitchen','Scissors','Scissors.jpeg'));
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 = 1×1 cell array
    {'Scissors'}
```

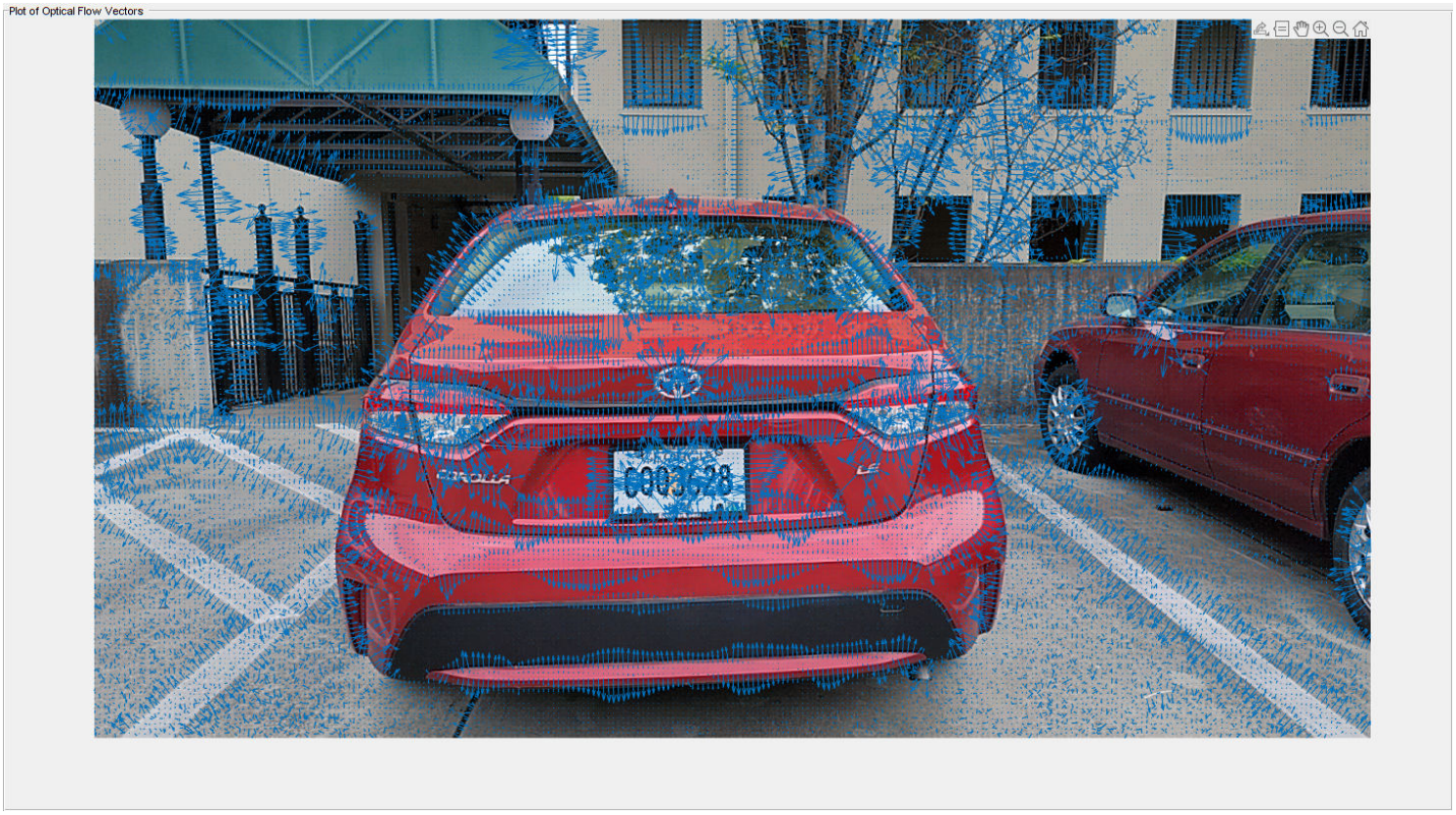
```
setDir  = fullfile('kitchen');
imgSets = imageSet(setDir,'recursive');

trainingSets = partition(imgSets,2);

bag = bagOfFeatures(trainingSets,'Verbose',false);

img = read(imgSets(1),1);
featureVector = encode(bag,img);
```

```
Encoding images using Bag-Of-Features.
-------------------------------------
* Encoding an image...done.
```

```
setDir  = fullfile('kitchen');
imds = imageDatastore(setDir,'IncludeSubfolders',true,'LabelSource',...
    'foldernames');

extractor = @exampleBagOfFeaturesExtractor;
bag = bagOfFeatures(imds,'CustomExtractor',extractor)
```

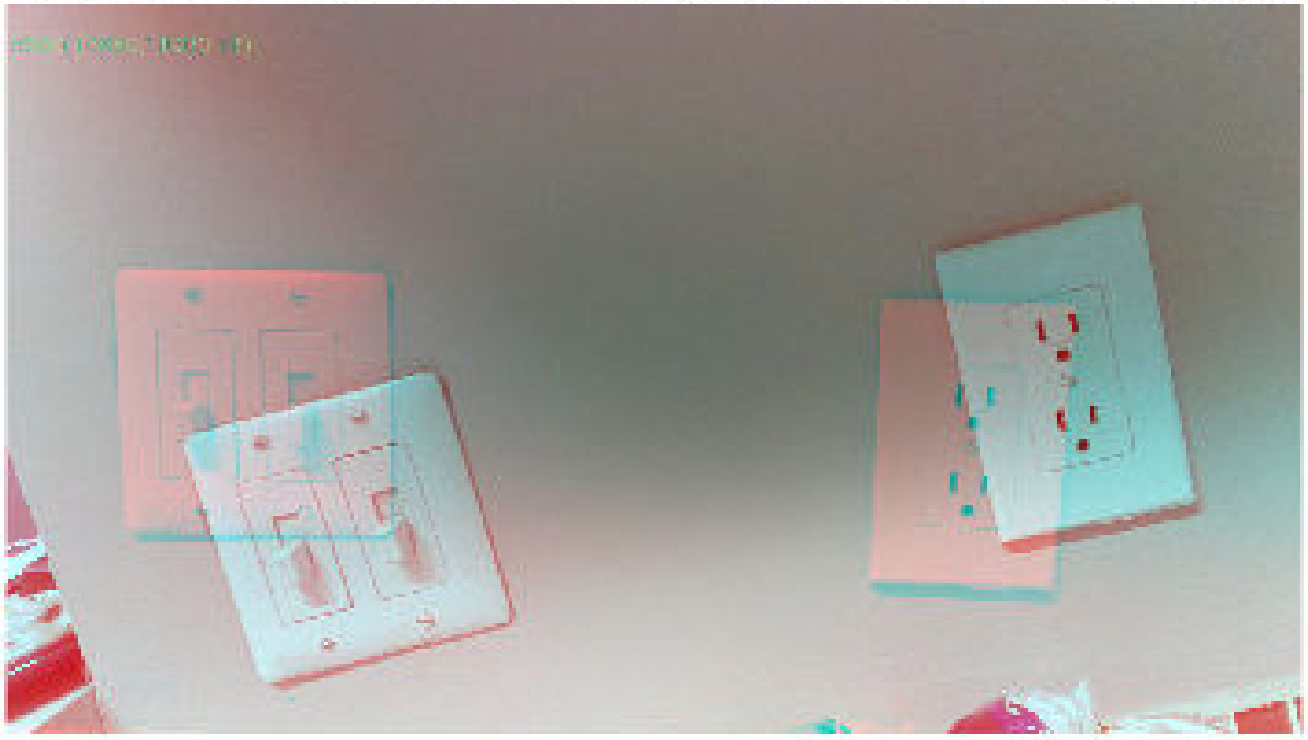
```
Creating Bag-Of-Features.
-------------------------
* Image category 1: Bottle
* Image category 2: Fork
* Image category 3: Peeler
* Image category 4: Scissors
* Image category 5: mug
* Extracting features using a custom feature extraction function: exampleBagOfFeaturesExtractor.

* Extracting features from 50 images...done. Extracted 9830400 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          : 7864320
* Number of clusters          : 500
* Initializing cluster centers...100.00%.
* Clustering...completed 20/100 iterations (~13.39 seconds/iteration)...converged in 20 iterations.

* Finished creating Bag-Of-Features
bag =
  bagOfFeatures with properties:

      CustomExtractor: @exampleBagOfFeaturesExtractor
       NumVisualWords: 500
       TreeProperties: [1 500]
    StrongestFeatures: 0.8000
```

```
vidReader = VideoReader('video.mp4');
opticFlow = opticalFlowHS;
h = figure;
movegui(h);
hViewPanel = uipanel(h,'Position',[0 0 1 1],'Title','Plot of Optical Flow Vectors');
hPlot = axes(hViewPanel);
c=0;
n=1;
while hasFrame(vidReader)
        frameRGB = readFrame(vidReader);
        frameGray = im2gray(frameRGB);
    if mod(c,n)==0
        flow = estimateFlow(opticFlow,frameGray);
        imshow(frameRGB)
        hold on
        plot(flow,'DecimationFactor',[5 5],'ScaleFactor',60,'Parent',hPlot);
        hold off
        pause(10^-3)
    else
        imshow(frameRGB)
    end


    c=c+1;
end
```

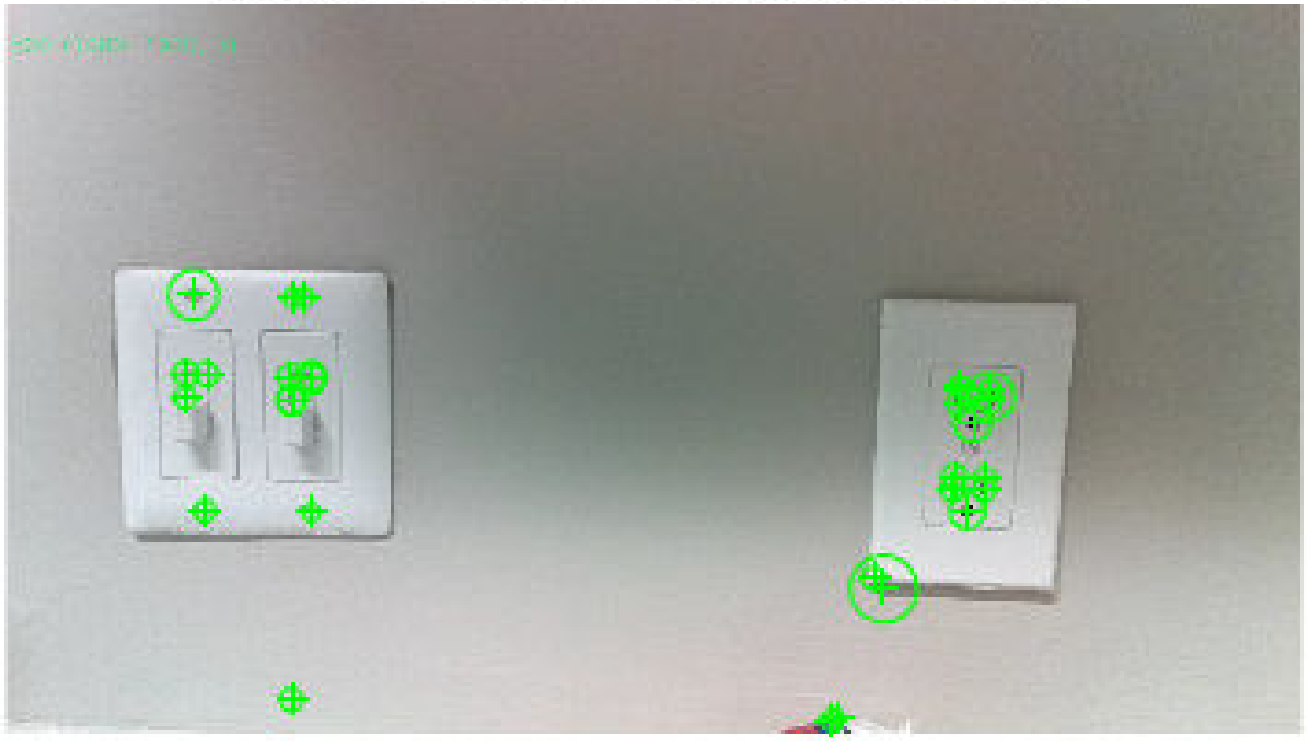


```
I1 = imread('image1.jpg');
I2 = imread('image2.jpg');

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

figure;
imshowpair(I1, I2,'montage');
title('I1 (left); I2 (right)');
```

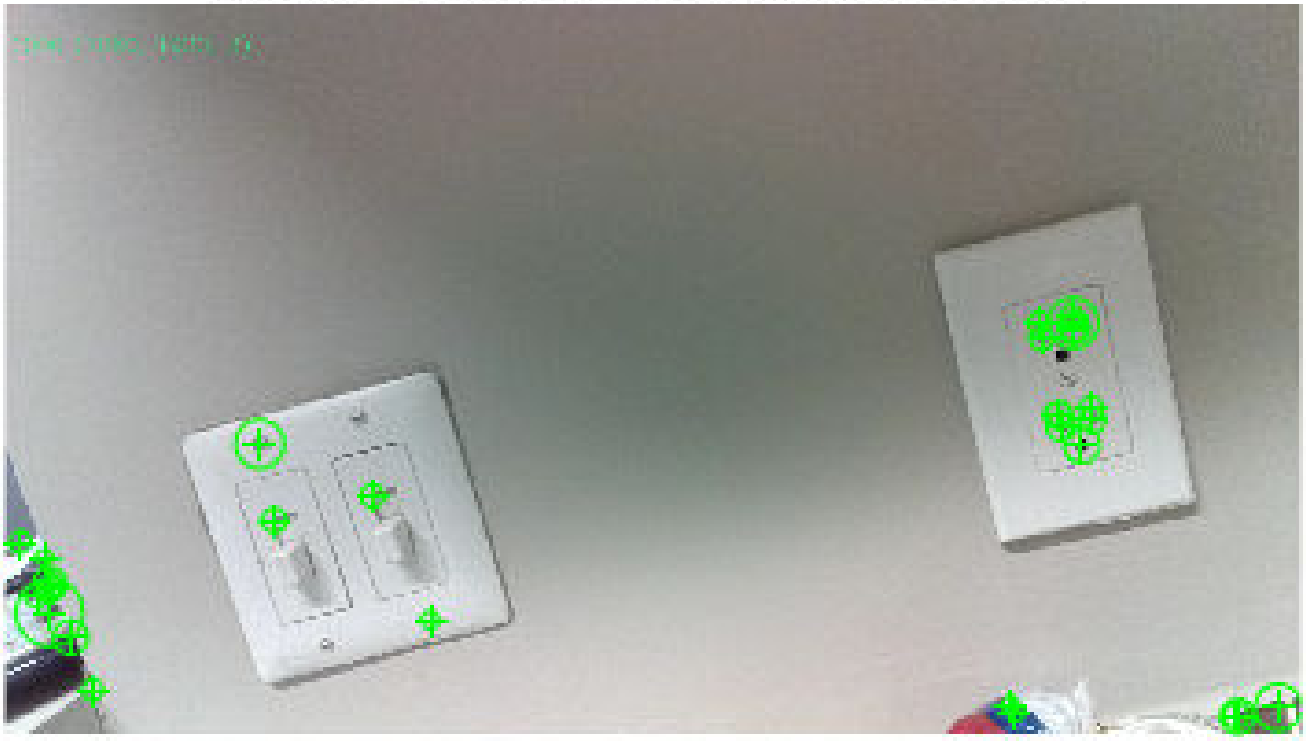
```
figure;
imshow(stereoAnaglyph(I1,I2));
title('Composite Image (Red - Left Image, Cyan - Right Image)');
```



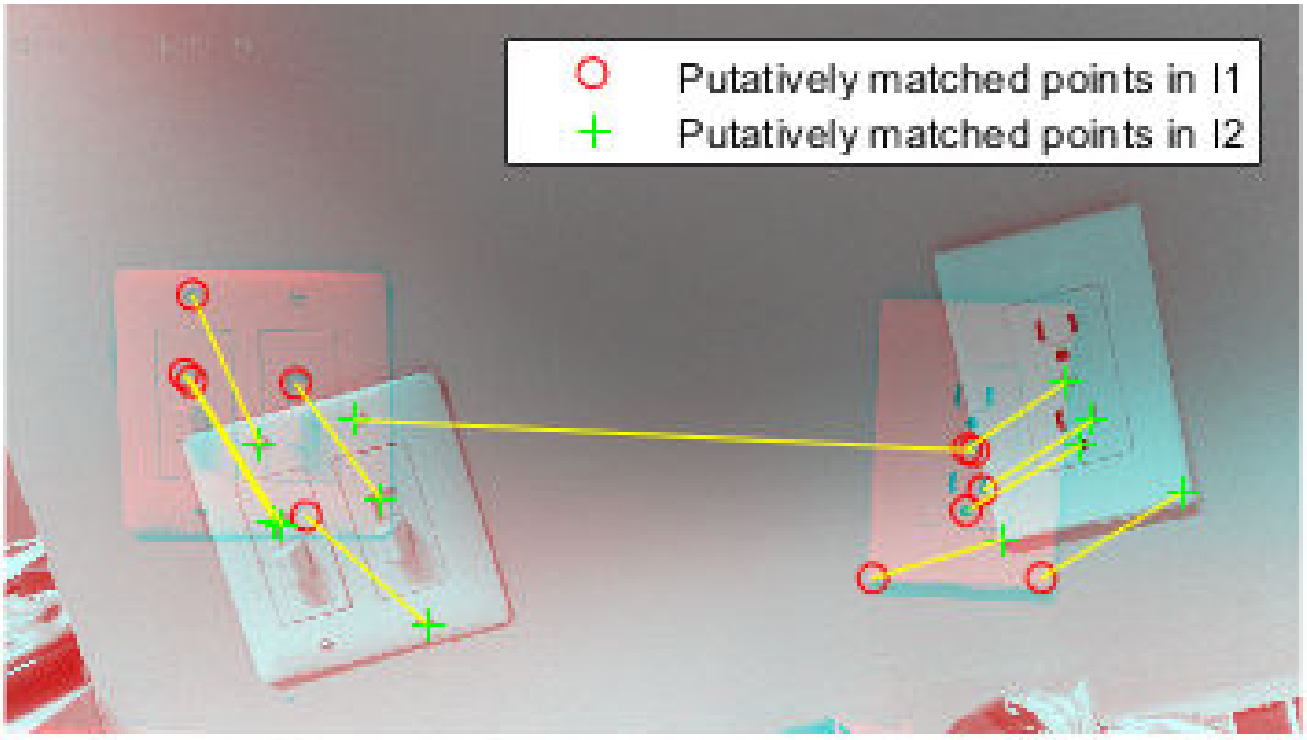
```
blobs1 = detectSURFFeatures(I1gray, 'MetricThreshold', 2000);
blobs2 = detectSURFFeatures(I2gray, 'MetricThreshold', 2000);

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



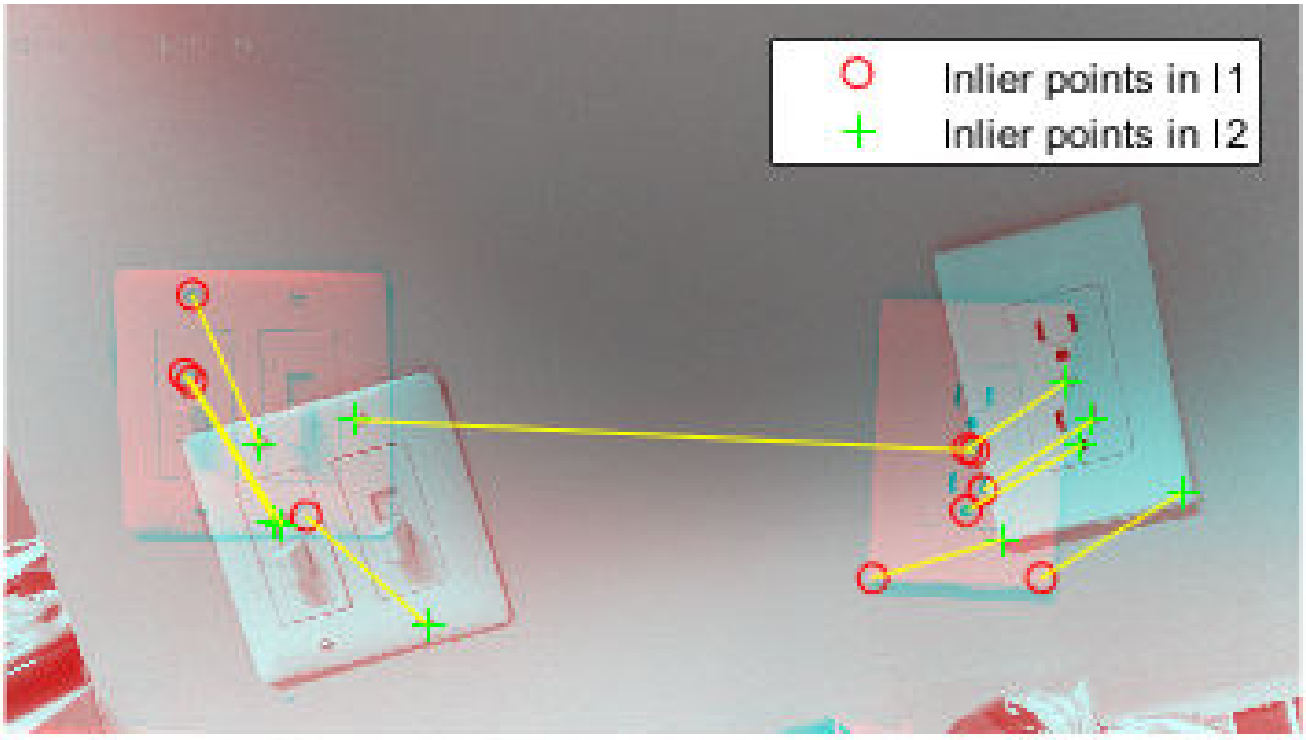
```
figure;
imshow(I2);
hold on;
plot(selectStrongest(blobs2, 30));
title('Thirty strongest SURF features in I2');
```



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

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

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

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




```
[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');
```




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

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

I Stereo Images (Red - Left Image, Cyan - Righ

```
%RectifyImages('rgb_image1.jpg', 'rgb_image16.jpg');
```