CS 5670 Computer Vision

Spring 2015

HW 2 Writeup

Tara Wilson (taw84)

Jai Chaudhary (jc2855)

SIFT

Assuming your goal is to find interest points that are consistent under changes in the image, or noise, which of the operators will you choose?

The DOG operator is more robust to noise as the gaussian kernel convolution blurs out the high frequency noise.
DoG acts as a band-pass filter, which removes high frequency components representing noise, and also some low frequency components representing the homogeneous areas in the image(illumination sensitive component). The gradient magnitude takes care of the homogeneous components but is highly sensitive to high frequency noise.

Use SIFT to find interest points and descriptors to all the provided input images.

We used, OpenCV built with its python bindings. We tried compiling the latest release of opencv for Mac, but were unsuccesful. We instead used the brew binary with version 2.4.11.

```
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('StopSign1.jpg',0)

sift = cv2.SIFT()
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
kp, des = sift.detectAndCompute(gray,None)

img=cv2.drawKeypoints(gray,kp,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imshow('sift_keypoints',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Write a function "findMatches" that gets as input two sets of descriptors and their locations in the image (one set per image). The function should return a list or an array of corresponding descriptors (their coordinates). The matching should be based on similarity between the descriptors.

Similarity measure is euclidean distance between feature descriptors

```
class SIFT:
    MIN_MATCH_COUNT = 10
    extractor = cv2.SIFT()


    """
    returns list of DMatch Objects (matches) sorted by distance
    that contains keypoints matched in images.

    :param keypoints1: list of keypoint objects for image 1
    :param descriptorSet1:  list of descriptor objects for image 1
    :param keypoints2:  list of keypoint objects for image 2
    :param descriptorSet2:  list of descriptor objects for image 2
    :returns: list of DMatch Objects
    """
    def findMatches(self,keypoints1, descriptorSet1, keypoints2, descriptorSet2):
        # create BFMatcher object
        bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

        # Match descriptors.
        matches = bf.match(descriptorSet1,descriptorSet2)

        # Sort them in the order of their distance.
        matches = sorted(matches, key = lambda x:x.distance)

        # visualize the matches
        print '#matches:', len(matches)
        dist = [m.distance for m in matches]

        print 'distance: min: %.3f' % min(dist)
        print 'distance: mean: %.3f' % (sum(dist) / len(dist))
        print 'distance: max: %.3f' % max(dist)

        return matches
```

Write a function "showMatches" that gets as input two images, and the list (or array) of matches from the previous item. The function will generate an image where the matches are displayed. Please generate these images for the following image pairs: StopSign1-StopSign2, StopSign1-StopSign3, StopSign1-StopSign4

```
class SIFT:
   MIN_MATCH_COUNT = 10
   extractor = cv2.SIFT()

...

    """
     generate an image where the matches are visualizated by
     connecting matching pixels by highlighted lines

    :param imageFilename1: path to image1
    :param imageFilename2:  path to image2
    :returns: void
    """
    def showImages(self, imageFilename1, imageFilename2):


        # find the keypoints and descriptors with SIFT
        img1 = cv2.cvtColor(cv2.imread(imageFilename1), cv2.COLOR_BGR2GRAY)
        img2 = cv2.cvtColor(cv2.imread(imageFilename2), cv2.COLOR_BGR2GRAY)

        kp1, des1 = self.extractor.detectAndCompute(img1,None)
        kp2, des2 = self.extractor.detectAndCompute(img2,None)
        matches = self.findMatches(kp1, des1, kp2, des2)
        self.drawMatches(img1, kp1, img2, kp2, matches)

    """
     Visualization
     My own implementation of cv2.drawMatches as brew binary for MacOSX
     OpenCV 2.4.11 does not suppourt this function  but it's supported in
     OpenCV 3.0.0

    :param img1: image1 Object
    :param kp1: list of keypoint objects for image 1
    :param img2: image2 Object
    :param kp2: list of keypoint objects for image 2
    :param matches: list of DMatch Objects that contains keypoints matched in images.
```

```
    :returns: void
    """
    def drawMatches(self, img1, kp1, img2, kp2, matches):


        h1, w1 = img1.shape[:2]
        h2, w2 = img2.shape[:2]
        view = sp.zeros((max(h1, h2), w1 + w2, 3), sp.uint8)
        view[:h1, :w1, 0] = img1
        view[:h2, w1:, 0] = img2
        view[:, :, 1] = view[:, :, 0]
        view[:, :, 2] = view[:, :, 0]


        for m in matches:

            # Get the matching keypoints for each of the images
            img1_idx = m.queryIdx
            img2_idx = m.trainIdx

            # x - columns
            # y - rows
            (x1,y1) = kp1[img1_idx].pt
            (x2,y2) = kp2[img2_idx].pt
            color = tuple([sp.random.randint(0, 255) for _ in xrange(3)])
            cv2.line(view, (int(x1), int(y1)) , ((int(x2) + w1), int(y2)), color)
            cv2.circle(view, (int(x1),int(y1)), 4, (255, 0, 0), 1)
            cv2.circle(view, (int(x2)+w1,int(y2)), 4, (255, 0, 0), 1)

        cv2.imshow("Draw Matches", view)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
```

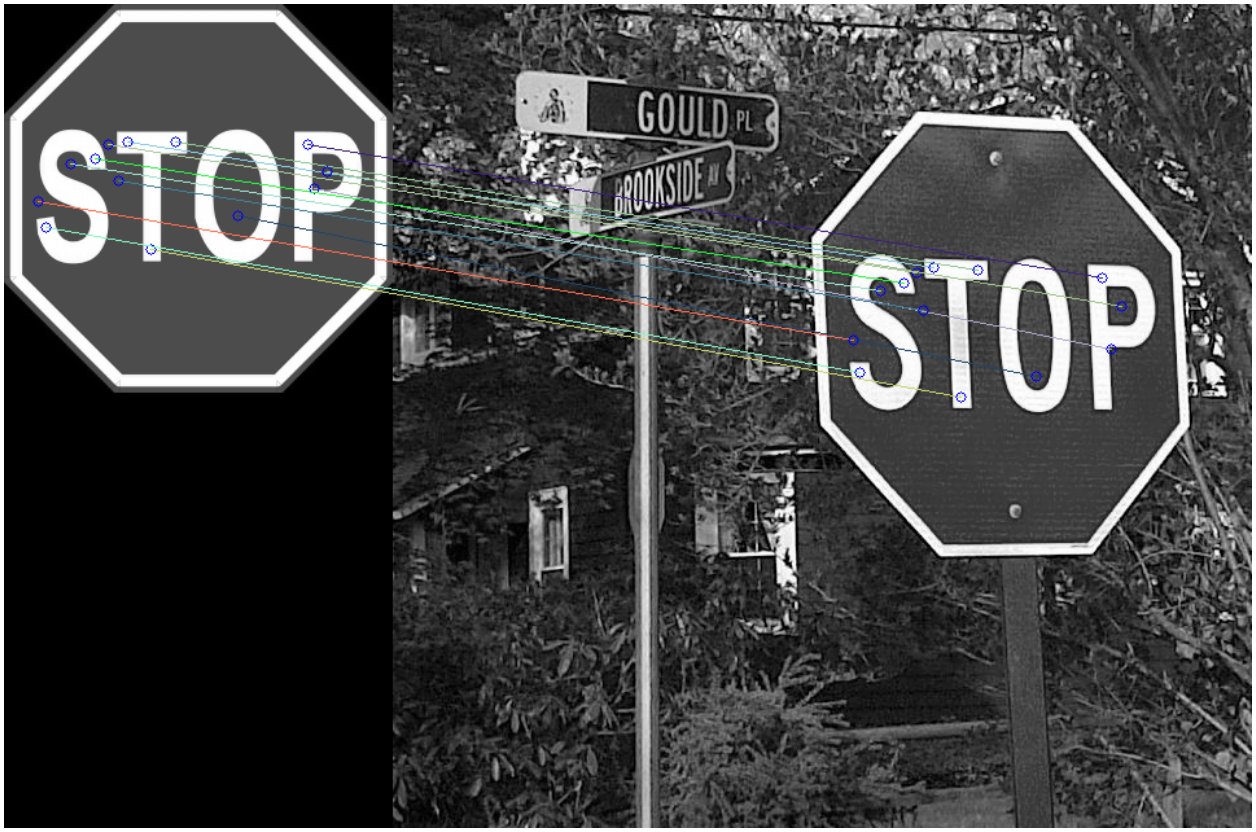

STOP
GOULD PL
BROOKSIDE AV
STOP

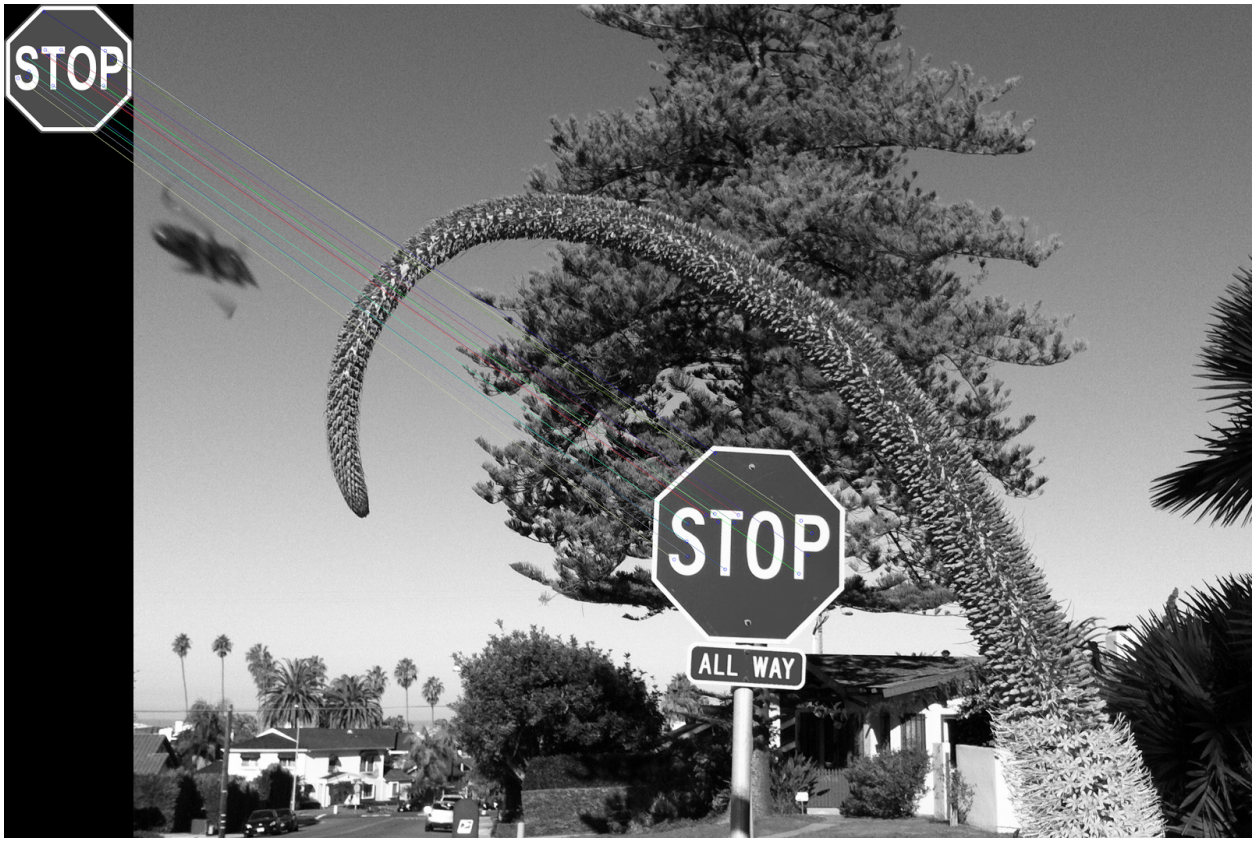

STOP
STOP
ALL WAY


STOP
STOP

Write a function "affineMatches" that takes a list of matches between two images, uses RANSAC to find an affine transformation between the pair of images and rejects the outlier matches. You can implement RANSAC yourself, or download code for it. The function should return the computed affine transformation and the inlier matches. Generate again the visualizations of the image matches, this time using only inliers. Did RANSAC do what you expected it to do? How many matches did you have before and after RANSAC? Are all the remaining matches consistent geometrically?

```
from skimage.measure import  ransac
from skimage.transform import warp, AffineTransform

class SIFT:
   MIN_MATCH_COUNT = 10
   extractor = cv2.SIFT()

....

   def affineMatches(self, matches, kp1, kp2):
      src_pts = np.float32([ kp1[m.queryIdx].pt for m in matches ])
      dst_pts = np.float32([ kp2[m.trainIdx].pt for m in matches ])
      model = AffineTransform()
      model.estimate(src_pts, dst_pts)

      model_robust, inliers = ransac((src_pts, dst_pts), AffineTransform, min_samples=3,
                        residual_threshold=10, max_trials=500)

      print(model.scale, model.translation, model.rotation)
      print(model_robust.scale, model_robust.translation, model_robust.rotation)

      return inliers, model_robust
```


STOP
GOULD PL
BROOKSIDE AV
STOP

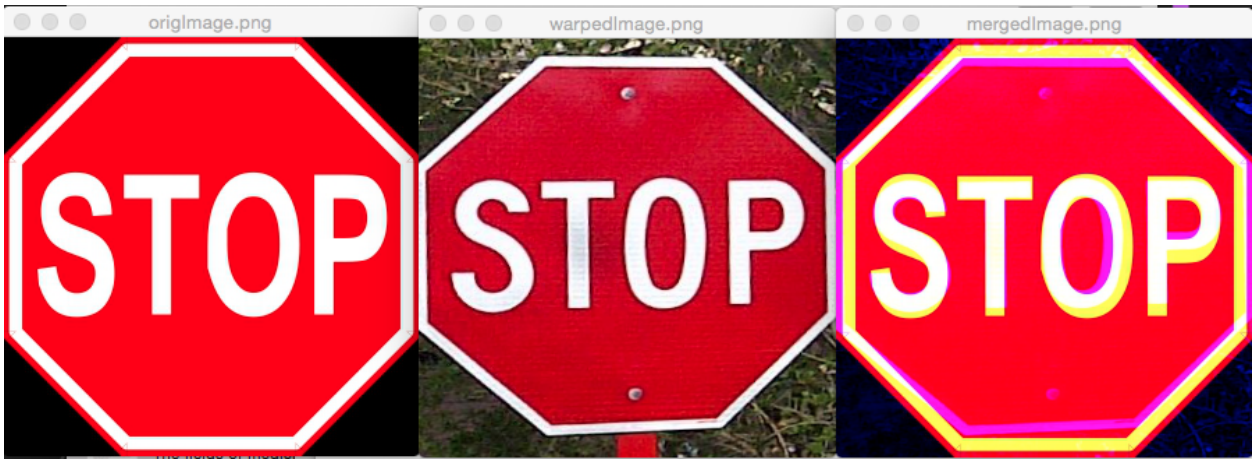

STOP
STOP
ALL WAY

Did RANSAC do what you expected it to do? How many matches did you have before and after RANSAC? Are all the remaining matches consistent geometrically?

Yeah RANSAC as expected took care of the outliers and helped us identify a more robust match. Before RANSAC we had 59 matches , however after RANSAC we had 18 features. Yes all the remaining matches are geometrically consistent

You will write a function “alignImages” that gets as input a pair of images and the affine transformation between them. The function will apply the transformation to one of the images (warp) and will “merge” the warped image with the other image. Merging will be done as follows: Take the red and green channels from one image and the blue channel from the other image. Use them to generate a new image.

```
def alignImages(self, imageFilename1, imageFilename2, affineTransform):
    img1 = cv2.imread(imageFilename1)
    img2 = cv2.imread(imageFilename2)
    img_warped = warp(img2, affineTransform, output_shape=img1.shape)

    b1,g1,r1 = cv2.split(img1)

    b_warped,g_warped,r_warped = cv2.split(img_warped)
    img_merged = cv2.merge((b_warped, np.float64(g1), np.float64(r1)))
    cv2.imshow("origImage.png", img1)
    cv2.imshow("warpedImage.png", img_warped)
    cv2.imshow("mergedImage.png", img_merged)
```



How can you tell if the transformation you got is accurate? Did you get satisfactory results? When does it succeed and when does it fail?

The transformation is impressive but not completely accurate. The merging of slightly distorted blue channel causes the glitter effect. It succeeds when there exists a strict transformation between images, however if one part of the image scales differently than other (STOP vs Outer Hexagon in StopSign 1 and 2) then the affine transformåtion is imperfect.

Come up with a quantitative measure to assess the success of the alignment. The measure you propose and its results should make sense and explain well your results. Discuss the quantitative results you got.

We think the most appropriate quantitative measure to assess the success of alignment is Correlation of HISTOGRAM. As SIFT is robust towards changes in illumination, the measure should be as well. We calculate histogram for both images, and then compare them using correlation. We used Bhattacharya Distance as a measure of similarity between the two discrete probability distributions.

```
class SIFT:

…

    def alignmentError(self, img1, img2):
        hist1 = cv2.calcHist([img1.astype('float32')],[0],None,[256],[0,256])
        hist2 = cv2.calcHist([img2.astype('float32')],[0],None,[256],[0,256])
        return cv2.compareHist(hist1, hist2, cv2.cv.CV_COMP_BHATTACHARYYA)
```

It is following for original and warped images:

StopSign1.jpg & StopSign1.jpg	0.0
StopSign1.jpg & StopSign2.jpg	0.436
StopSign1.jpg & StopSign3.jpg	0.431
StopSign1.jpg & StopSign4.jpg	0.430

It makes sense as the StopSign1.jpg & StopSign2.jpg do not have a consistent affine transformation for the hexagon boundary and the STOP letters.

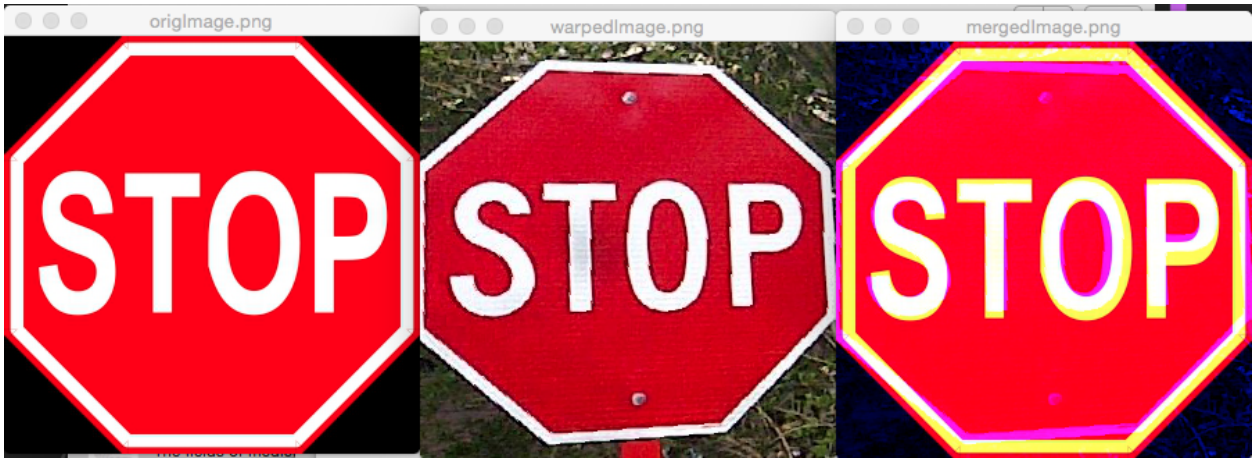
Repeat Items 5 and 6, this time for a homography. Compare the quantitative results of with affine transformations and homographies. Can you explain the results?

```
  class SIFT:

…
   def homographyMatches(self, matches, kp1, kp2):
      src_pts = np.float32([ kp1[m.queryIdx].pt for m in matches ]).reshape(-1,1,2)
      dst_pts = np.float32([ kp2[m.trainIdx].pt for m in matches ]).reshape(-1,1,2)

      transformMatrix, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,30.0)

      inliers = mask.ravel().tolist()

      inlier_idxs = np.nonzero(inliers)[0]

      homographyMatches = [matches[idx] for idx in inlier_idxs]

      print ("Homography Matches", len(homographyMatches))
      return homographyMatches, transformMatrix

   def alignHomography(self, imageFilename1, imageFilename2, homographyTransform):
      img1 = cv2.imread(imageFilename1)
      img2 = cv2.imread(imageFilename2)
      img_warped = cv2.warpPerspective(img2, homographyTransform, img1.shape[:2],
(1000,1000), flags = cv2.WARP_INVERSE_MAP)

      b1,g1,r1 = cv2.split(img1)

      b_warped,g_warped,r_warped = cv2.split(img_warped)
      img_merged = cv2.merge((np.float64(b_warped), np.float64(g1), np.float64(r1)))
      cv2.imshow("origImage.png", img1)
      cv2.imshow("warpedImage.png", img_warped)
      cv2.imshow("mergedImage.png", img_merged)
```


STOP
GOULD PL
BROOKSIDE AV
STOP


STOP
STOP
ALL WAY

The affine transformation had slightly less number of matches



How can you tell if the transformation you got is accurate? Did you get satisfactory results? When does it succeed and when does it fail?

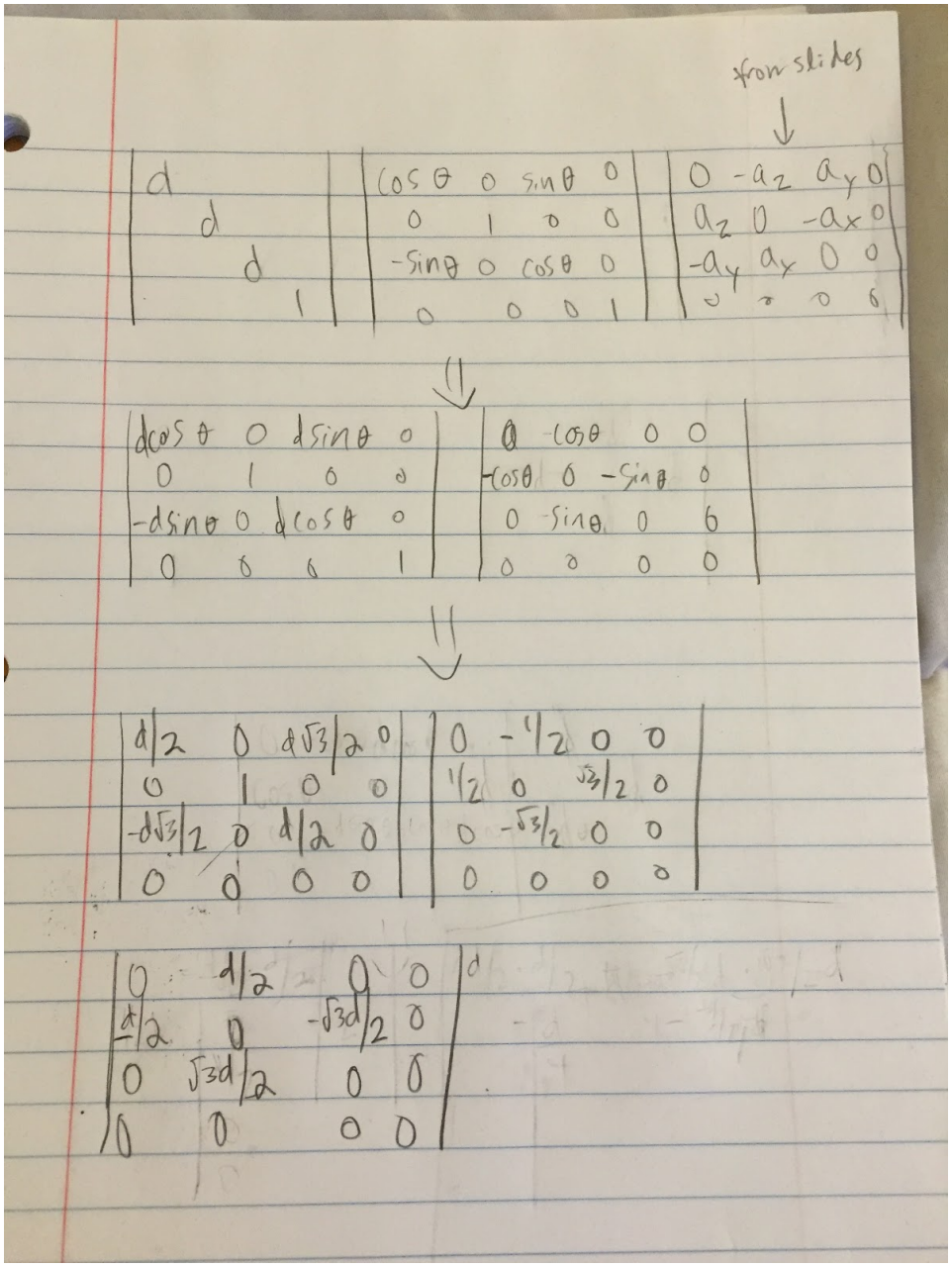
Although I must say I had higher expectations from Homography. The images are still distorted. Interestingly, It performs well even when there does not exist a strict transformation between images, (STOP vs Outer Hexagon) where the affine transformåtion is imperfect.

The quantitative results are following for original and warped images:

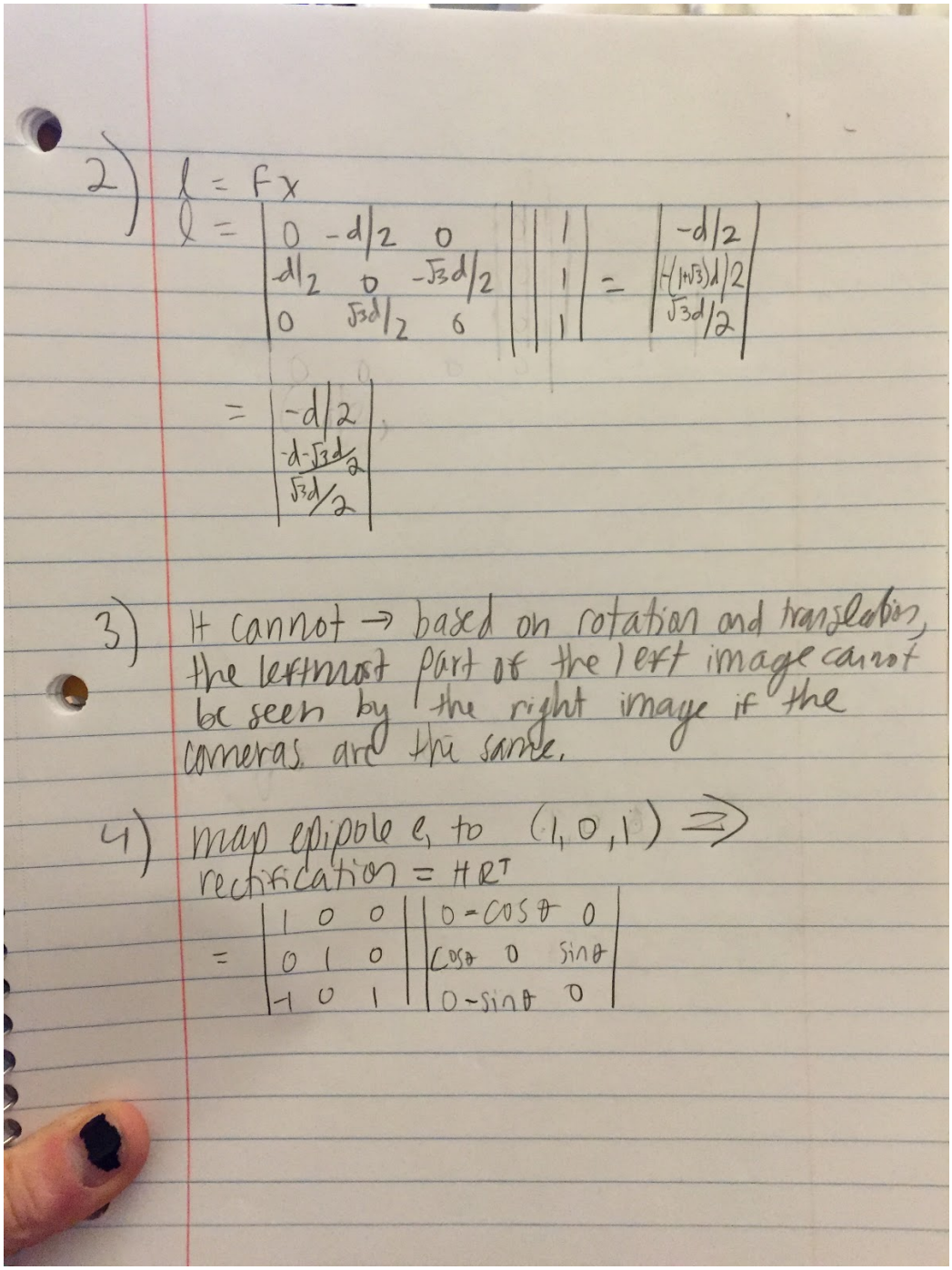
StopSign1.jpg & StopSign1.jpg	0.0
StopSign1.jpg & StopSign2.jpg	0.709
StopSign1.jpg & StopSign3.jpg	0.805
StopSign1.jpg & StopSign4.jpg	0.881

Epipolar Geometry and Camera Calibration Task One:

Problem One



Problem Two Through Four



Epipolar Geometry and Camera Calibration Task Two:

See Code.