CSC420 A4
Zhihong Wang
wangz154
1002095207

Q1. Stereo Matching Costs

(a). Imagine that two images for stereo matching are captured by the same camera but under different exposure times. Compare the two main cost functions for matching, i.e. SSD and NC, for estimating the correspondences in these two images. Which one performs better and why?

NC is better. SSD is the sum of squared differences. NC is the normalized correlation. SSD only compares the corresponding pixels, however, NC will normalize the pixels before computing the correlation. If there are two same patches, the only difference between them is the brightness. In this case, SSD will be strongly affected, since the brightness changes values of pixels. However, NC won’t be affected since the values of pixels are normalized.

Q2. Stereo Matching Implementation

Q2. (a). Write a program to compute the depth for each pixel in the given bounding box of car. Use the algorithm given in class, where given a left patch, compare it with all the patches on the right image's scanline. To reduce computation complexity, you can try to use a small patch size, or sample patches (e.g. every other pixel) from scanline instead of comparing with all possible patches. Report what is the patch size, sampling method, and matching cost function you used. Use the parameters given and show how depth is computed for each pixel. Also visualize the depth information. Are there any outliers coming from incorrect point correspondences?

Patch size = 7
Sampling method: Scanline (every patches from left to target)
Matching cost function: NC

Yes, there are outliers. Please check the visualization for the outliers.

Read files:

```
left_path = "./drive/My Drive/csc420/a4/000020_left.jpg"
right_path = "./drive/My Drive/csc420/a4/000020_right.jpg"
I1_path = "./drive/My Drive/csc420/a4/I1.jpg"
I2_path = "./drive/My Drive/csc420/a4/I2.jpg"
I3_path = "./drive/My Drive/csc420/a4/I3.jpg"

left_car_box_path = "./drive/My Drive/csc420/a4/000020.txt"
camera_features_path = "./drive/My Drive/csc420/a4/000020_allcalib.txt"

left_car_box = open(left_car_box_path, "r")
left_car_box_list = left_car_box.readline().split(" ")[1:]
left_car_top_left = (float(left_car_box_list[0]), float(left_car_box_list[1]))
left_car_bottom_right = (float(left_car_box_list[2]), float(left_car_box_list[3]))

camera_features = open(camera_features_path, "r")
camera_features_list = camera_features.readlines()
f = float(camera_features_list[0].split(" ")[1])
px = float(camera_features_list[1].split(" ")[1])
py = float(camera_features_list[2].split(" ")[1])
baseline = float(camera_features_list[3].split(" ")[1])
```

```
left_img = cv2.imread(left_path)
left_img_gray = cv2.cvtColor(left_img, cv2.COLOR_BGR2GRAY)
left_img_float = left_img.astype(np.float64)
left_img_gray_float = left_img_gray.astype(np.float64)

right_img = cv2.imread(right_path)
right_img_gray = cv2.cvtColor(right_img, cv2.COLOR_BGR2GRAY)
right_img_float = right_img.astype(np.float64)
right_img_gray_float = right_img_gray.astype(np.float64)
```

```
I1_img = cv2.imread(I1_path, cv2.IMREAD_COLOR)
I1_img = cv2.cvtColor(I1_img, cv2.COLOR_BGR2RGB)
I1_img_gray = cv2.cvtColor(I1_img, cv2.COLOR_BGR2GRAY)
I1_img_float = I1_img.astype(np.float64)
I1_img_gray_float = I1_img_gray.astype(np.float64)

I2_img = cv2.imread(I2_path, cv2.IMREAD_COLOR)
I2_img = cv2.cvtColor(I2_img, cv2.COLOR_BGR2RGB)
I2_img_gray = cv2.cvtColor(I2_img, cv2.COLOR_BGR2GRAY)
I2_img_float = I2_img.astype(np.float64)
I2_img_gray_float = I2_img_gray.astype(np.float64)

I3_img = cv2.imread(I3_path, cv2.IMREAD_COLOR)
I3_img = cv2.cvtColor(I3_img, cv2.COLOR_BGR2RGB)
I3_img_gray = cv2.cvtColor(I3_img, cv2.COLOR_BGR2GRAY)
I3_img_float = I3_img.astype(np.float64)
I3_img_gray_float = I3_img_gray.astype(np.float64)

I1_img_resized = cv2.resize(I1_img, (int(0.5*I1_img.shape[1]),int(0.5*I1_img.shape[0])))
I2_img_resized = cv2.resize(I2_img, (int(0.5*I2_img.shape[1]),int(0.5*I2_img.shape[0])))
I1_img_resized_3 = cv2.resize(I1_img, (int(0.2*I1_img.shape[1]),int(0.2*I1_img.shape[0])))
I3_img_resized = cv2.resize(I3_img, (int(0.2*I3_img.shape[1]),int(0.2*I3_img.shape[0])))
```

Helper

```
[ ]  1 def depth(f, baseline, disparity):
     2     Z = (f * baseline) / disparity
     3     return Z
     4
     5 def get_car(car, top_left, bottom_right):
     6     x0 = int(top_left[0])
     7     x1 = int(bottom_right[0])
     8     y0 = int(top_left[1])
     9     y1 = int(bottom_right[1])
    10     cropped = car[y0:y1, x0:x1]
    11     top_left = (y0, x0)
    12     bottom_right = (y1, x1)
    13     return cropped, top_left, bottom_right
```

NC

```
[ ]  1 def nc(patch_l, patch_r):
     2     product = np.sum(patch_l * patch_r)
     3     norm_l = np.sum(patch_l ** 2)
     4     norm_r = np.sum(patch_r ** 2)
     5     numerator = product
     6     denominator = np.sqrt(norm_l * norm_r)
     7     return numerator / denominator
```

Stereo Match

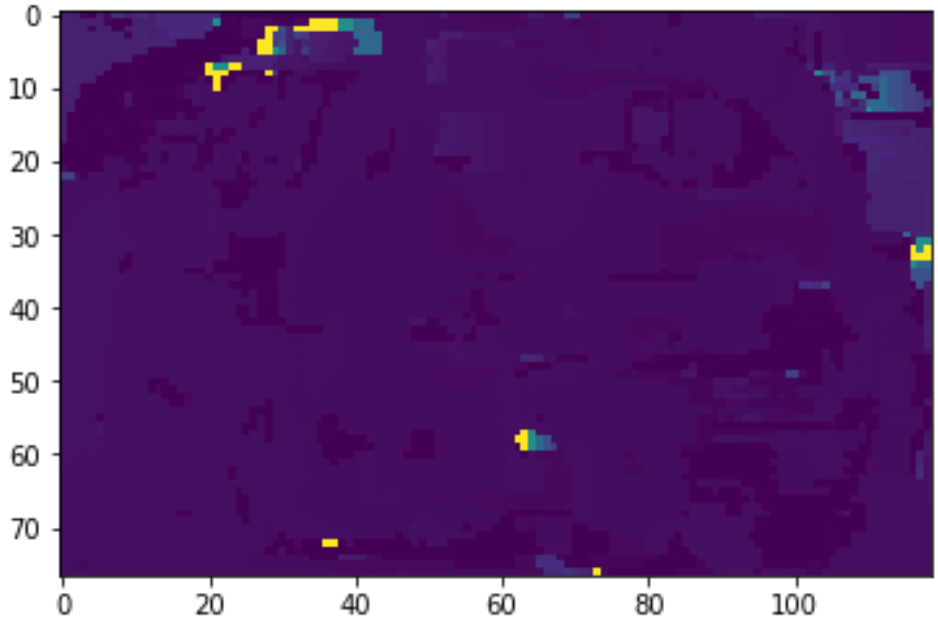
```
 1 def stereo_match(left_img, right_img, patch_size, box_top_left, box_bottom_right):
 2
 3     car, top_left, bottom_right = get_car(left_img, box_top_left, box_bottom_right)
 4
 5     result = np.copy(car)
 6     # result = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
 7
 8     x0 = top_left[1]
 9     y0 = top_left[0]
10     x1 = bottom_right[1]
11     y1 = bottom_right[0]
12
13     patch_center = top_left
14     scanline = y0
15     right_row = right_img.shape[0]
16     right_col = right_img.shape[1]
17
18     d = 0
19     dy = 0
20     dx = 0
22   for y in range(y0, y1):
23       print(".", end="", flush=True)
24
25       for x_left in range(x0, x1):
26
27           patch_left_center = (y, x_left)
28           patch_left_x0 = x_left - math.floor(patch_size / 2)
29           patch_left_y0 = y - math.floor(patch_size / 2)
30           patch_left_x1 = x_left + math.ceil(patch_size / 2)
31           patch_left_y1 = y + math.ceil(patch_size / 2)
32
33           patch_left = np.copy(left_img[patch_left_y0:patch_left_y1, patch_left_x0:patch_left_x1])
34           patch_right_best = np.zeros([patch_left_y1 - patch_left_y0, patch_left_x1 - patch_left_x0])
35           nc_best = 0
36           x_right_best = 0
37
38           for x_right in range(math.floor(patch_size / 2), x_left):
39
40               patch_right_center = (y, x_right)
41               patch_right_x0 = x_right - math.floor(patch_size / 2)
42               patch_right_y0 = y - math.floor(patch_size / 2)
43               patch_right_x1 = x_right + math.ceil(patch_size / 2)
44               patch_right_y1 = y + math.ceil(patch_size / 2)
45
46               patch_right = np.copy(right_img[patch_right_y0:patch_right_y1, patch_right_x0:patch_right_x1])
47               nc_value = nc(patch_left, patch_right)
48
49               if nc_value > 1000:
50                   continue
51               if nc_value > nc_best:
52                   nc_best = nc_value
53                   patch_right_best = patch_right
54                   x_right_best = x_right
```

```
            # print("nc_best = ", nc_best)
            # print("patch right = ", patch_right_best)
            disparity = x_left - x_right_best
            # print("x_left = ", x_left)
            # print("x_right_best = ", x_right_best)
            # print("disparity = ", disparity)
            d = depth(f, baseline, disparity)
            # print("depth = ", d)
            # print("============================================================")

            result[dy, dx] = d
            dx += 1
        dy += 1
        dx = 0

    plt.imshow(result)
    return result
```

TEST

```
[ ]  1 stereo_match(left_img_gray_float, right_img_gray_float, 7, left_car_top_left, left_car_bottom_right)
```

Result:



You can see there are some pixels which show clearly different colors. They are outliers from incorrect point correspondences.

Q2. (b). After you compute depth using scanline (above), go to KITTI Stereo 2015 (http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo), and pick a machine learning model from the scoreboard. You can pick a model that comes with code and a pre-trained model so you don't need to implement yourself. Compute the depth for the whole image using their pretrained model. Compare your results from the previous questions with this results from this model. What is the difference, both in terms of quality and speed?

Model: GANet from KITTI Stereo 2015: https://github.com/feihuzhang/GANet

Import

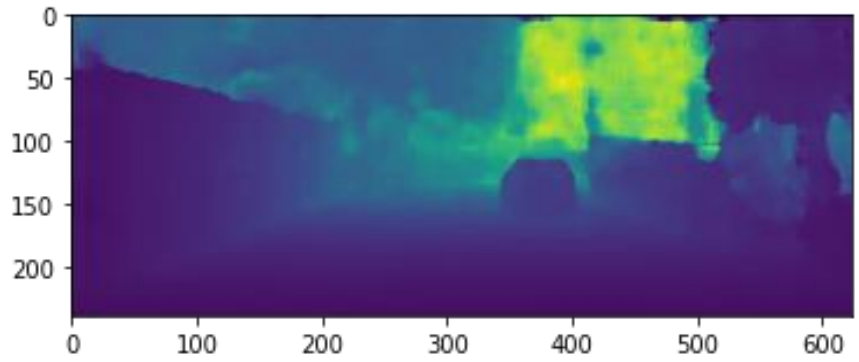
```
[ ]  1 %cd "drive"
     2 %cd "My Drive"
     3 %cd "csc420"
     4 %cd "a4"
     5 %cd "GANet"
     6 !chmod 777 ./compile.sh
     7 !./compile.sh
```

```
[ ]  1 import models.GANet_deep as GANet
```

```
[ ]  1 %cd ..
     2 %cd ..
     3 %cd ..
     4 %cd ..
     5 %cd ..
```

```
[ ]  1 if torch.cuda.is_available():
     2     torch.cuda.empty_cache()
     3
     4 net = nn.DataParallel(GANet.GANet())
     5 net.load_state_dict(torch.load("./drive/My Drive/csc420/a4/kitti2015_final.pth")["state_dict"])
```

```
1 transform = transforms.Compose([transforms.ToPILImage(),
2                                 transforms.Resize((240, 624)),
3                                 transforms.ToTensor()])
4 left_img_transformed = transform(left_img).reshape(1,3,240,624)
5 right_img_transformed = transform(right_img).reshape(1,3,240,624)
6
7 disparity, _, _ = net(left_img_transformed, right_img_transformed)
8 disparity_narray = disparity.cpu().detach().numpy()[0]
9 depth = f * baseline / disparity_narray
10
11 plt.imshow(depth)
```



Comparing to the previous result, this is better. The quality of output is much higher, and the time-cost is much lower.

Q2. (c). Write a short summary of workflow for the model you pick, e.g. what layers do they use, what modules do they use, etc.

Layers: This model has two novel neural net layers, aimed at capturing local and the whole-image cost dependencies respectively. First, the Semi-Global Aggregation (SGA) Layer. This is a differentiable semi-global matching (SGM) over the whole image. Second, the Local Guided Aggregation (LGA) Layer. This follows a traditional cost filtering strategy to refine thin structures and edges. It learns guided filtering and recovers the loss of accuracy in down-sampling.

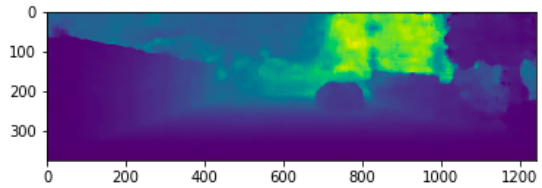
Workflow: The inputs (i.e. left and right images) are fed to a weight-sharing feature extraction pipeline. It consists of a stacked hourglass CNN and is connected by concatenations. The extracted left and right image features are then used to form a 4D cost volume, which is fed into a cost aggregation block for regularization, refinement and disparity regression. The guidance subnet (green) generates the weight matrices for the guided cost aggregations (SGA and LGA). SGA layers semi-globally aggregate the cost volume in four directions. The LGA layer is used before the disparity regression and locally refines the 4D cost volume for several times.

Modules: PyTorch, Numpy, math, skimage, PIL, etc.

Q2. (d). Using the depth information from models, try to determine within the bounding box, which pixel belong to the car based on its distance to the box center pixel's 3D location (use a threshold). After that, try to determine a 3D bounding box for the car (min & max along X, Y, Z). Visualize the segmentation of pixels within the 2D box, and on another image visualize the 3D box. State the classification threshold for distance.

```
resized_result = cv2.resize(depth, (1242,375), interpolation = cv2.INTER_CUBIC)
plt.imshow(resized_result)
```

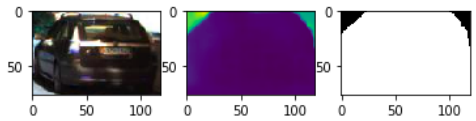
```
<matplotlib.image.AxesImage at 0x7f674ba0bd30>
```



2D Box Segmentation:

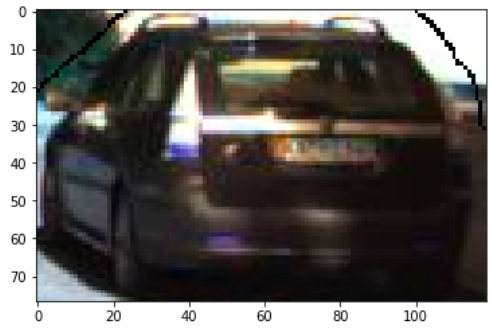
```
car, top_left, bottom_right = get_car(left_img, left_car_top_left, left_car_bottom_right)
plt.subplot(1,3,1)
plt.imshow(car)

car_depth, top_left, bottom_right = get_car(resized_result, left_car_top_left, left_car_bottom_right)
plt.subplot(1,3,2)
plt.imshow(car_depth)

ret, binary_mask = cv2.threshold(car_depth,50,1,0)
plt.subplot(1,3,3)
plt.imshow(binary_mask,cmap=plt.cm.binary)
```

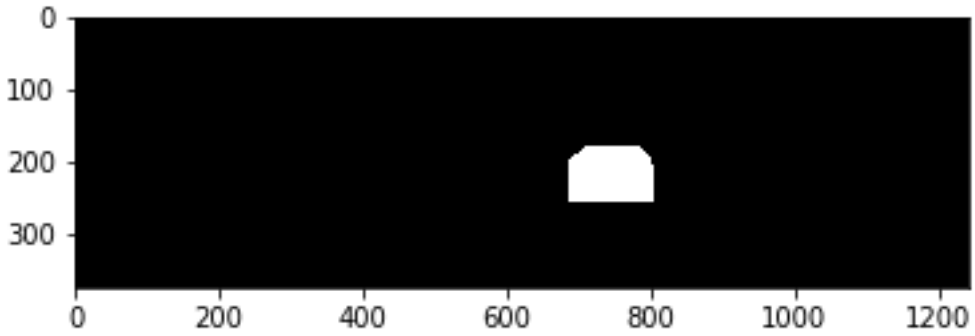
```
<matplotlib.image.AxesImage at 0x7f50e1ae0ba8>
```




```
output = np.array(binary_mask)*255
result = np.copy(car)
output_canny = cv2.Canny(np.uint8(output), 20, 250)
result[output_canny == 255] = [0,1,0]
plt.imshow(result)
```

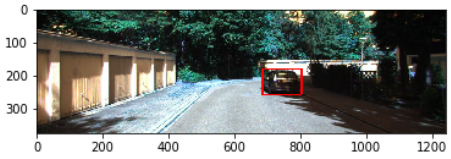
```
<matplotlib.image.AxesImage at 0x7f50e1b9b860>
```





```
draw_2D_box = np.copy(left_img)
print("top_left = ", top_left)
print("bottom_right = ", bottom_right)
draw_2D_box = cv2.rectangle(draw_2D_box, (top_left[1], top_left[0]), (bottom_right[1], bottom_right[0]), (255,0,
plt.imshow(draw_2D_box)
```

```
top_left =  (181, 685)
bottom_right =  (258, 804)
<matplotlib.image.AxesImage at 0x7f674ba09240>
```



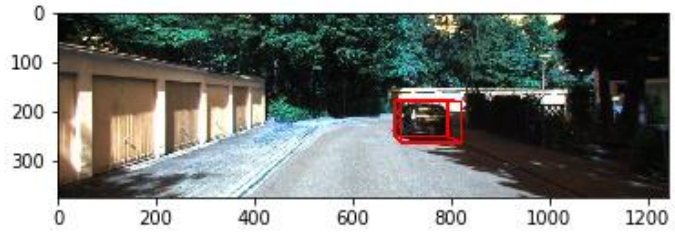
3D Box:

```
def draw_3D_box(depth_img, original_img, box_top_left, box_bottom_right, threshold):
    car_depth, top_left, bottom_right = get_car(depth_img, box_top_left, box_bottom_right)
    car, top_left, bottom_right = get_car(original_img, box_top_left, box_bottom_right)

    x0 = top_left[1]
    y0 = top_left[0]
    x1 = bottom_right[1]
    y1 = bottom_right[0]

    result_3d = np.copy(left_img)

    depths_patch = car_depth
    left_patch = car

    Z = depths_patch
    x = np.ones((Z.shape[0], 1)) * np.arange(x0, x1)
    y = (np.ones((Z.shape[1], 1)) * np.arange(y0, y1)).T

    X = Z * (x - px) / f
    Y = Z * (y - py) / f
    distance = (X - X[X.shape[0] // 2, X.shape[1] // 2]) ** 2 + \
               (Y - Y[Y.shape[0] // 2, Y.shape[1] // 2]) ** 2 + \
               (Z - Z[Z.shape[0] // 2, Z.shape[1] // 2]) ** 2

    x_threshold = X[distance < threshold]
    y_threshold = Y[distance < threshold]
    z_threshold = Z[distance < threshold]

    P = np.array([np.array([np.min(x_threshold), np.min(y_threshold), np.min(z_threshold)]),
                  np.array([np.min(x_threshold), np.min(y_threshold), np.max(z_threshold)]),
                  np.array([np.min(x_threshold), np.max(y_threshold), np.min(z_threshold)]),
                  np.array([np.min(x_threshold), np.max(y_threshold), np.max(z_threshold)]),
                  np.array([np.max(x_threshold), np.min(y_threshold), np.min(z_threshold)]),
                  np.array([np.max(x_threshold), np.min(y_threshold), np.max(z_threshold)]),
                  np.array([np.max(x_threshold), np.max(y_threshold), np.min(z_threshold)]),
                  np.array([np.max(x_threshold), np.max(y_threshold), np.max(z_threshold)])])

    points = []
    for i in range(P.shape[0]):
        points.append((int(np.round(f * P[i,0] / P[i,2] + px)), int(np.round(f * P[i,1] / P[i,2] + py))))
        print("point ", i, " = ", (int(np.round(f * P[i,0] / P[i,2] + px)), int(np.round(f * P[i,1] / P[i,2] + p
```

```
    cv2.line(result_3d, points[0], points[1], (255,0,0), 4)
    cv2.line(result_3d, points[0], points[2], (255,0,0), 4)
    cv2.line(result_3d, points[0], points[4], (255,0,0), 4)
    cv2.line(result_3d, points[1], points[3], (255,0,0), 4)
    cv2.line(result_3d, points[1], points[5], (255,0,0), 4)
    cv2.line(result_3d, points[2], points[3], (255,0,0), 4)
    cv2.line(result_3d, points[2], points[6], (255,0,0), 4)
    cv2.line(result_3d, points[3], points[7], (255,0,0), 4)
    cv2.line(result_3d, points[4], points[5], (255,0,0), 4)
    cv2.line(result_3d, points[4], points[6], (255,0,0), 4)
    cv2.line(result_3d, points[5], points[7], (255,0,0), 4)
    cv2.line(result_3d, points[6], points[7], (255,0,0), 4)

    plt.imshow(result_3d)
```

```
draw_3D_box(resized_result, left_img, left_car_top_left, left_car_bottom_right, 3)
```

```
point  0  =  (698, 181)
point  1  =  (686, 180)
point  2  =  (698, 264)
point  3  =  (686, 252)
point  4  =  (821, 181)
point  5  =  (793, 180)
point  6  =  (821, 264)
point  7  =  (793, 252)
```



Q3. Fundamental Matrix - for this question, take 3 images (I1, I2, I3) of an object or a stationary scene as follows: Images I1 , I2 from almost the same viewpoint, but with a (roll) rotation. That is, take an image (I1), don't move, but rotate the camera in place around 30-45 degrees, and take another image (I2). Take the third image (I3) from a different viewpoint, e.g. move the camera~20 cm to the right and rotate (out of plane) to point the camera towards the object again.

Q3. (a).
Use SIFT matching (or any other point matching technique) to find a number of point correspondences in the (I1, I2) image pair and in the (I1, I3) image pair. Visualize the results. If there are any outliers, either manually remove them or increase the matching threshold so no outliers remain. Pick 8 point correspondences from the remaining set for each image pair, i.e. (I1, I2) and (I1, I3). Visualize those 8 point matches. It helps in the later steps if the 8 point matches are somewhat distributed over the images rather than being clustered in a small region.

Distribute points:

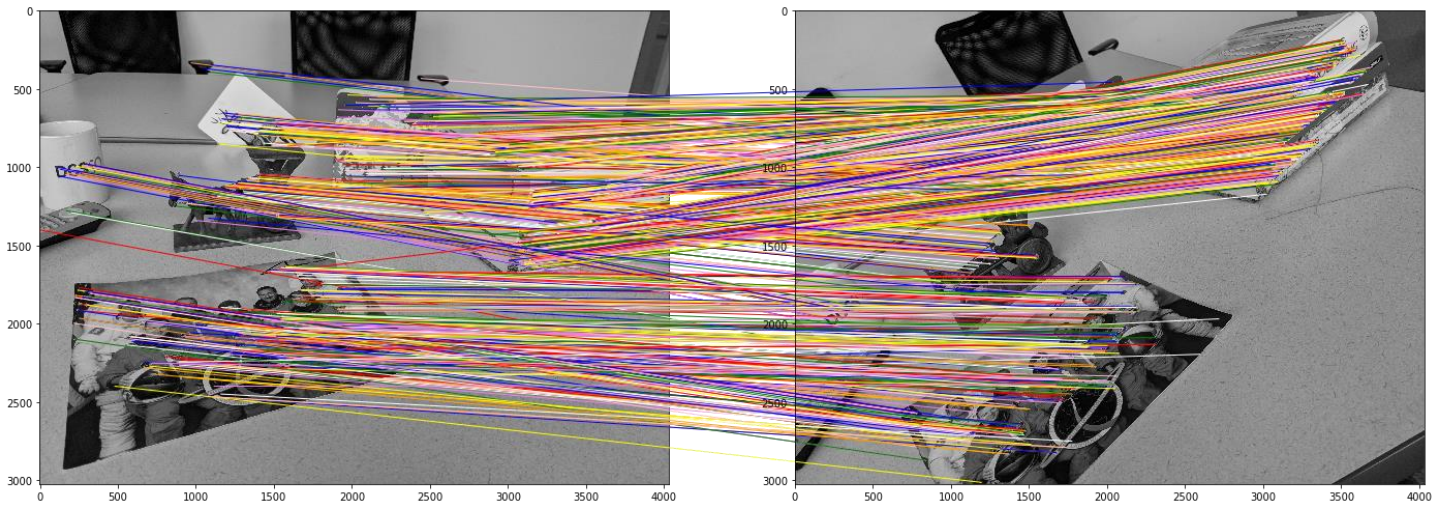
```python
def bad_points(top_eight, d):
    bad = []
    for item in top_eight:
        distance_list = []
        index = top_eight.index(item)
        for i in range(len(top_eight)):
            if i != index:
                distance = np.sum(np.array(item[0][0].pt) - np.array(top_eight[i][0][0].pt))**2
                # print("for #", index, " point, the distance with #", i, " = ", distance)
                distance_list.append(distance)
        for dis in distance_list:
            if dis < d:
                bad.append(item)
                break
                # print("bad point = ", item)
    return bad

def remove_bad_points(top_eight, bad_points):
    if bad_points is False:
        return top_eight

    for item in bad_points:
        top_eight.remove(item)
    return top_eight
def best_eight_with_distance(top_eight, sorted_keypointes, d):
    # print("num of points = ", len(top_eight))
    index = 8
    bad_points_list = bad_points(top_eight, d)

    if bad_points_list is False:
        return top_eight

    else:
        rest_points_list = remove_bad_points(top_eight, bad_points_list)

    while bad_points_list:

        # print("num of rest points = ", len(rest_points_list))
        num_of_new_points = len(bad_points_list)
        # print("num of bad and new points = ", len(bad_points_list))

        for i in range(index, index + num_of_new_points):
            # print("index = ", i)
            rest_points_list.append(sorted_keypointes[i])

        index = index + num_of_new_points
        # print("next index = ", index)
        bad_points_list = bad_points(rest_points_list, d)
        # print("next num of bad and new points = ", len(bad_points_list))
        rest_points_list = remove_bad_points(rest_points_list, bad_points_list)
        # print("next num of rest points = ", len(rest_points_list))
        # print("================================================================")

    return rest_points_list
```

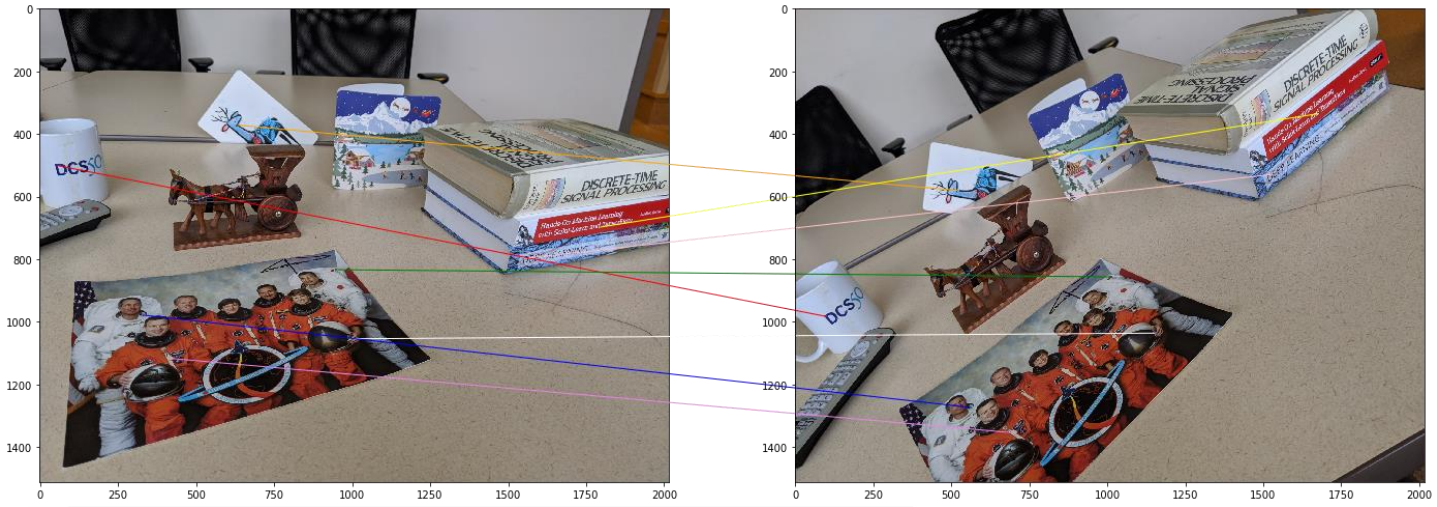
SIFT:

```python
def sift_extract(image, resize):
    img = cv2.imread(image)
    img = cv2.resize(img, (int(resize*img.shape[1]),int(resize*img.shape[0])))
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    sift = cv2.xfeatures2d.SIFT_create(2000)

    keypoints, features = sift.detectAndCompute(gray_img, None)
    result = cv2.drawKeypoints(gray_img, keypoints, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS, outImage=N

    # cv2.imwrite('sift_keypoints.jpg', result)
    return keypoints, features
def sift_matching(image1, image2, ratio_threshold, norm_level, d, resize):
    img1 = cv2.imread(image1)
    img2 = cv2.imread(image2)
    img1 = cv2.resize(img1, (int(resize*img1.shape[1]),int(resize*img1.shape[0])))
    img2 = cv2.resize(img2, (int(resize*img2.shape[1]),int(resize*img2.shape[0])))
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    kp_1, des_1 = sift_extract(image1, resize)
    kp_2, des_2 = sift_extract(image2, resize)
    num_des_1 = np.shape(des_1)[0]
    num_des_2 = np.shape(des_2)[0]
    num_matching_keypoint = num_des_1

    candidate_keypoint_location = []
    candidate_distance = []

    for i in range(num_des_1):
        distance = []
        for j in range(num_des_2):
            euclidean_dis = euclidean_distance(des_1[i], des_2[j], norm_level)
            distance.append(euclidean_dis)
        two_smallest_distance = heapq.nsmallest(2, distance)
        candidate_distance.append(two_smallest_distance)
        np_distance = np.array(distance)
        min_dis_index = np.unravel_index(np.argmin(np_distance, axis=None), np_distance.shape)[0]
        candidate_keypoint_location.append(min_dis_index)

    keypoint1 = []
    keypoint2 = []
    keypoints_pairs = {}
    count = 0
    for i in range(num_matching_keypoint):
        ratio = np.true_divide(candidate_distance[i][0], candidate_distance[i][1])
        if ratio < ratio_threshold:
            count = count + 1
            kp1 = kp_1[i]
            kp2 = kp_2[candidate_keypoint_location[i]]
            keypoint1.append(kp1)
            keypoint2.append(kp2)
            keypoints_pairs[(kp1,kp2)] = ratio

    sorted_keypointes = sorted(keypoints_pairs.items(), key=lambda kv: kv[1])

    print("number of matching = ", count)

    top_eight = sorted_keypointes[0:8]
    print("top eight = ", top_eight)

    best_eight = best_eight_with_distance(top_eight, sorted_keypointes, d)

    print("best eight = ", best_eight)

    result_img1 = cv2.drawKeypoints(gray1, keypoint1, outImage=None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYP
    result_img2 = cv2.drawKeypoints(gray2, keypoint2, outImage=None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYP
    cv2.imwrite("sample1_keypoints_final.jpg", result_img1)
    cv2.imwrite("sample2_keypoints_final.jpg", result_img2)
    # show_matching(gray1, keypoint1, gray2, keypoint2)

    return best_eight
```

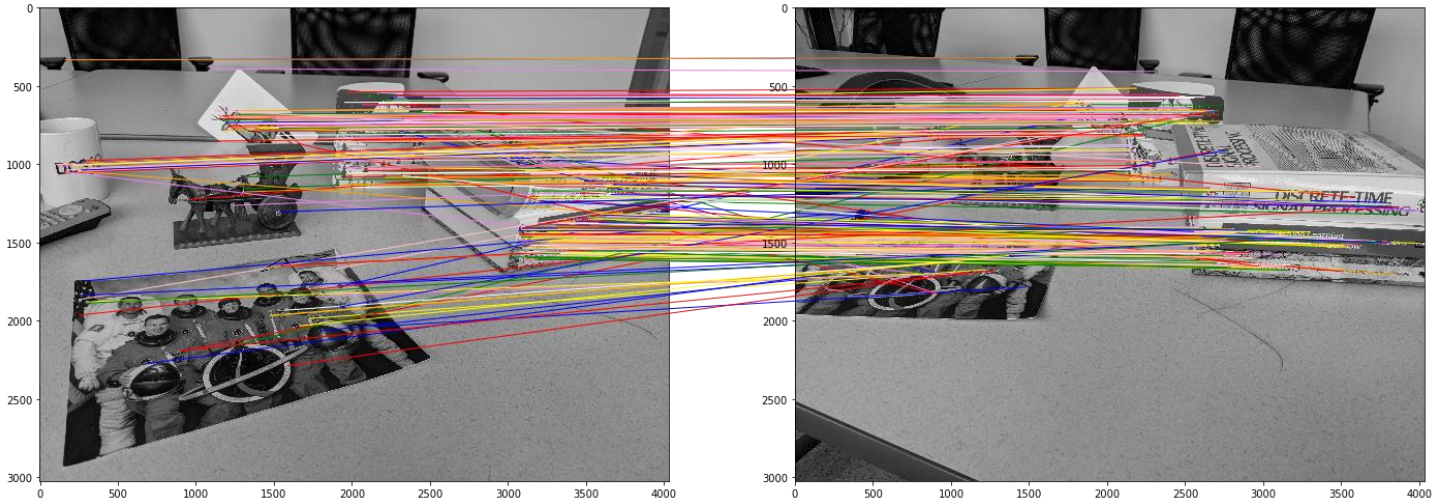
```python
def euclidean_distance(vector1, vector2, norm_level):
    vector1 = vector1.reshape(-1)
    vector2 = vector2.reshape(-1)
    distance = np.linalg.norm(vector1 - vector2, ord=norm_level)
    return distance
def show_matching(image1, keypoint1, image2, keypoint2):
    plt.figure(figsize=(24,64))
    ax1 = plt.subplot(1, 2, 1)
    ax2 = plt.subplot(1, 2, 2)
    img1 = cv2.drawKeypoints(image1, keypoint1, None)
    img2 = cv2.drawKeypoints(image2, keypoint2, None)

    ax1.imshow(img1)
    ax2.imshow(img2)

    colors = ["white", "red", "blue", "orange", "pink", "yellow", "violet", "green"]
    color_i = 0
    for kp1, kp2 in zip(keypoint1, keypoint2):
        coord1 = kp1.pt
        coord2 = kp2.pt
        c = colors[color_i % 8]
        con = ConnectionPatch(xyA=coord2, xyB=coord1, coordsA="data", coordsB="data", axesA=ax2, axesB=ax1, arro
        ax2.add_patch(con)
        color_i += 1

    plt.show()
def generate_top_eight_color(image1, image2, top, resize):
    img1 = cv2.imread(image1, cv2.IMREAD_COLOR)
    img2 = cv2.imread(image2, cv2.IMREAD_COLOR)
    img1 = cv2.resize(img1, (int(resize*img1.shape[1]),int(resize*img1.shape[0])))
    img2 = cv2.resize(img2, (int(resize*img2.shape[1]),int(resize*img2.shape[0])))
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)

    keypoint1 = []
    keypoint2 = []

    for item in top:
        keypoint1.append(item[0][0])
        keypoint2.append(item[0][1])

    show_matching(img1, keypoint1, img2, keypoint2)
```

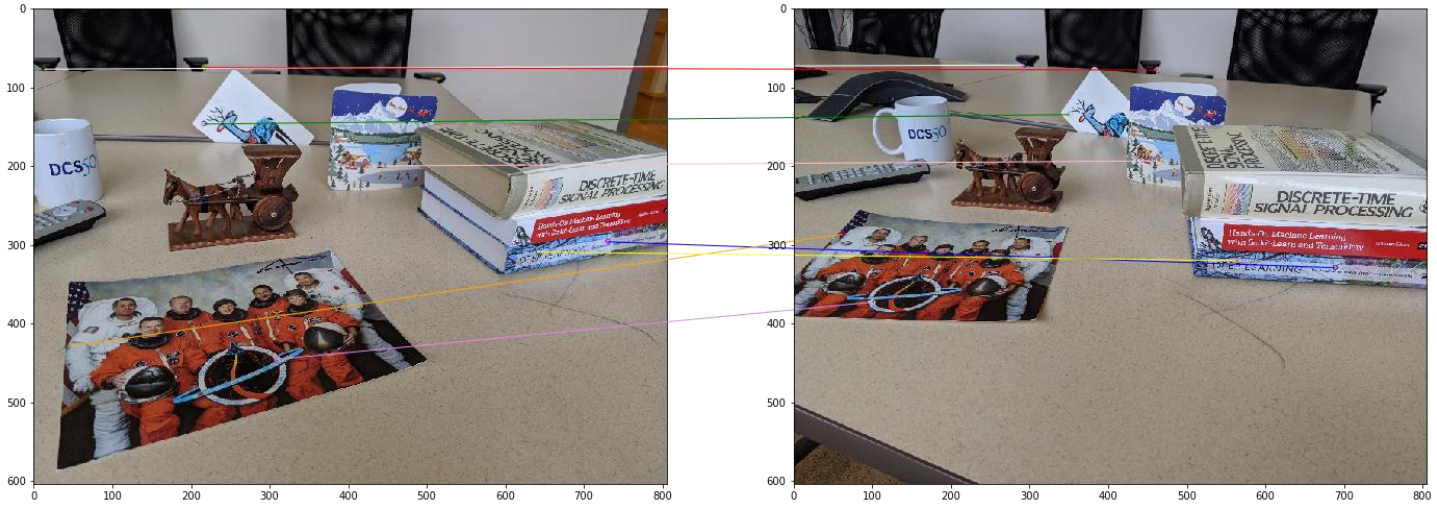
```python
[ ]  1 top_eight_i1i2 = sift_matching(I1_path, I2_path, 0.8, 2, 15205, 0.5)
     2 print(len(top_eight_i1i2))
     3 generate_top_eight_color(I1_path, I2_path, top_eight_i1i2, 0.5)
```





```
1 top_eight_ili3 = sift_matching(I1_path, I3_path, 0.8, 2, 5000, 0.2)
2 print(len(top_eight_ili3))
3 generate_top_eight_color(I1_path, I3_path, top_eight_ili3, 0.2)
```





Q3. (b).
Using what we have learned in class (standard 8-point algorithm) calculate the fundamental matrix F12 for image pair (I1, I2) and Fundamental matrix F13 for image pair (I1, I3). (for this question, implement your own standard 8-point algorithm)

```python
m2 = []
for item in top_eight_ili2:
    m2.append((item[0][0].pt,item[0][1].pt))
m2 = np.array(m2)

m3 = []
for item in top_eight_ili3:
    m3.append((item[0][0].pt,item[0][1].pt))
m3 = np.array(m3)

print("m2 = ", m2)
print("==========================================")
print("m3 = ", m3)
print("==========================================")
```

```python
m2_left = []
for item in m2:
    m2_left.append(item[0])
m2_left = np.array(m2_left)

m2_right = []
for item in m2:
    m2_right.append(item[1])
m2_right = np.array(m2_right)

print("m2_left = ", m2_left)
print("==========================================")
print("m2_right = ", m2_right)
print("==========================================")

m3_left = []
for item in m3:
    m3_left.append(item[0])
m3_left = np.array(m3_left)

m3_right = []
for item in m3:
    m3_right.append(item[1])
m3_right = np.array(m3_right)

print("m3_left = ", m3_left)
print("==========================================")
print("m3_right = ", m3_right)
print("==========================================")
```

```python
m2_left_x = []
for item in m2_left:
    m2_left_x.append(item[0])
m2_left_x = np.array(m2_left_x)

m2_left_y = []
for item in m2_left:
    m2_left_y.append(item[1])
m2_left_y = np.array(m2_left_y)

print("m2_left_x = ", m2_left_x)
print("==========================================")
print("m2_left_y = ", m2_left_y)
print("==========================================")

m2_right_x = []
for item in m2_right:
    m2_right_x.append(item[0])
m2_right_x = np.array(m2_right_x)

m2_right_y = []
for item in m2_right:
    m2_right_y.append(item[1])
m2_right_y = np.array(m2_right_y)

print("m2_right_x = ", m2_right_x)
print("==========================================")
print("m2_right_y = ", m2_right_y)
print("==========================================")
```

```python
m3_left_x = []
for item in m3_left:
    m3_left_x.append(item[0])
m3_left_x = np.array(m3_left_x)

m3_left_y = []
for item in m3_left:
    m3_left_y.append(item[1])
m3_left_y = np.array(m3_left_y)

print("m3_left_x = ", m3_left_x)
print("==========================================")
print("m3_left_y = ", m3_left_y)
print("==========================================")

m3_right_x = []
for item in m3_right:
    m3_right_x.append(item[0])
m3_right_x = np.array(m3_right_x)

m3_right_y = []
for item in m3_right:
    m3_right_y.append(item[1])
m3_right_y = np.array(m3_right_y)

print("m3_right_x = ", m3_right_x)
print("==========================================")
print("m3_right_y = ", m3_right_y)
print("==========================================")
```

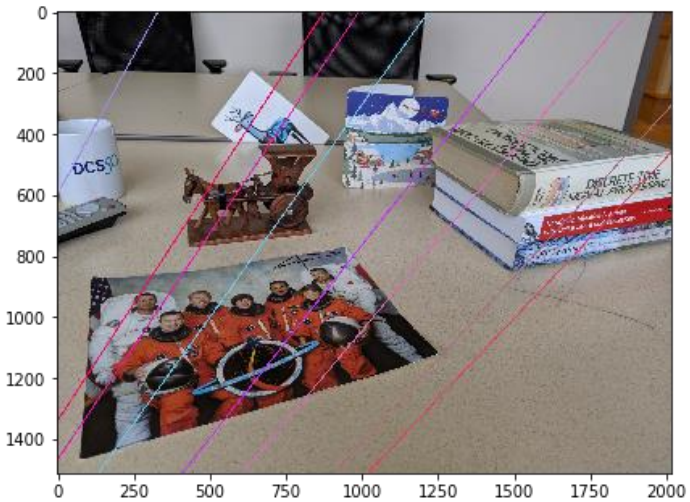
```python
def generate_F(m_left_x, m_left_y, m_right_x, m_right_y):

    a = [m_left_x * m_right_x, m_left_y * m_right_x, m_right_x,
         m_left_x * m_right_y, m_left_y * m_right_y, m_right_y,
         m_left_x, m_left_y, 1]

    A = np.zeros((8, 9))

    for i in range(9):
        A[:, i] = a[i]

    u, s, vh = np.linalg.svd(A)
    U, D, VT = np.linalg.svd(vh[-1].reshape(3,3))
    D[-1] = 0

    F = np.dot(np.dot(U, np.diag(D)), VT)

    return F

F2 = generate_F(m2_left_x, m2_left_y, m2_right_x, m2_right_y)
F3 = generate_F(m3_left_x, m3_left_y, m3_right_x, m3_right_y)

print("F_12 for I1 and I2 = ", F2)
print("F_13 for I1 and I3 = ", F3)
```

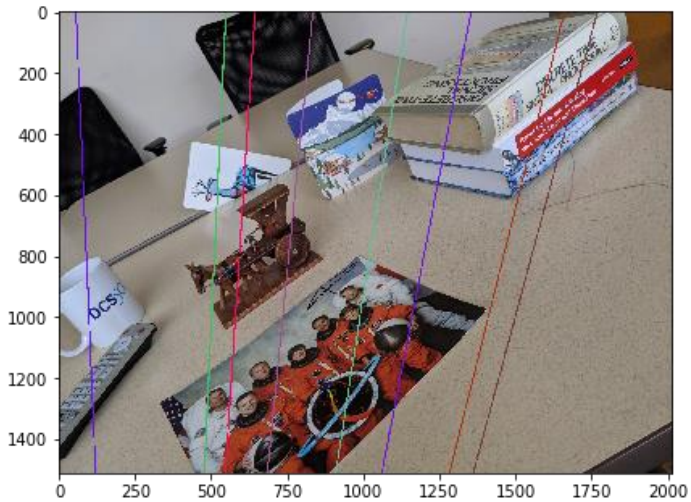
```
F_12 for I1 and I2 =  [[ 2.99966064e-07 -5.71906161e-07  3.82450139e-03]
 [ 6.58199699e-07  3.93666269e-07 -3.85383673e-04]
 [-3.69181592e-03 -1.98988072e-03  9.99983818e-01]]
F_13 for I1 and I3 =  [[ 1.05885143e-06  3.84134672e-06 -2.33830982e-03]
 [-1.59431908e-06  6.88838957e-06 -5.86826657e-03]
 [ 1.41536716e-03 -4.33544910e-04  9.99978952e-01]]
```

Q3. (c).
Using F12, calculate the epipolar lines in the right image for each of the 8 points in the left image and plot them on the right image. (for this question you can use any OpenCV functions you want)

```python
def drawlines(img1, lines):
    r = img1.shape[0]
    c = img1.shape[1]

    for r in lines:
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1]])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1]])
        img3 = cv2.line(img1, (x0,y0), (x1,y1), color, 3)

    return img3
```

```python
def epipolarline(img1_path, img2_path, kp1, kp2, F, resize):
    img1 = cv2.imread(img1_path, cv2.IMREAD_COLOR)
    img2 = cv2.imread(img2_path, cv2.IMREAD_COLOR)
    img1 = cv2.resize(img1, (int(resize*img1.shape[1]),int(resize*img1.shape[0])))
    img2 = cv2.resize(img2, (int(resize*img2.shape[1]),int(resize*img2.shape[0])))
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)

    points1 = np.int32(kp1)
    points2 = np.int32(kp2)

    lines1 = cv2.computeCorrespondEpilines(points2.reshape(-1,1,2),2,F)
    lines1 = lines1.reshape(-1,3)
    img1 = drawlines(img1, lines1)

    lines2 = cv2.computeCorrespondEpilines(points1.reshape(-1,1,2),1,F)
    lines2 = lines2.reshape(-1,3)
    img2 = drawlines(img2, lines2)

    fig = plt.figure(figsize=(15, 15))
    plt.subplot(1,2,1),plt.imshow(img1)
    plt.subplot(1,2,2),plt.imshow(img2)
    plt.show()

    return img1, img2
```

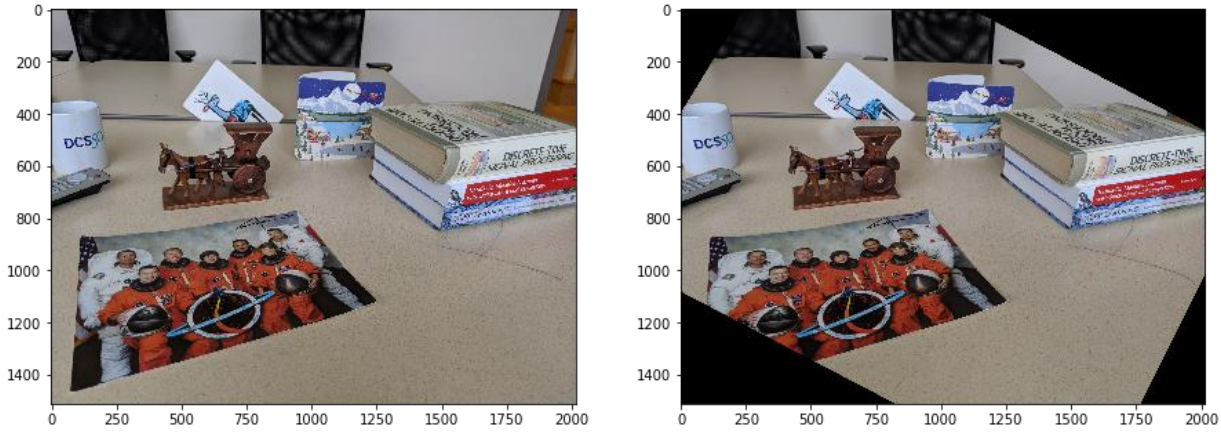




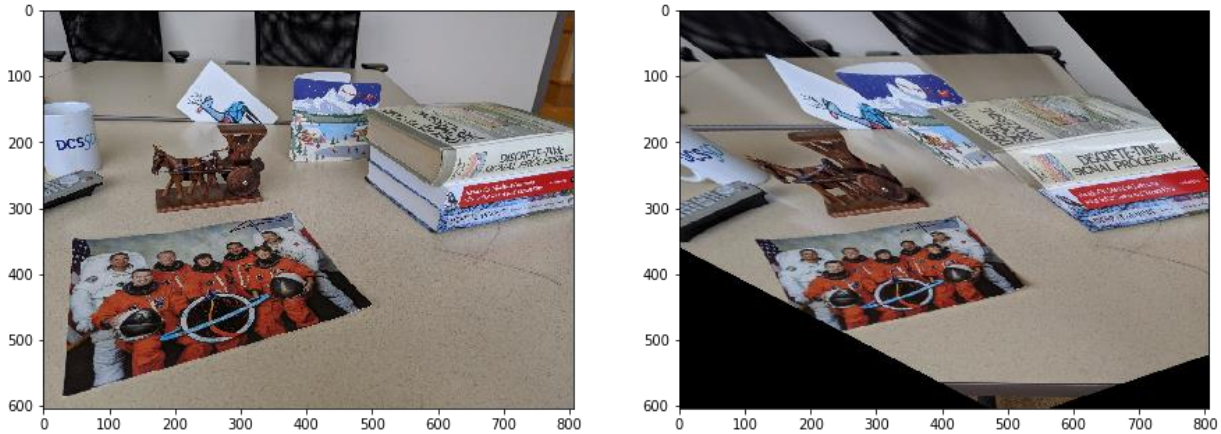
Q3. (d).
Using F12, rectify I2 with I1 and visualize the resulting image side by side with I1. Do the same for I3 using F13. (for this question you can use any OpenCV functions you want)
Resize:

```
resize_2 = 0.5
resize_3 = 0.2
col_2 = int(resize_2*I1_img.shape[1])
row_2 = int(resize_2*I1_img.shape[0])
col_3 = int(resize_3*I1_img.shape[1])
row_3 = int(resize_3*I1_img.shape[0])
```

```
_, H2_left, H2_right = cv2.stereoRectifyUncalibrated(m2_left, m2_right, F2, (col_2, row_2))

H2_new = np.linalg.inv(H2_left).dot(H2_right)

rec2_right = cv2.warpPerspective(I2_img_resized, H2_new, (col_2, row_2))

fig = plt.figure(figsize=(15, 30))
ax1 = fig.add_subplot(1,2,1)
ax1.imshow(I1_img_resized)
ax2 = fig.add_subplot(1,2,2)
ax2.imshow(rec2_right)
plt.show()
```



```
_, H3_left, H3_right = cv2.stereoRectifyUncalibrated(m3_left, m3_right, F3, (col_3, row_3))

H3_new = np.linalg.inv(H3_left).dot(H3_right)

rec3_right = cv2.warpPerspective(I3_img_resized, H3_new, (col_3, row_3))

fig = plt.figure(figsize=(15, 30))
ax1 = fig.add_subplot(1,2,1)
ax1.imshow(I1_img_resized_3)
ax2 = fig.add_subplot(1,2,2)
ax2.imshow(rec3_right)
plt.show()
```



Q3. (e).
Using OpenCV, compute F’12 and F’13 and compare with your results. I.e. Are they the same? Are they similar? Briefly discuss.

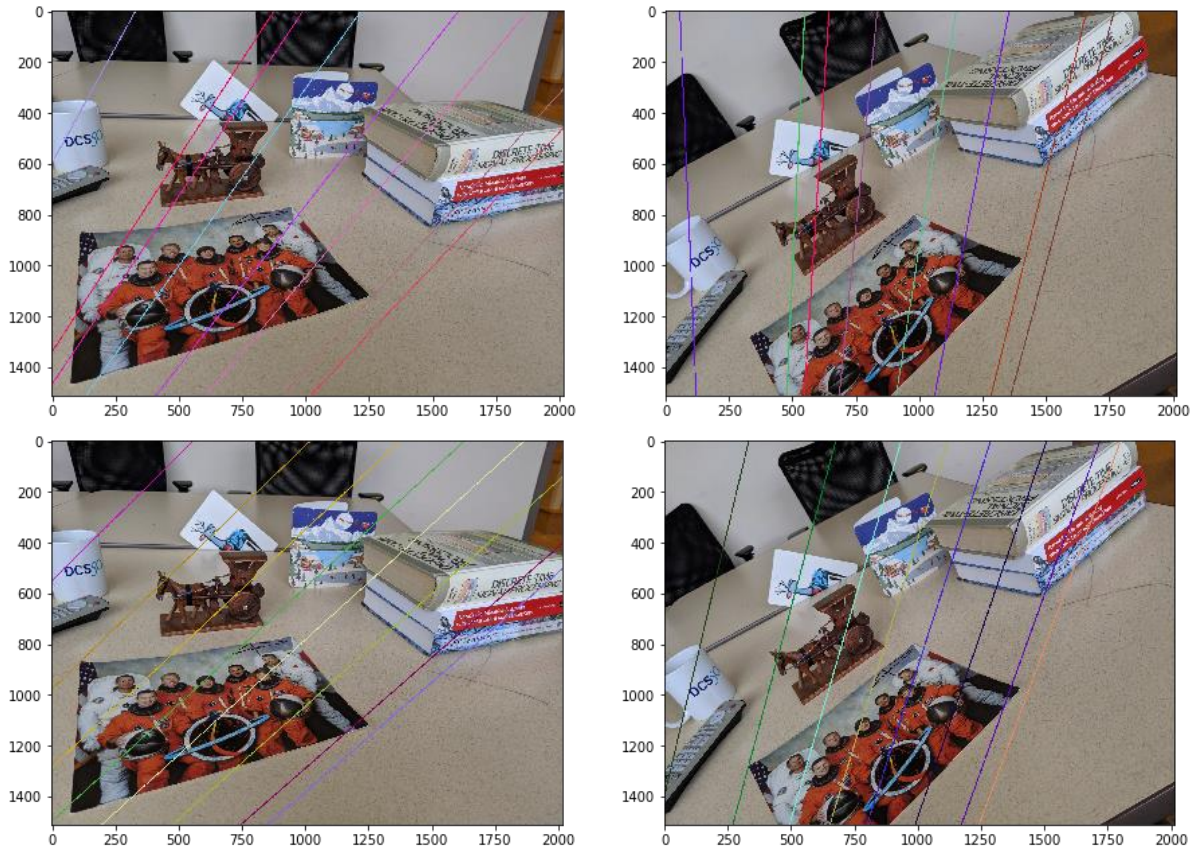
```
def cv2_F(m_left, m_right):
    F, mask = cv2.findFundamentalMat(np.int32(m_left),np.int32(m_right),cv2.FM_8POINT)
    return F

F2_prime = cv2_F(m2_left, m2_right)
F3_prime = cv2_F(m3_left, m3_right)

print("F_12 prime = ", F2_prime)
print("F_13 prime = ", F3_prime)
```

```
F_12 prime =  [[ 2.25324436e-07 -5.31586029e-07  1.06922119e-02]
 [ 7.78711153e-07  6.06811768e-07  2.13927053e-03]
 [-8.19916954e-03 -8.04072290e-03  1.00000000e+00]]
F_13 prime =  [[ 1.20394471e-06  4.82397810e-06 -2.37948066e-03]
 [-3.30851062e-06  8.98631879e-06 -5.22083256e-03]
 [ 1.53013774e-03 -1.48049222e-03  1.00000000e+00]]
```

Compare with my F12 and F13 before:

```
F_12 for I1 and I2 =  [[ 2.99966064e-07 -5.71906161e-07  3.82450139e-03]
 [ 6.58199699e-07  3.93666269e-07 -3.85383673e-04]
 [-3.69181592e-03 -1.98988072e-03  9.99983818e-01]]
F_13 for I1 and I3 =  [[ 1.05885143e-06  3.84134672e-06 -2.33830982e-03]
 [-1.59431908e-06  6.88838957e-06 -5.86826657e-03]
 [ 1.41536716e-03 -4.33544910e-04  9.99978952e-01]]
```

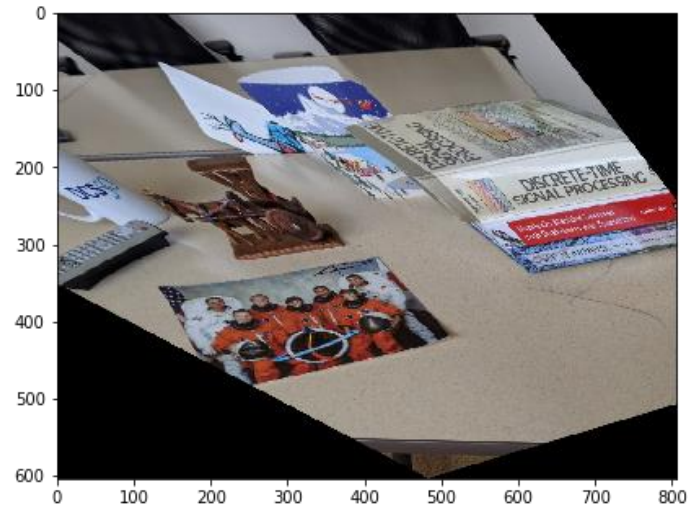
They are not the same.
Because my F generator function calculates the fundamental matrix by following the lecture slides and notes. The results of cv2 are slightly better than mine since the OpenCV performs automatically normalization and optimization.
e.g. my F (two images at the first row) VS F from cv2 (two images at the second row)
(i.e. comparing the epipolar lines)



Q3. (f).
Using OpenCV, rectify the images using F'12 and F'13 and compare with your rectifications (part d). Discuss any differences.

```
_, H2_left_prime, H2_right_prime = cv2.stereoRectifyUncalibrated(m2_left, m2_right, F2_prime, (col_2, row_2))

H2_prime_new = np.linalg.inv(H2_left_prime).dot(H2_right_prime)

rec2_right_prime = cv2.warpPerspective(I2_img_resized, H2_prime_new, (col_2, row_2))

fig = plt.figure(figsize=(15, 30))
ax1 = fig.add_subplot(1,2,1)
ax1.imshow(I1_img_resized)
ax2 = fig.add_subplot(1,2,2)
ax2.imshow(rec2_right_prime)
plt.show()
```


```
_, H3_left_prime, H3_right_prime = cv2.stereoRectifyUncalibrated(m3_left, m3_right, F3_prime, (col_3, row_3))

H3_prime_new = np.linalg.inv(H3_left_prime).dot(H3_right_prime)

rec3_left_prime = cv2.warpPerspective(I3_img_resized, H3_prime_new, (col_3, row_3))
fig = plt.figure(figsize=(15, 30))
ax1 = fig.add_subplot(1,2,1)
ax1.imshow(I1_img_resized_3)
ax2 = fig.add_subplot(1,2,2)
ax2.imshow(rec3_left_prime)
plt.show()
```



The differences between Q3.(d) are minor. They are similar.