CSC420 Assignment 3

Xinqi Shen

1.(a) NC performs better. Since SSD just simply compared the corresponding pixel values in each patch, which is very sensitive to the intensity or brightness change of the image. In the situation that two images are captured by the same camera but under different exposure times, it implies the large difference on the intensity or brightness change of two images. For NC (normalized correlation), since it normalized when calculating the correlation which helped eliminate the effects of intensity or brightness change. Thus, NC is more accurate than SSD in this case.

2.(a)

patch size: 5

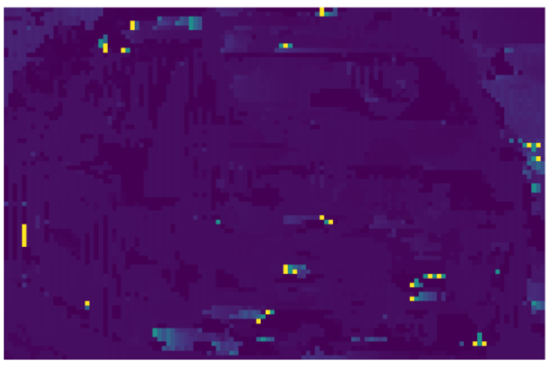
sampling method: compare the patches of even index of pixel from scanline (instead of comparing with all possible patches, we skip one pixel/patch every time). This sampling method will double the speed.

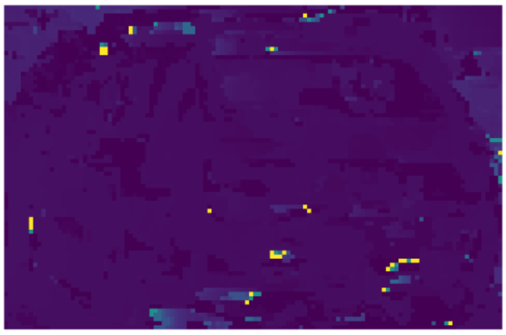
matching cost function: NC (normalized correlation)

For each pixel inside the bounding box in the left image, depth is computed by following steps:

1. Find the matching pixel's coordinate in the right image (only need find x coordinate due to rectified image, y coordinate is the same) by computing the matching cost in each patch from scanline and look for maxima.
2. After finding the matching pixel, the depth is calculated by formula $Z = \frac{f*T}{x_l - x_r}$

   where f is focal length, T is baseline length and $x_l - x_r$ is disparity (the difference of x coordinate between left and right image)

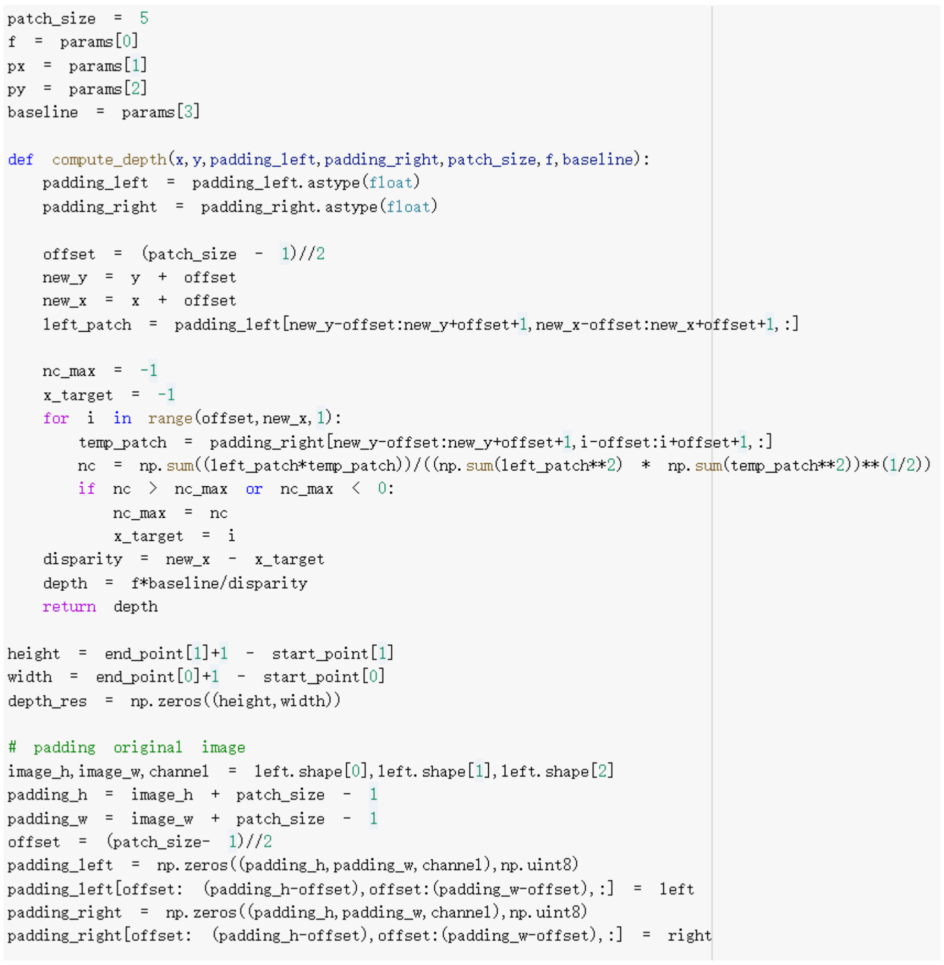


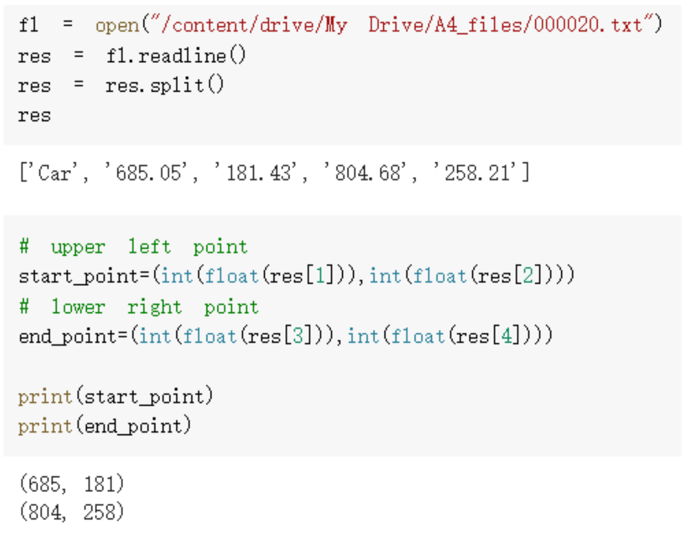

(without using sampling method)

I also attached the depth image without using sampling method, it just looked more smoothed, no large difference compared with using sampling method depth image. We reduced computation complexity by half and get similar result. :)

Outliers: There are some outliers clearly shown in the depth image, you can see some lighter yellow/blue pixels in the image, and it indicated the incorrect point correspondences. Since the depth is calculated from the disparity, if some pixels are mismatching, it will get incorrect depth and display as noise in the depth image.

Code:


```
patch_size = 5
f = params[0]
px = params[1]
py = params[2]
baseline = params[3]

def compute_depth(x,y,padding_left,padding_right,patch_size,f,baseline):
    padding_left = padding_left.astype(float)
    padding_right = padding_right.astype(float)

    offset = (patch_size - 1)//2
    new_y = y + offset
    new_x = x + offset
    left_patch = padding_left[new_y-offset:new_y+offset+1,new_x-offset:new_x+offset+1,:]

    nc_max = -1
    x_target = -1
    for i in range(offset,new_x,1):
        temp_patch = padding_right[new_y-offset:new_y+offset+1,i-offset:i+offset+1,:]
        nc = np.sum((left_patch*temp_patch))/((np.sum(left_patch**2) * np.sum(temp_patch**2))**(1/2))
        if nc > nc_max or nc_max < 0:
            nc_max = nc
            x_target = i
    disparity = new_x - x_target
    depth = f*baseline/disparity
    return depth

height = end_point[1]+1 - start_point[1]
width = end_point[0]+1 - start_point[0]
depth_res = np.zeros((height,width))

# padding original image
image_h,image_w,channel = left.shape[0],left.shape[1],left.shape[2]
padding_h = image_h + patch_size - 1
padding_w = image_w + patch_size - 1
offset = (patch_size- 1)//2
padding_left = np.zeros((padding_h,padding_w,channel),np.uint8)
padding_left[offset: (padding_h-offset),offset:(padding_w-offset),:] = left
padding_right = np.zeros((padding_h,padding_w,channel),np.uint8)
padding_right[offset: (padding_h-offset),offset:(padding_w-offset),:] = right
```

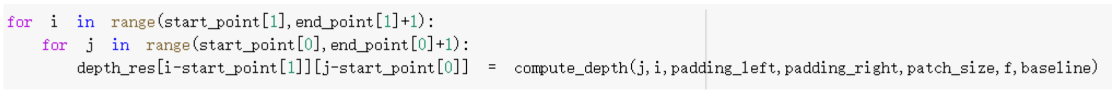


```
f1 = open("/content/drive/My Drive/A4_files/000020.txt")
res = f1.readline()
res = res.split()
res
```

```
['Car', '685.05', '181.43', '804.68', '258.21']
```

```
# upper left point
start_point=(int(float(res[1])),int(float(res[2])))
# lower right point
end_point=(int(float(res[3])),int(float(res[4])))

print(start_point)
print(end_point)
```

```
(685, 181)
(804, 258)
```

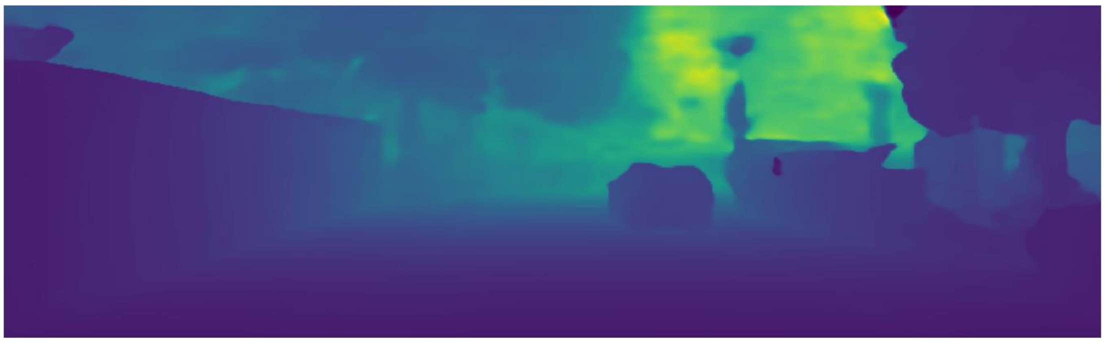


```
for i in range(start_point[1],end_point[1]+1):
    for j in range(start_point[0],end_point[0]+1):
        depth_res[i-start_point[1]][j-start_point[0]] = compute_depth(j,i,padding_left,padding_right,patch_size,f,baseline)
```

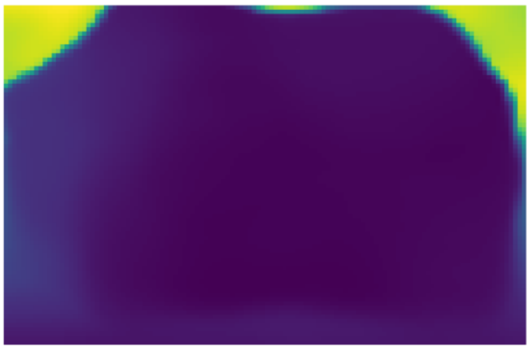
(b)

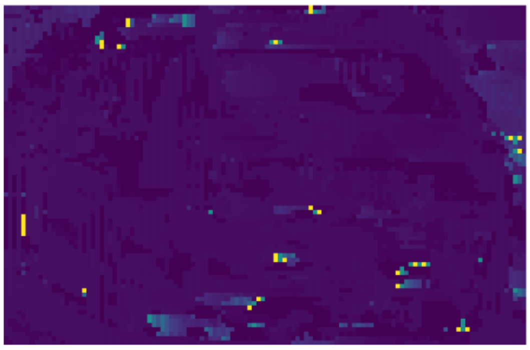
I use the GANet model from Stereo Evaluation 2015.
Code coming from: https://github.com/feihuzhang/GANet

The output depth for the whole image shown below:



Then I select only the depth for pixels in the given bounding box and normalize it, compared with result from previous question. Two images shown below:





In general, two depth images provide the similar result. We can see that the upper left and right corner of both images show the different depth than elsewhere in the image (visually the part that car located). Noted that, deeper the depth, the brighter the color.

However, in terms of **quality** and **speed**, the machine learning model provides the **better** quality and **faster** computation time. We can easily compare the quality from two depth images, the image output from ML model almost has no noise and provides very clear depth difference. In terms of speed, the ML model generate output immediately, but our function needs wait a few minutes.

Code:

```
import  models.GANet_deep  as  gan
import  torch
import  torch.nn  as  nn
from  torchvision.transforms  import  transforms  as  T
from  PIL  import  Image
```

```
model  =  gan.GANet()
model  =  nn.DataParallel(model)
model.load_state_dict(torch.load("kitti2015_final.pth")["state_dict"])

left  =  Image.open("/content/drive/My  Drive/A4_files/000020_left.jpg")
right  =  Image.open("/content/drive/My  Drive/A4_files/000020_right.jpg")

transform  =  T.Compose([
        T.Resize((240,624)),
        T.ToTensor()
])

left  =  transform(left)
right  =  transform(right)
disp0,  disp1,  disp2  =  model(torch.tensor(left).unsqueeze(0),  torch.tensor(right).unsqueeze(0))
```

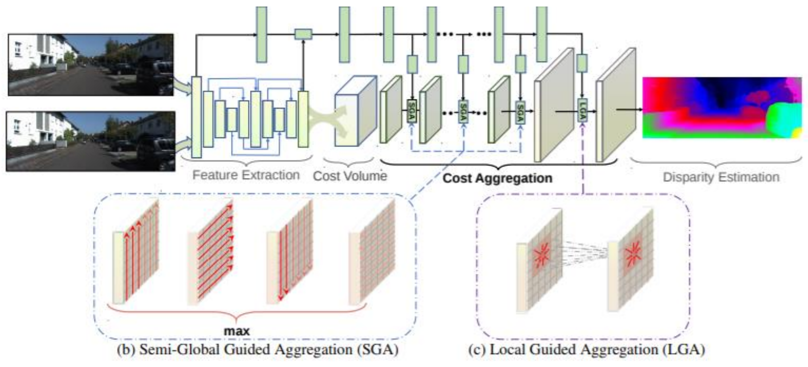
```
disp  =  disp2.cpu().detach().numpy().squeeze()
disp  =  cv2.resize(disp, (1242,375))
depth  =  (f*baseline)/disp
```

(c)

GANet is implemented by pytorch and the model consists 4 parts: feature extraction, cost volume, cost aggregation and disparity estimation. The workflow can be described as: Firstly, the 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.

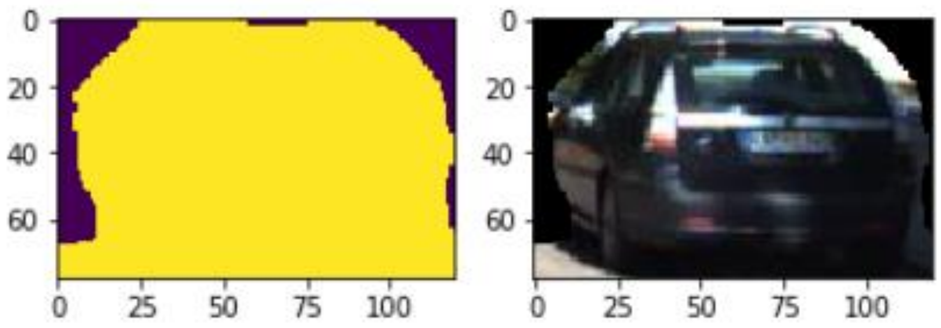
The guidance subnet generates the weight matrices for the guided cost aggregations which includes Semi-Global Guided Aggregation (SGA) layer and Local Guided Aggregation (LGA) layer. SGA layers semi-globally aggregate the cost volume in four directions and can be repeated several times in the neural network model to obtain better cost aggregation effects. The LGA layer is used before the disparity regression and locally refines the 4D cost volume for several times. This layer learns several guided filters to refine the matching cost and aid in the recovery of thin structure and object edges.

GANet Architecture shown below:

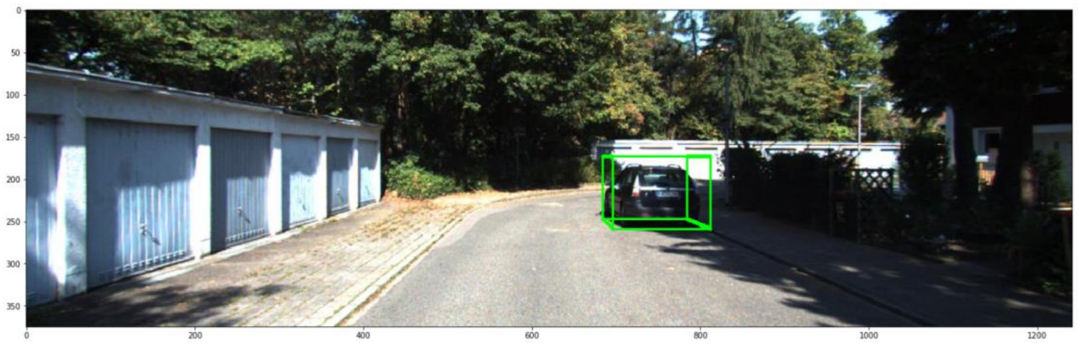

Feature Extraction
Cost Volume
Cost Aggregation
Disparity Estimation
SGA
LGA
max
(b) Semi-Global Guided Aggregation (SGA)
(c) Local Guided Aggregation (LGA)

(d)

2D box segmentation:

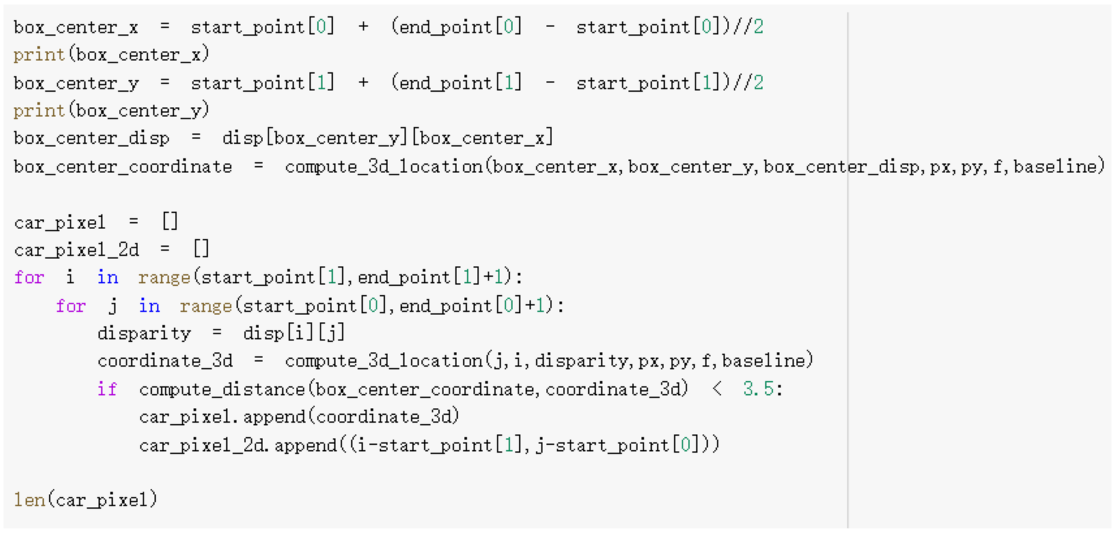


3D box:



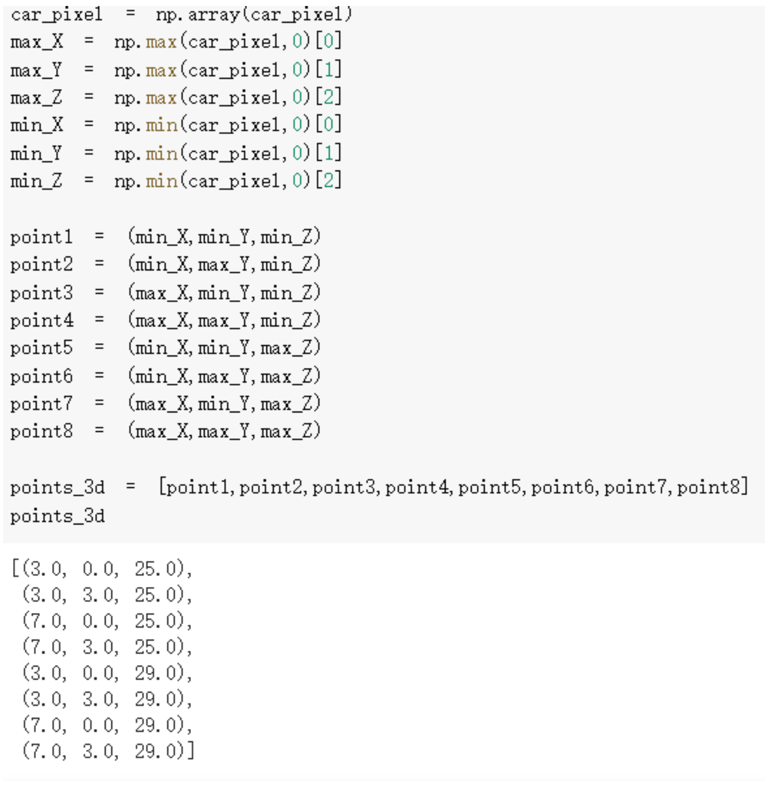
I set the classification threshold for distance **3.5**. Since the box center pixel is visually the back of the car and the type of the car is Hatchback, so I assume its length around **3.5m**. Then, I found all pixels' 3D location inside bounding box, and calculated their distance to box center pixel's 3D location, if less than **3.5**, that pixel belongs to the car.

Code:



```
box_center_x = start_point[0] + (end_point[0] - start_point[0])//2
print(box_center_x)
box_center_y = start_point[1] + (end_point[1] - start_point[1])//2
print(box_center_y)
box_center_disp = disp[box_center_y][box_center_x]
box_center_coordinate = compute_3d_location(box_center_x,box_center_y,box_center_disp,px,py,f,baseline)

car_pixel = []
car_pixel_2d = []
for i in range(start_point[1],end_point[1]+1):
    for j in range(start_point[0],end_point[0]+1):
        disparity = disp[i][j]
        coordinate_3d = compute_3d_location(j,i,disparity,px,py,f,baseline)
        if compute_distance(box_center_coordinate,coordinate_3d) < 3.5:
            car_pixel.append(coordinate_3d)
            car_pixel_2d.append((i-start_point[1],j-start_point[0]))

len(car_pixel)
```

```
744
219
7242
```



```
car_pixel = np.array(car_pixel)
max_X = np.max(car_pixel,0)[0]
max_Y = np.max(car_pixel,0)[1]
max_Z = np.max(car_pixel,0)[2]
min_X = np.min(car_pixel,0)[0]
min_Y = np.min(car_pixel,0)[1]
min_Z = np.min(car_pixel,0)[2]

point1 = (min_X,min_Y,min_Z)
point2 = (min_X,max_Y,min_Z)
point3 = (max_X,min_Y,min_Z)
point4 = (max_X,max_Y,min_Z)
point5 = (min_X,min_Y,max_Z)
point6 = (min_X,max_Y,max_Z)
point7 = (max_X,min_Y,max_Z)
point8 = (max_X,max_Y,max_Z)

points_3d = [point1,point2,point3,point4,point5,point6,point7,point8]
points_3d
```

```
[(3.0, 0.0, 25.0),
 (3.0, 3.0, 25.0),
 (7.0, 0.0, 25.0),
 (7.0, 3.0, 25.0),
 (3.0, 0.0, 29.0),
 (3.0, 3.0, 29.0),
 (7.0, 0.0, 29.0),
 (7.0, 3.0, 29.0)]
```

```
def compute_2d_location(coordinate,f,px,py):
    x = int(round(coordinate[0]*f/coordinate[2] + px))
    y = int(round(coordinate[1]*f/coordinate[2] + py))
    return x,y

points_2d = []
for i in range(8):
    points_2d.append(compute_2d_location(points_3d[i],f,px,py))
print(points_2d)
```

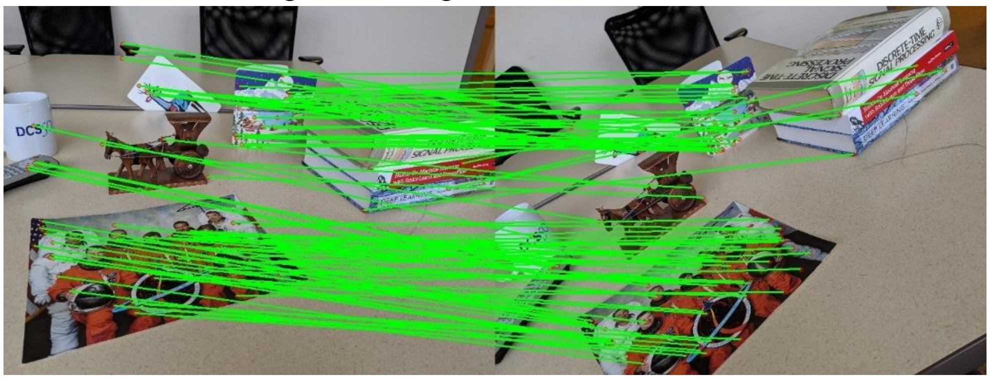
```
[(696, 173), (696, 259), (812, 173), (812, 259), (684, 173), (684, 247), (784, 173), (784, 247)]
```

```
# draw segmentation
mask_w = end_point[0]+1 - start_point[0]
mask_h = end_point[1]+1 - start_point[1]
mask = np.zeros((mask_h,mask_w))
for i in range(len(car_pixel_2d)):
    y = car_pixel_2d[i][0]
    x = car_pixel_2d[i][1]
    mask[y][x]=1
plt.imshow(mask)
```

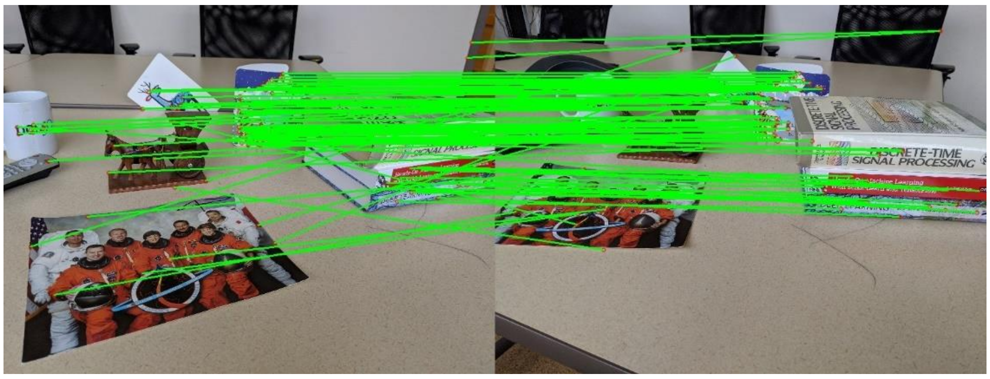
```
def draw_3D(img,points_2d):
    cv2.rectangle(img, points_2d[1], points_2d[2], (0,255,0),2)
    cv2.rectangle(img, points_2d[5], points_2d[6], (0,255,0),2)
    cv2.line(img,points_2d[0], points_2d[4],(0,255,0),3)
    cv2.line(img,points_2d[1],points_2d[5],(0,255,0),3)
    cv2.line(img,points_2d[2],points_2d[6],(0,255,0),3)
    cv2.line(img,points_2d[3],points_2d[7],(0,255,0),3)
    plt.figure(figsize=(24,64))
    plt.imshow(img)
left = cv2.imread('/content/drive/My Drive/A4_files/000020_left.jpg')
left = cv2.cvtColor(left, cv2.COLOR_BGR2RGB)
draw_3D(left,points_2d)
```

3.(a)

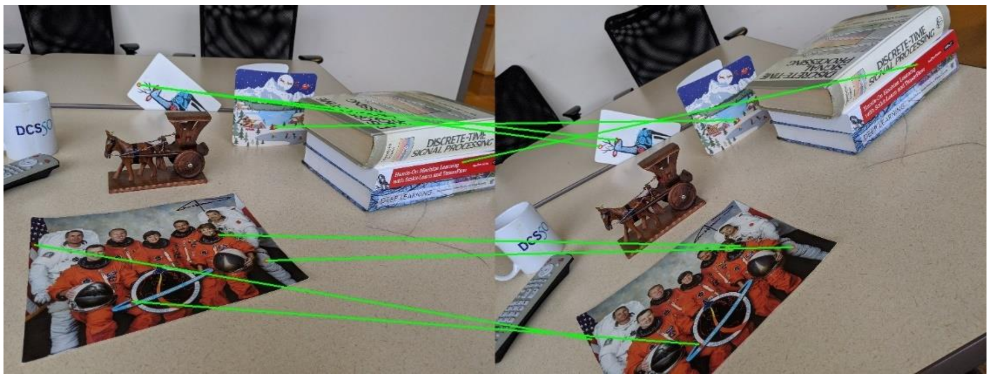
Matches between image1 and image2:



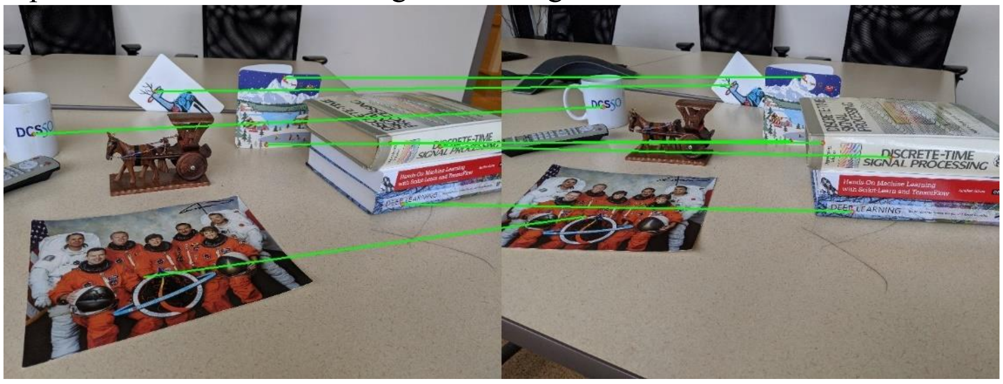
Matches between image1 and image3:



8 points matches between image1 and imag2:



8 points matches between image1 and imag3:



Code:

```
def point_distance(pt,pts):
    distance_list = []
    for i in range(len(pts)):
        target = pts[i]
        distance = ((pt[0]-target[0])**2+(pt[1]-target[1])**2)**(1/2)
        distance_list.append(distance)
    if len(pts)!=0 and min(distance_list) < 60:
        return True
    return False
```

```python
def  sift_matching(img1,img2,threshold,norm_type):
    sift  =  cv2.xfeatures2d.SIFT_create()
    kp1,  des1  =  sift.detectAndCompute(img1,None)
    kp2,  des2  =  sift.detectAndCompute(img2,None)
    matched_pair  =  {}
    for  i  in  range(len(kp1)):
        difference  =  des1[i,:]  -  des2
        differnece_abs  =  np.abs(difference)
        if  norm_type  ==  1:  #L1  norm
            res  =  np.sum(differnece_abs,axis=1)
        elif  norm_type  ==  2:  #L2  norm
            res_sum  =  np.sum(differnece_abs**2,axis=1)
            res  =  res_sum**(1/2)
        elif  norm_type  ==  3:  #L3  norm
            res_sum  =  np.sum(differnece_abs**3,axis=1)
            res  =  res_sum**(1/3)
        index_sorted  =  np.argsort(res)[:2]
        smallest  =  index_sorted[0]
        second_smallest  =  index_sorted[1]
        ratio  =  res[smallest]  /  res[second_smallest]
        if  ratio  <  threshold:
            matched_pair[ratio]  =  (kp1[i],kp2[smallest])

    return  matched_pair
def  find_kp(result,num_match):
    points_in_img1  =  []
    points_coordinate_img1  =  []
    points_in_img2  =  []
    points_coordinate_img2  =  []
    count  =  0
    for  key  in  sorted(result):
        if  count  ==  num_match:
            break
        img1_pt  =  (int(result[key][0].pt[0]),int(result[key][0].pt[1]))
        img2_pt  =  (int(result[key][1].pt[0]),int(result[key][1].pt[1]))
        if  img1_pt  not  in  points_coordinate_img1:
            #  commmented  code  for  making  matching  points  distributed  over
            #  the  images  and  remove  outliers  between  image1  and  image3
            #  if  point_distance(img1_pt,points_coordinate_img1):
            #      continue
            #  if  img1_pt[0]  >  750:
            #      continue
            points_in_img1.append(result[key][0])
            points_coordinate_img1.append(img1_pt)
            points_in_img2.append(result[key][1])
            points_coordinate_img2.append(img2_pt)
            count  +=  1
    return  points_in_img1,points_coordinate_img1,points_in_img2,points_coordinate_img2
```

```python
def  draw_lines(new_res,new_res2,points_coordinate_img1,points_coordinate_img2,num_match):
    offset  =  new_res.shape[1]
    combine_image  =  np.concatenate((new_res,new_res2),axis  =  1)
    for  i  in  range(num_match):
        coordinate1_x  =  int(points_coordinate_img1[i][0])
        coordinate1_y  =  int(points_coordinate_img1[i][1])
        coordinate1  =  (coordinate1_x,coordinate1_y)
        coordinate2_x  =  int(points_coordinate_img2[i][0]  +  offset)
        coordinate2_y  =  int(points_coordinate_img2[i][1])
        coordinate2  =  (coordinate2_x,  coordinate2_y)
        cv2.line(combine_image,coordinate1,coordinate2,(0,255,0),2)
    cv2.imwrite("/content/drive/My  Drive/combine1_6.jpg",combine_image)
```

```
img1 = cv2.imread('/content/drive/My Drive/A4_files/I1.jpg')
img1 = cv2.resize(img1, (w_new, h_new), interpolation=cv2.INTER_CUBIC)
img2 = cv2.imread('/content/drive/My Drive/A4_files/I2.jpg')
img2 = cv2.resize(img2, (w_new, h_new), interpolation=cv2.INTER_CUBIC)
result = sift_matching(img1,img2,0.8,2)
points_in_img1,points_coordinate_img1,points_in_img2,points_coordinate_img2 = find_kp(result,8)
```

```
res1 = cv2.drawKeypoints(img1,points_in_img1,None,color=(0,0,255))
```

```
res2 = cv2.drawKeypoints(img2,points_in_img2,None,color=(0,0,255))
```

```
draw_lines(res1,res2,points_coordinate_img1,points_coordinate_img2,8)
```

Provided the code for best 8 points SIFT matches between image1 and image2, to generate other output images, just change according parameter.

(b)

```
F12 = compute_fundamental_matrix(np.int64(points_coordinate_img1),np.int64(points_coordinate_img2))
```

```
[[ 3.17752531e-07 -6.06694868e-07 -4.58488037e-04]
 [ 7.06874835e-07 -1.67485606e-07 -4.74372906e-03]
 [-2.09670332e-03  4.64629040e-03  9.99975651e-01]]
```

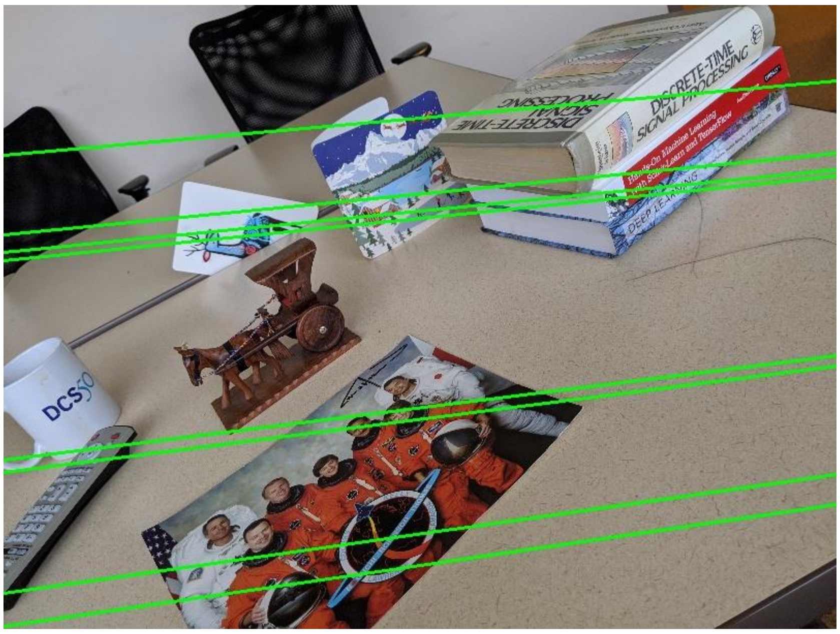
```
F13 = compute_fundamental_matrix(np.int64(points_coordinate_img1_1),np.int64(points_coordinate_img3))
```

```
[[ 4.99112174e-06 -7.37597339e-05 -8.76870928e-03]
 [-2.26540573e-05 -2.36791385e-06  1.70975687e-01]
 [ 6.12405689e-03 -1.20669244e-01  9.77799493e-01]]
```

Code:

```
def compute_fundamental_matrix(points_coordinate_img1,points_coordinate_img2):
    a = np.zeros((8,9))
    for i in range(8):
        x_l = points_coordinate_img1[i][0]
        y_l = points_coordinate_img1[i][1]
        x_r = points_coordinate_img2[i][0]
        y_r = points_coordinate_img2[i][1]
        item1 = x_r*x_l
        item2 = x_r*y_l
        item3 = x_r
        item4 = y_r*x_l
        item5 = y_r*y_l
        item6 = y_r
        item7 = x_l
        item8 = y_l
        item9 = 1
        a[i,:] = np.array([item1,item2,item3,item4,item5,item6,item7,item8,item9])
    u, s, v = np.linalg.svd(a)
    F1 = np.reshape(v[-1],(3,3))
    u1,s1,v1 = np.linalg.svd(F1)
    diagonal_value = np.array([s1[0],s1[1],0])
    new_s1 = np.diag(diagonal_value)
    F_res = u1.dot(new_s1).dot(v1)
    print(F_res)
    return F_res
```

(c)



Since image 2 only performs rotation from image1, the epipolar lines must be parallel (epipoles does not exist).

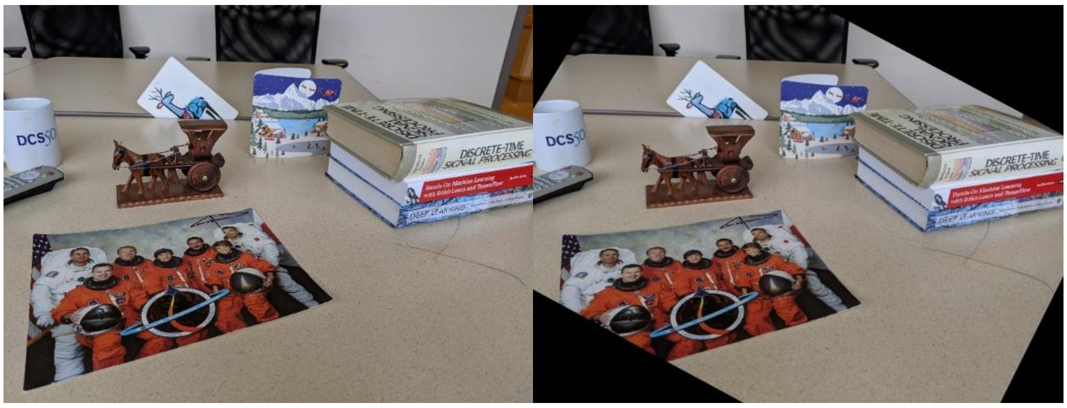
Code:

```
def  compute_epipolar_lines(points,F,img):
    lines  =  []
    for  i  in  range(8):
        point_x  =  points[i][0]
        point_y  =  points[i][1]
        point  =  np.array([point_x,point_y,1])
        line  =  F.dot(point)
        lines.append(line)
    h,w,c  =  img.shape
    for  i  in  range(8):
        point_1_x  =  0
        point_2_x  =  w
        a  =  lines[i][0]
        b  =  lines[i][1]
        c  =  lines[i][2]
        point_1_y  =  (-a*point_1_x-c)/b
        point_2_y  =  (-a*point_2_x-c)/b
        point1  =  (int(point_1_x),  int(point_1_y))
        point2  =  (int(point_2_x),  int(point_2_y))
        cv2.line(img,point1,point2,(0,255,0),2)
        cv2.imwrite("/content/drive/My  Drive/output11.jpg",img)
    plt.imshow(img)
```

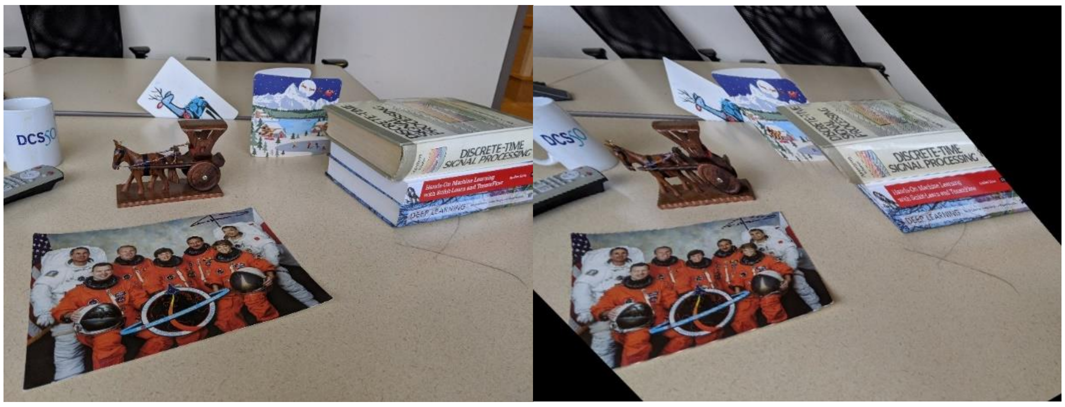
```
img2  =  cv2.imread('/content/drive/My  Drive/A4_files/I2.jpg')
img2  =  cv2.resize(img2,  (w_new,  h_new),  interpolation=cv2.INTER_CUBIC)
compute_epipolar_lines(points_coordinate_img1,F12,img2)
```

(d)

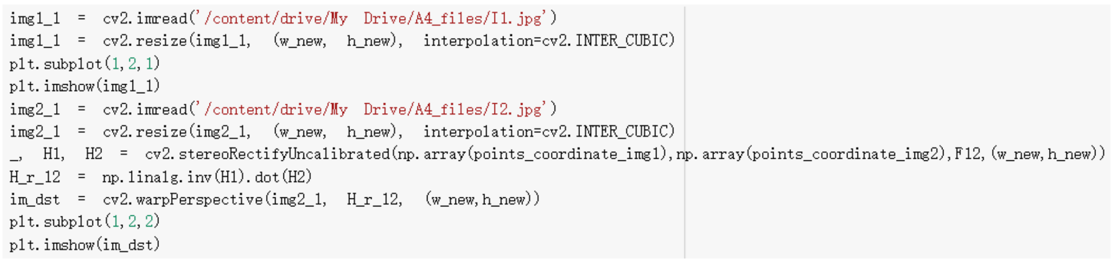
Rectify image2:



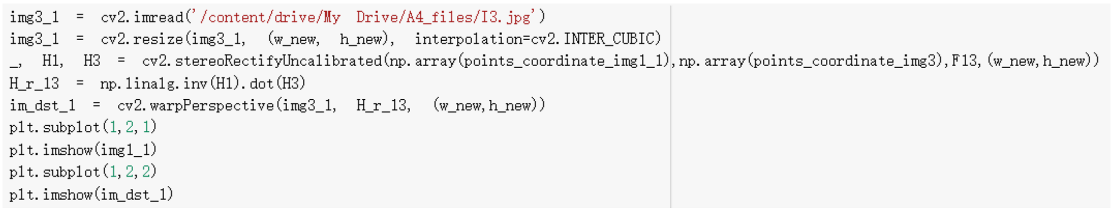
Rectify image3:



Code:



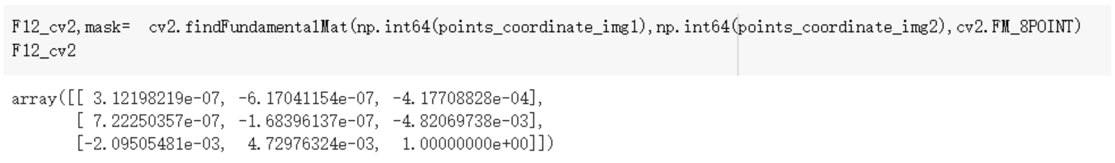
```
img1_1  =  cv2.imread('/content/drive/My  Drive/A4_files/I1.jpg')
img1_1  =  cv2.resize(img1_1,   (w_new,   h_new),   interpolation=cv2.INTER_CUBIC)
plt.subplot(1,2,1)
plt.imshow(img1_1)
img2_1  =  cv2.imread('/content/drive/My  Drive/A4_files/I2.jpg')
img2_1  =  cv2.resize(img2_1,   (w_new,   h_new),   interpolation=cv2.INTER_CUBIC)
_,  H1,  H2  =  cv2.stereoRectifyUncalibrated(np.array(points_coordinate_img1),np.array(points_coordinate_img2),F12,(w_new,h_new))
H_r_12  =  np.linalg.inv(H1).dot(H2)
im_dst  =  cv2.warpPerspective(img2_1,  H_r_12,   (w_new,h_new))
plt.subplot(1,2,2)
plt.imshow(im_dst)
```



```
img3_1  =  cv2.imread('/content/drive/My  Drive/A4_files/I3.jpg')
img3_1  =  cv2.resize(img3_1,   (w_new,   h_new),   interpolation=cv2.INTER_CUBIC)
_,  H1,  H3  =  cv2.stereoRectifyUncalibrated(np.array(points_coordinate_img1_1),np.array(points_coordinate_img3),F13,(w_new,h_new))
H_r_13  =  np.linalg.inv(H1).dot(H3)
im_dst_1  =  cv2.warpPerspective(img3_1,  H_r_13,   (w_new,h_new))
plt.subplot(1,2,1)
plt.imshow(img1_1)
plt.subplot(1,2,2)
plt.imshow(im_dst_1)
```

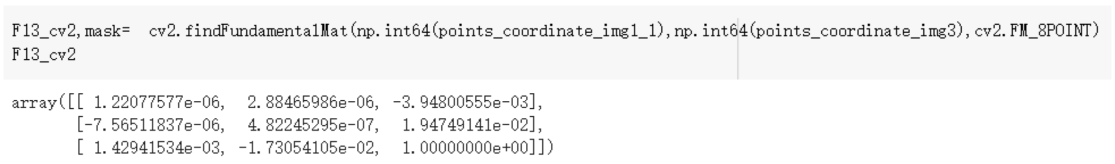
(e)

Fundamental matrix using OpenCV:



```
F12_cv2,mask=  cv2.findFundamentalMat(np.int64(points_coordinate_img1),np.int64(points_coordinate_img2),cv2.FM_8POINT)
F12_cv2
```

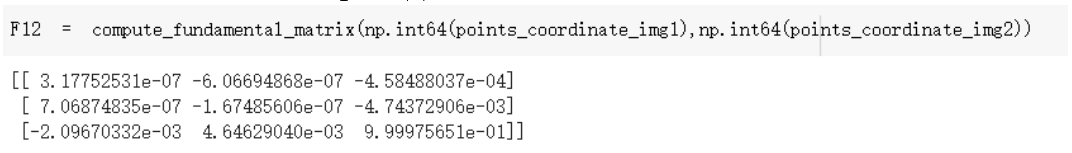
```
array([[ 3.12198219e-07, -6.17041154e-07, -4.17708828e-04],
       [ 7.22250357e-07, -1.68396137e-07, -4.82069738e-03],
       [-2.09505481e-03,  4.72976324e-03,  1.00000000e+00]])
```



```
F13_cv2,mask=  cv2.findFundamentalMat(np.int64(points_coordinate_img1_1),np.int64(points_coordinate_img3),cv2.FM_8POINT)
F13_cv2
```

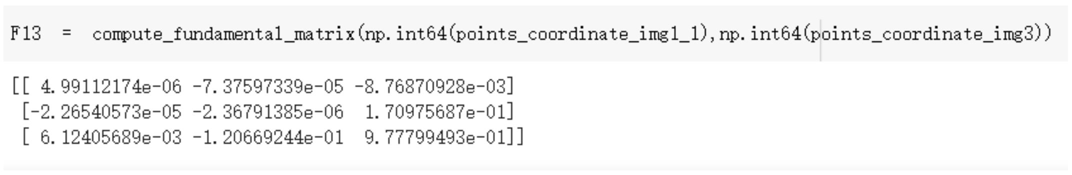
```
array([[ 1.22077577e-06,  2.88465986e-06, -3.94800555e-03],
       [-7.56511837e-06,  4.82245295e-07,  1.94749141e-02],
       [ 1.42941534e-03, -1.73054105e-02,  1.00000000e+00]])
```

Fundamental matrix from part (c)



```
F12  =  compute_fundamental_matrix(np.int64(points_coordinate_img1),np.int64(points_coordinate_img2))
```

```
[[ 3.17752531e-07 -6.06694868e-07 -4.58488037e-04]
 [ 7.06874835e-07 -1.67485606e-07 -4.74372906e-03]
 [-2.09670332e-03  4.64629040e-03  9.99975651e-01]]
```



```
F13  =  compute_fundamental_matrix(np.int64(points_coordinate_img1_1),np.int64(points_coordinate_img3))
```

```
[[ 4.99112174e-06 -7.37597339e-05 -8.76870928e-03]
 [-2.26540573e-05 -2.36791385e-06  1.70975687e-01]
 [ 6.12405689e-03 -1.20669244e-01  9.77799493e-01]]
```

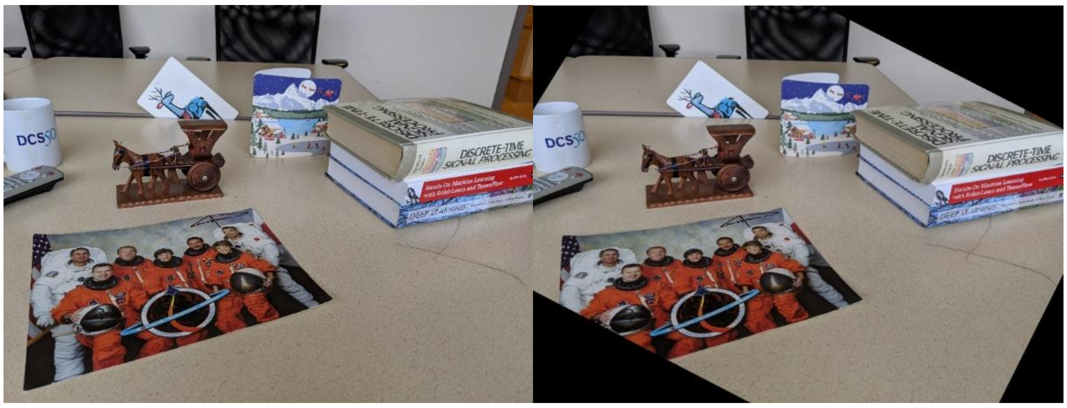
The results are very similar between **F12** and **F'12** since I still use the 8-point algorithm to compute fundamental matrix from OpenCV library. The specific calculation may be different, and there might be some difference due to rounding. However, the results look different between **F13** and **F'13**, I think it may be caused by following reasons:

1. Same reason as before, the specific calculation using in OpenCV may be different or it may perform some extra optimization.
2. Since fundamental matrix compose of essential matrix which contains rotation and translation matrix, same transformation can be reached by different ways e.g. change the order of rotation and translation or approach from different direction.

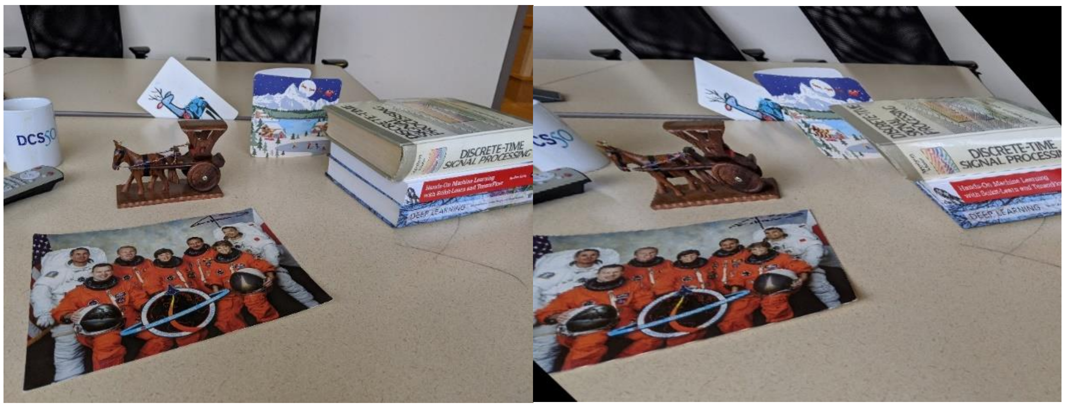
Even though fundamental matrixes are different, we can still reach the similar output. In (f), we see that different **F** get similar results.

(f)

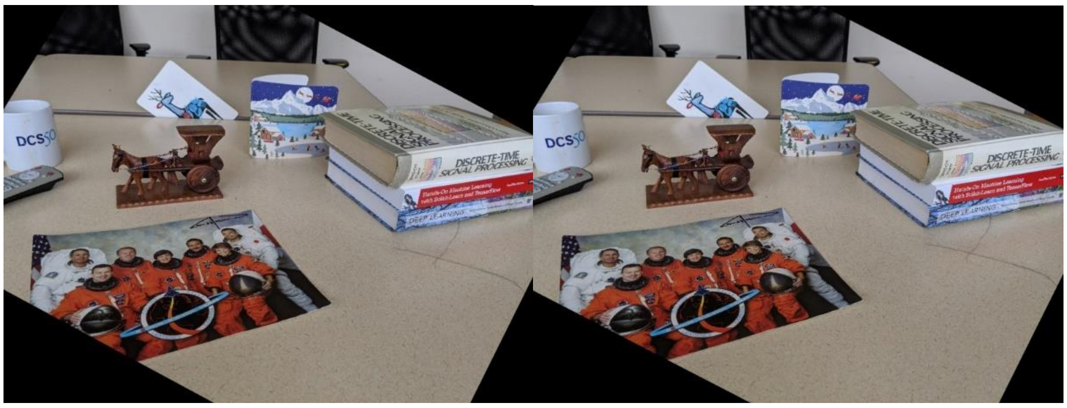
Rectify image2 using **F'12** from OpenCV:



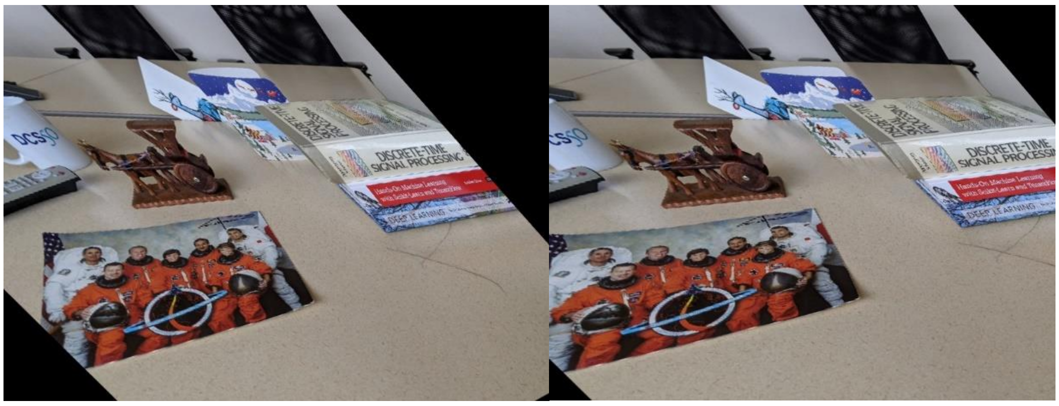
Rectify image3 using **F'13** from OpenCV:



Compare rectified image 2 with part(d):



Compare rectified image 3 with part(d):



The results are quite **similar** between rectified image using fundamental matrix from OpenCV and our function. Not surprisingly, rectified image 2 visually looks almost the same due to very similar **F**. For rectified image 3, even though it looks a tiny difference, we can still visually see that two rectified images have the same point of view. More specific, all the matching points stand on their same horizontal line. Thus, we can conclude that OpenCV and our function reach the same result.

Code:

```
img1_2 = cv2.imread('/content/drive/My Drive/A4_files/I1.jpg')
img1_2 = cv2.resize(img1_2, (w_new, h_new), interpolation=cv2.INTER_CUBIC)
plt.subplot(1,2,1)
plt.imshow(img1_1)
img2_2 = cv2.imread('/content/drive/My Drive/A4_files/I2.jpg')
img2_2 = cv2.resize(img2_2, (w_new, h_new), interpolation=cv2.INTER_CUBIC)
_, H1, H2 = cv2.stereoRectifyUncalibrated(np.array(points_coordinate_img1),np.array(points_coordinate_img2),F12_cv2,(w_new,h_new))
H_r_12 = np.linalg.inv(H1).dot(H2)
im_dst = cv2.warpPerspective(img2_2, H_r_12, (w_new,h_new))
plt.subplot(1,2,2)
plt.imshow(im_dst)
```

```
img3_2 = cv2.imread('/content/drive/My Drive/A4_files/I3.jpg')
img3_2 = cv2.resize(img3_2, (w_new, h_new), interpolation=cv2.INTER_CUBIC)
_, H1, H3 = cv2.stereoRectifyUncalibrated(np.array(points_coordinate_img1_1),np.array(points_coordinate_img3),F13_cv2,(w_new,h_new))
H_r_13 = np.linalg.inv(H1).dot(H3)
im_dst_1 = cv2.warpPerspective(img3_2, H_r_13, (w_new,h_new))
plt.subplot(1,2,1)
plt.imshow(img1_1)
plt.subplot(1,2,2)
plt.imshow(im_dst_1)
```