Fan Vincent Ka Chun 1002563380

Question 1

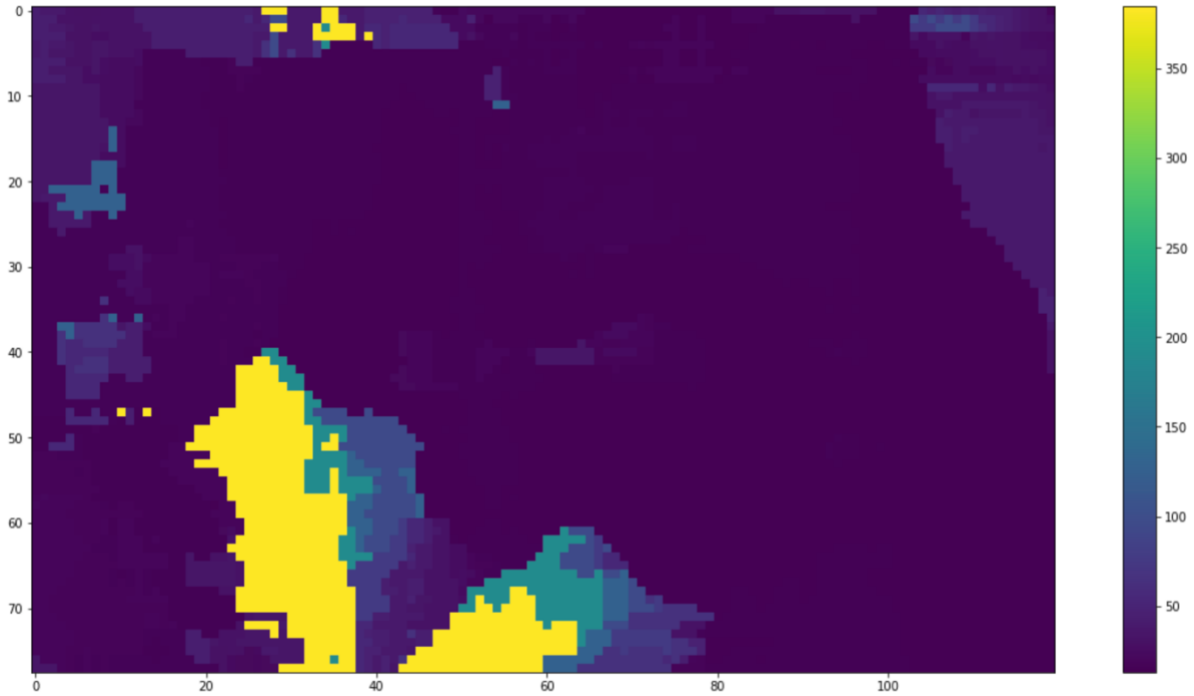
SSD is the sum of squared difference between each corresponding pixel in the given patches.
NC is the normalized correlation between each corresponding pixel in the given patches.

Note that SSD is easier to implement and lower computer cost, however if the light setting changed in between taking the pictures, the difference of pixel intensities will directly affect the SSD score since it measures the minimum intensity differences. On the other hand, NC measures the max pixel intensity “similarity” between patches, so with different light settings, NC can still recognize the most similar patch since the correlation will remain about the same in different lightings. However Nc is more computational costly.

Fan Vincent Ka Chun 1002563380

Question 2

(a): I implemented the SSD algorithm and picked a patch size of 17 pixels, I start from the corresponding pixel on the right image and go left of the same row of pixels since we know it cannot be on the right. Also, I only sampled 30 pixels since after drawing the matching box on the right image, I noticed it's not very far away and to reduce computation cost I set it to sample 30 patches along the scan line. The resulting depth map is at follows:



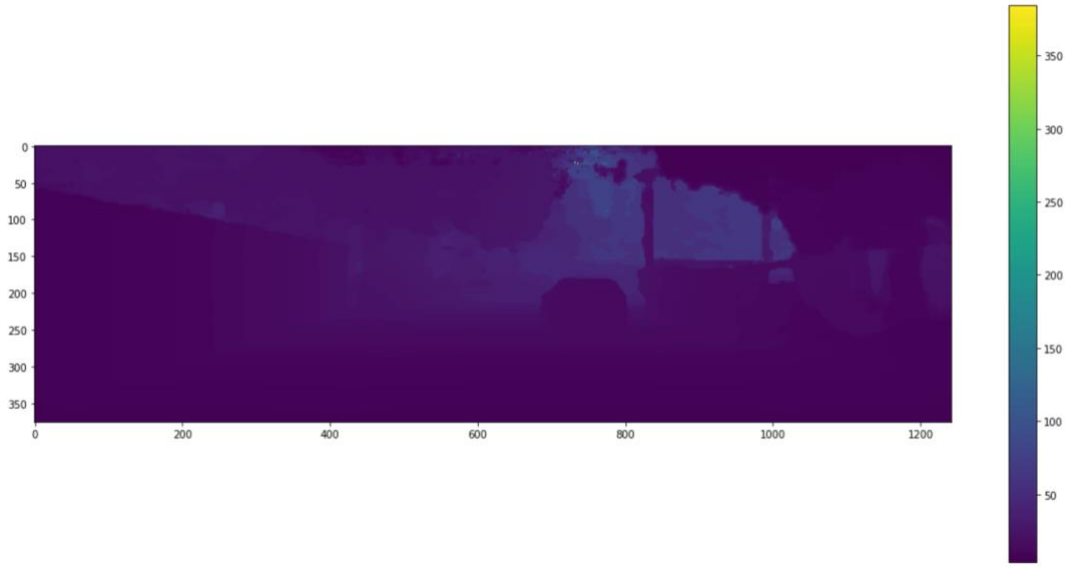
The results are not very good since I have multiple matches where the first pixel is the most similar thus disparity becomes 0, therefore I set it to 1 if it happens. Also, it does not really resemble the shape of the car.

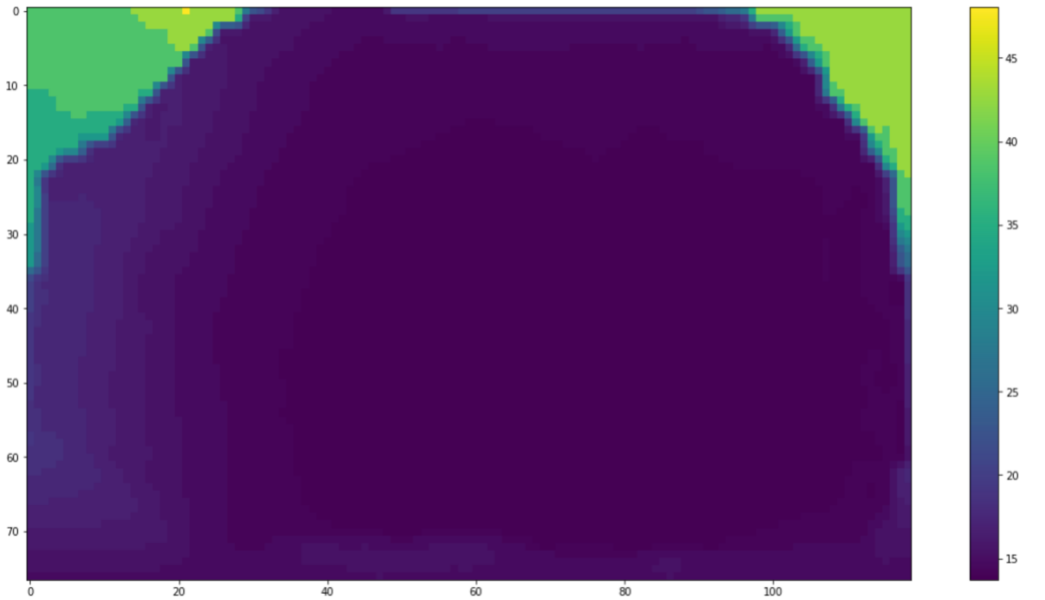
```
def SSD(left, right, point, length, patchsize=7):
    x, y = point
    upper, lower = int(np.ceil(patchsize/2)), int(np.floor(patchsize/2))
    result = []
    target = left[y-lower:y+upper,x-lower:x+upper]
    for i in range(length):
        patch = right[y-lower:y+upper,(x-i)-lower:(x-i)+upper]
        result.append(np.sum((target - patch)**2))
    xmin = np.argmin(result)
    return x-xmin

def depth(xmin,xmax,ymin,ymax,left,right,length,patchsize):
    Z = np.zeros((ymax-ymin+1, xmax-xmin+1))
    for i in range(xmin, xmax+1):
        for j in range(ymin, ymax+1):
            disparity = i-SSD(left,right,(i,j),length,patchsize)
            if disparity == 0:
                disparity = 1
            Z[j-ymin][i-xmin]= f*baseline/(disparity)
    return Z
```

Fan Vincent Ka Chun 1002563380

(b): I choose the HD3 model (https://github.com/ucbdrive/hd3) and ran the testing script with the added file lists. After obtaining the result disparity map (upper), I used the formula: focal length * baseline / disparity which is each pixel's intensity and I obtained the depth map of the entire left image (middle). Finally, I extracted only the depth of bounding box (bottom).





Fan Vincent Ka Chun 1002563380

(c): In my understanding, HD3 uses Pytorch backend, its structure can be broken down into an encoder (DLAUpEncoder) and decoder (HDADecoder). In the encoder, it has 3 Sequential layers consisting of convolution and ReLu, then it has 5 Tree blocks, finally it has a DLAUp blocks which consist of more convolution and ReLu layers. In the decoder, it has PreActBlock with mostly consist of identity, convTranspose layers which is like upsampling layers followed by with some convolution and ReLu layers. Then within the last bit of decoder, it has a Context block.

(d): I took the depth of bounding box and convert all 2D points into 3D points with the following function:

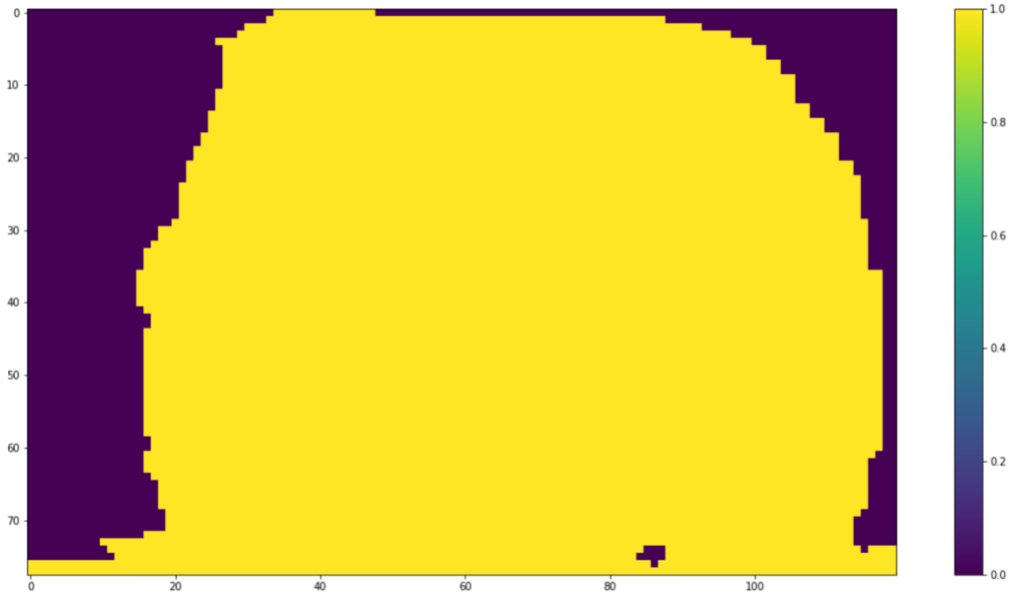
```
def convert2World(point, z):
    x = ((point[0] - px)*z)/f
    y = ((point[1] - py)*z)/f
return x,y,z
```

```
worldcoor = []
imgcoor = []
for i in range(y1,y2+1):
    row = []
    for j in range(x1,x2+1):
        row.append(convert2World((j,i),hd3_depth[i][j]))
        imgcoor.append([j,i])
    worldcoor.append(row)
worldcoor = np.array(worldcoor)
imgcoor = np.array(imgcoor)
```

Then I calculated the L2 norm between each point to the centre point to determine which pixels belong to the car.

```
def l2norm(point, center):
return np.linalg.norm(center-point)
```

I set the threshold to be within 2 units of L2 norm which gives me the following results where the yellow pixels indicate pixels belong to the car:



We can see that the results aren't very accurate however it does distinguish the background on the sides at least.

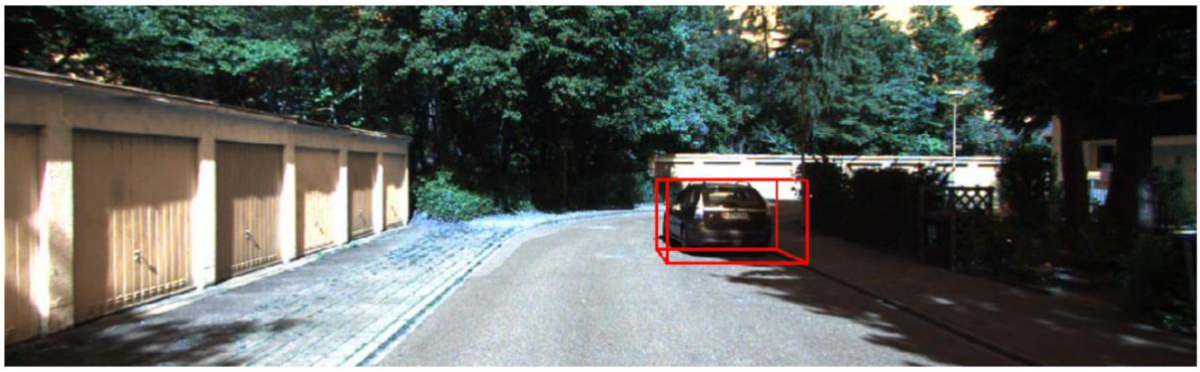
Then using this binary mask, I extracted the max and min of X,Y,Z directions by multiplying the 3D coordinates with the truth mask and ignore the 0s for finding min.

```
X,Y,Z = worldcoor[:,:,0], worldcoor[:,:,1], worldcoor[:,:,2]
car_X, car_Y, car_Z = X* filtered.astype(int),Y* filtered.astype(int),Z*
filtered.astype(int)
Xmin, Xmax = np.min(car_X[np.nonzero(car_X)]),
np.max(car_X[np.nonzero(car_X)])
Ymin, Ymax = np.min(car_Y[np.nonzero(car_Y)]),
np.max(car_Y[np.nonzero(car_Y)])
Zmin, Zmax = np.min(car_Z[np.nonzero(car_Z)]),
np.max(car_Z[np.nonzero(car_Z)])
eightCorners = np.array([[Xmin, Ymin, Zmin], [Xmin, Ymax, Zmin], [Xmax,
Ymax, Zmin], [Xmax, Ymin, Zmin],
                         [Xmin, Ymin, Zmax], [Xmin, Ymax, Zmax], [Xmax,
Ymax, Zmax], [Xmax, Ymin, Zmax]])
```

Now, using the follow code, I transform the 8 corner coordinates back into 2D.
def convert2Img(point):

```
    x,y,z = point
    x_img = ((f*x)/z) + px
    y_img = ((f*y)/z) + py
    return x_img, y_img
```

Here are the results that I got:

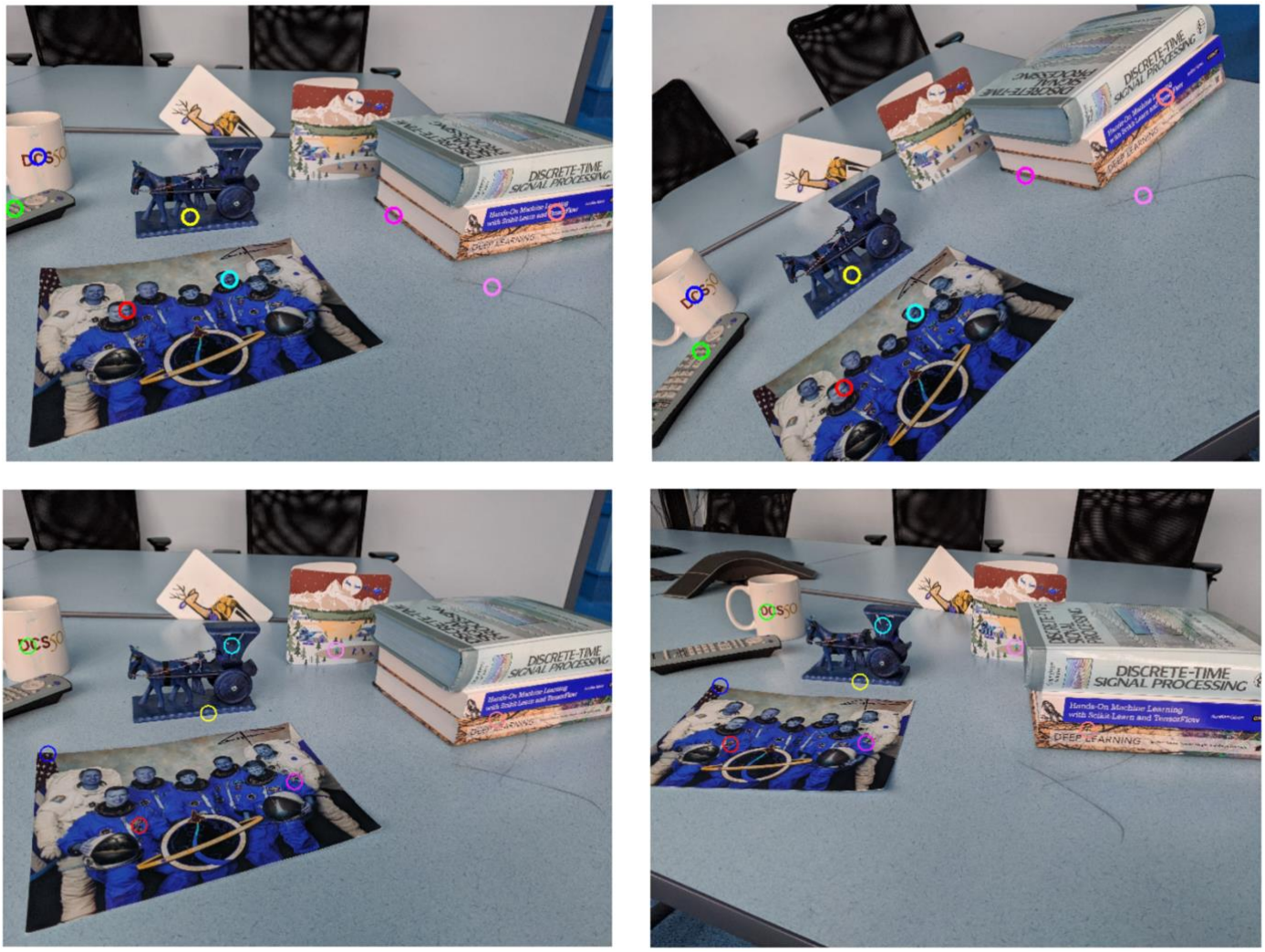


The results are surprisingly not bad since the edges are align with the side walk and car.

Fan Vincent Ka Chun 1002563380

Question 3

(a): Using the cv2 SIFT and BFMatcher, I manually picked the matched points and got the following results:



Same colours mean matching.

(b): Here is my 8 point algorithm implementation:

```
def eightpoints(lpoints, rpoints):
    results = []
    for i in range(len(rpoints)):
        row = [lpoints[i][0]*rpoints[i][0], lpoints[i][0]*rpoints[i][1],
lpoints[i][0],
               lpoints[i][1]*rpoints[i][0], lpoints[i][1]*rpoints[i][1],
lpoints[i][1],
               rpoints[i][0], rpoints[i][1], 1.]
        results.append(row)
    results = np.array(results)
    U,S,V = np.linalg.svd(results)
    F = V[-1].reshape(3,3)
    U,S,V = np.linalg.svd(F)
    S[2] = 0
    F = np.dot(U, np.dot(np.diag(S), V))
return F/F[2,2]
```

Fan Vincent Ka Chun 1002563380

The Fundamental matrix F_{12} and F_{13} I got are following:

```
[[ 6.77596582e-08 -1.25020460e-07  1.28190369e-03]
 [ 1.39207047e-07  7.48632893e-08 -4.34643056e-04]
 [-1.42089146e-03 -3.92617413e-04  1.00000000e+00]]

[[ 8.75995489e-08  6.71637208e-07 -6.76076670e-04]
 [-5.06405322e-07  5.51072707e-07 -1.20422994e-03]
 [ 3.72559204e-04 -5.04332606e-04  1.00000000e+00]]
```

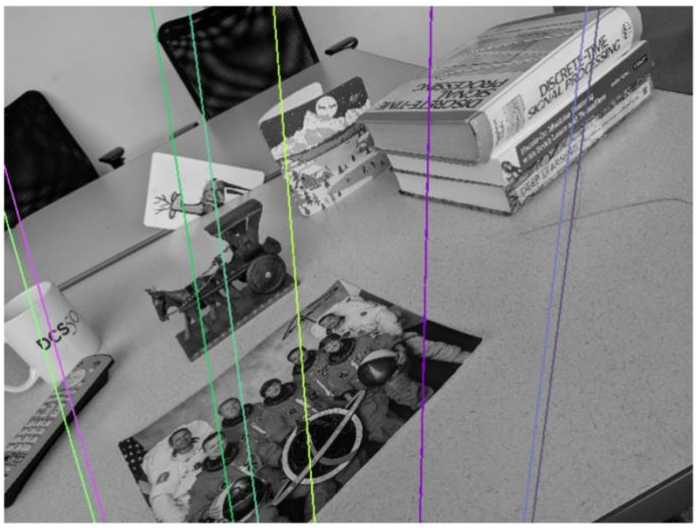
(c): I used the following cv2 functions to get the epipolar lines:

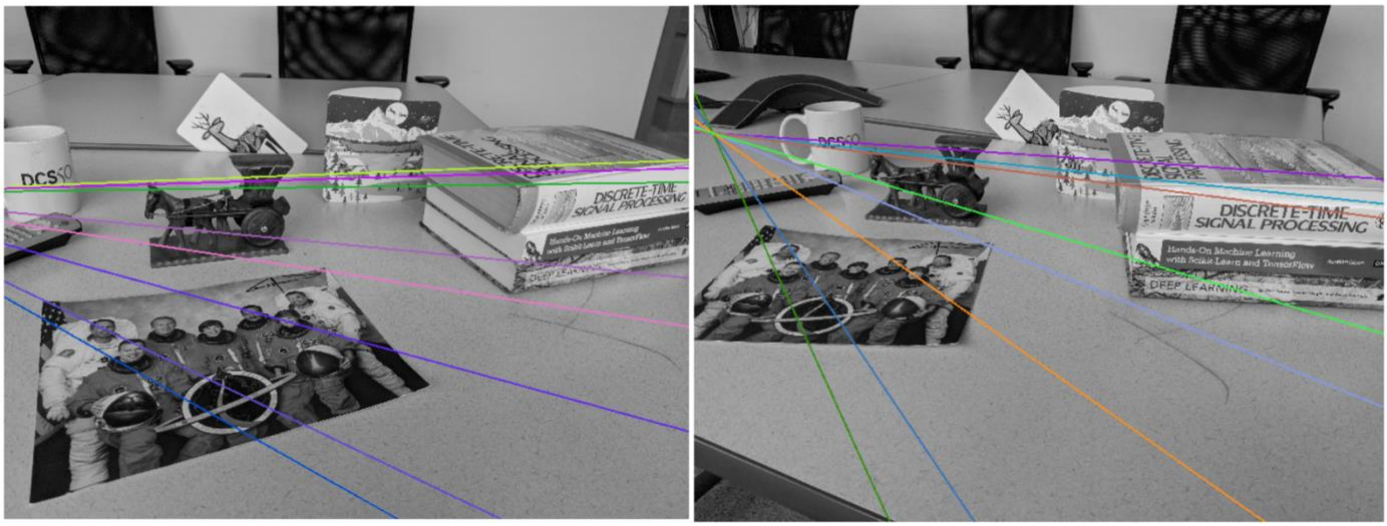
```
def drawlines(img1,img2,lines,pts1,pts2):
    r,c = img1.shape
    img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
    img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img = cv2.line(img1, (x0,y0), (x1,y1), color,10)
return img1
lines1 = cv2.computeCorrespondEpilines(lpoints1.reshape(-1,1,2),
2,fmatrix1)
lines1 = lines1.reshape(-1,3)
img1 = drawlines(I1,I2,lines1,rpoints1,lpoints1)

lines2 = cv2.computeCorrespondEpilines(rpoints1.reshape(-1,1,2),
1,fmatrix1)
lines2 = lines2.reshape(-1,3)
img2 = drawlines(I2,I1,lines2,lpoints1,rpoints1)

fig3, graph =plt.subplots(figsize=(20,10))
graph.imshow(img1)
graph.axis('off')
fig4, graph =plt.subplots(figsize=(20,10))
graph.imshow(img2)
graph.axis('off')
```

Here are the following results that I got:





The first pair is I_1 and I_2. The second pair is I_1 and I_3

(d): Here is the cv2 code I used to get the results:

```
success, H1, H2 = cv2.stereoRectifyUncalibrated(rpoints1, lpoints1,
fmatrix1, (I1.shape[1],I1.shape[0]))
rect_I1 = cv2.warpPerspective(I1, H1, (I1.shape[1],I1.shape[0]))
rect_I2 = cv2.warpPerspective(I2, H2, (I2.shape[1],I2.shape[0]))
fig3, graph =plt.subplots(figsize=(20,10))
graph.imshow(rect_I1,cmap="gray")
graph.axis('off')
fig4, graph =plt.subplots(figsize=(20,10))
graph.imshow(rect_I2,cmap="gray")
graph.axis('off')
```

Here are the resulting rectified images I got:

This is the pair of I_1 and I_2. As we can see the images are lined up almost identical.

This is the pair of I_1 and I_3. As we can see the images are not that well lined up. However, the astronaut image are somewhat lined up.

(e): Using cv2.findFundamentalMat(r1,l1,cv2. FM_8POINT) to compute F_{12}' and F_{13}':

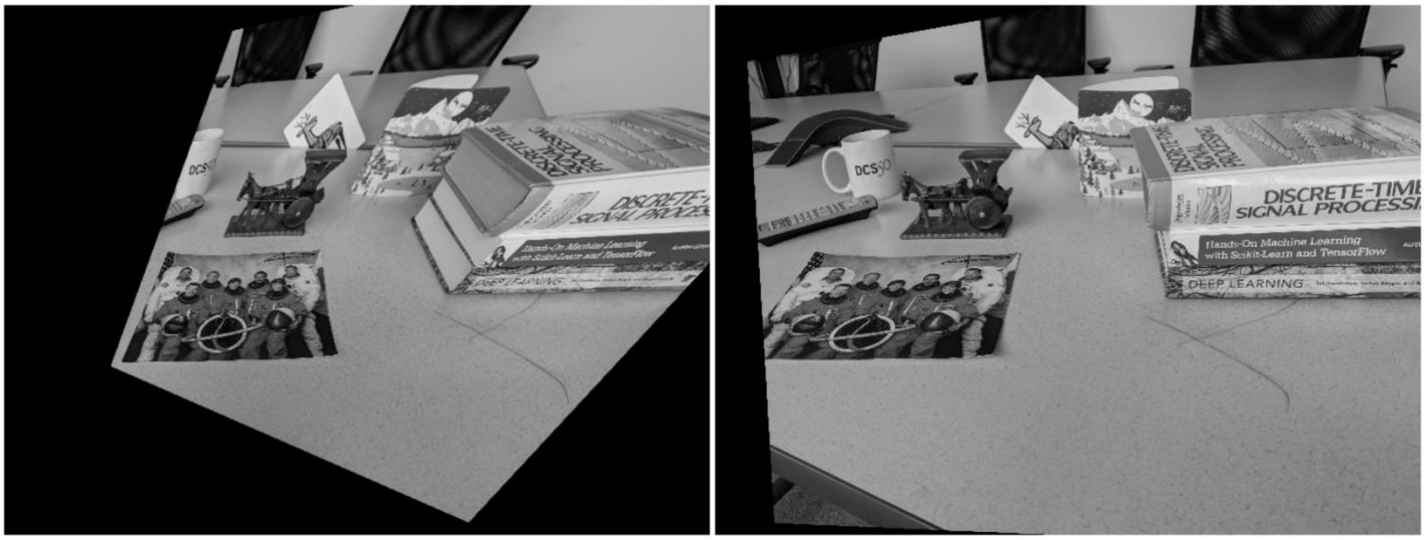
```
[[ 8.69264528e-08 -1.81916661e-07 -1.37090954e-03]
 [ 1.52967202e-07  6.68051266e-08 -1.44330349e-03]
 [ 3.56800295e-04  1.89919008e-03  1.00000000e+00]]

[[ 1.00706322e-07  7.09065781e-07 -1.07114278e-03]
 [-1.14055993e-06  8.07291460e-07  4.91386456e-03]
 [ 6.76648593e-04 -5.27314542e-03  1.00000000e+00]]
```

Note that the matrices for F_{12} are similar in terms of the signs and magnitude. However for F_{13} the signs have some difference but the magnitude is still roughly the same.

Since I used more than 8 points for this part, the cv2 functions can consider more matches thus it should be more accurate.

(f): Using the Fundamental Matrices found in (e), and using the same technique in (d).



The first pair is I_1 and I_2, the second pair is I_1 and I_3.

For pair I_1 and I_2, note that the Homograph matrices found in (f) does not rotate both image just the result in (d), however both result is both close to identical.

For pair I_1 and I_3, note that the results in (f) is better than the result in (d), since the result showed more of the entire image and the "X" on the table is close to identical. However, both results are not as good compared to the I_1 and I_2 pair.