CSC420 A4

Ziqi Chen

November 2019

1 Stereo Matching Costs

I believe Normalise Correlation will be much better in this case than Sum of Squared Difference, since different exposure time means different amount of light came in, the picture taken with longer exposure time would be brighter than the other one, in this case SSD will be very high due to change of brightness, therefore SSD won't be very accurate in this case, whereas normalised correlation will average over all pixels of both picture, and thus cost will be less sensitive to brightness and it will be more accurate.

2 Stereo Matching Implementation

a. For the patch size I choose 14, and for sampling method I used down sampling, matching cost function I used sum of squared difference.
Here is how sliding window scan is implemented:

```
def scan_all(scan_size=50, downsample=True, img1=left_grey, img2=right_grey, patch_size=patch_size
    ):
    if downsample:
        img1 = down_sample(img1)
        img2 = down_sample(img2)
    height1, width1 = img1.shape[0], img1.shape[1]
    height2, width2 = img2.shape[0], img2.shape[1]
    assert height1 == height2 and width1 == width2

    # Setup progress bar
    ite_num = height1 * width1 * 2 * min(scan_size, width2)
    bar = progressbar.ProgressBar(maxval=ite_num,
                                  widgets=[progressbar.Bar('=', '[', ']'), ' ', progressbar.
                                      Percentage()])
    bar.start()
    count = 0
    # Start sliding window
    diff = np.zeros((height1, width1))
    scores = np.zeros((height1, width1))
    for y1 in range(height1):
        for x1 in range(width1):
            x1i = max(0, x1 - patch_size)
            toleft = abs(x1 - x1i)
            x1d = min(width1, x1 + patch_size)
            toright = abs(x1 - x1d)
            y1i = max(0, y1 - patch_size)
            toup = abs(y1 - y1i)
            y1d = max(height1, y1 + patch_size)
            todown = abs(y1 - y1d)

            patch1 = img1[y1i:y1d, x1i:x1d]
            y2 = y1

            best_score = None
```

```
            best_ind = None
            for x2 in range(max(x1 - scan_size, toleft), min(x1 + scan_size + 1, width2 - toright))
                :
                x2i = x2 - toleft
                x2d = x2 + toright
                y2i = y2 - toup
                y2d = y2 + todown
                patch2 = img2[y2i:y2d, x2i:x2d]
                score = ssd(patch1, patch2)
                if best_score is None or score <= best_score:
                    best_score = score
                    best_ind = x2
                count += 1
                bar.update(count)
            scores[y1, x1] = abs(best_score)
            diff[y1, x1] = abs(best_ind - x1)
    # This is a simple idea I implemented to avoid diff = 0, normalize ssd cost to a range betwen
        0 to 1, which means even if there is no change in pixel, but if the cost funciton is not 0
        then it means this pixel still moved a bit.
    scores = np.interp(scores, (scores.min(), scores.max()), (0, 1.0))
    diff += scores
    if downsample:
        diff = up_sample(diff)
    return diff
```

Here is how depth is calculated:

```
def calculate_depth(diff, f=f, T=baseline):
    depths = (f * T) / (diff)
    return depths
```

Some util functions, functions too general such as loading and saving images are not included:

```
def ssd(patch1, patch2):
    diff = patch1 - patch2
    ssd = np.sum(diff**2)
    return ssd


def nc(patch1, patch2):
    a = np.sum(patch1 * patch2)
    b = np.sum(patch1 ** 2) * np.sum(patch2 ** 2)
    c = a * 1./b
    return c

def down_sample(img, factor=2, ite=1):
    for i in range(ite):
        height, width = img.shape[0], img.shape[1]
        img = cv.pyrDown(img, dstsize=(width // factor, height // factor))
    return img


def up_sample(img, factor=2, ite=1):
    for i in range(ite):
        height, width = img.shape[0], img.shape[1]
        img = cv.pyrUp(img, dstsize=(width * factor, height * factor))
    return img
```

Depth for the patch given:

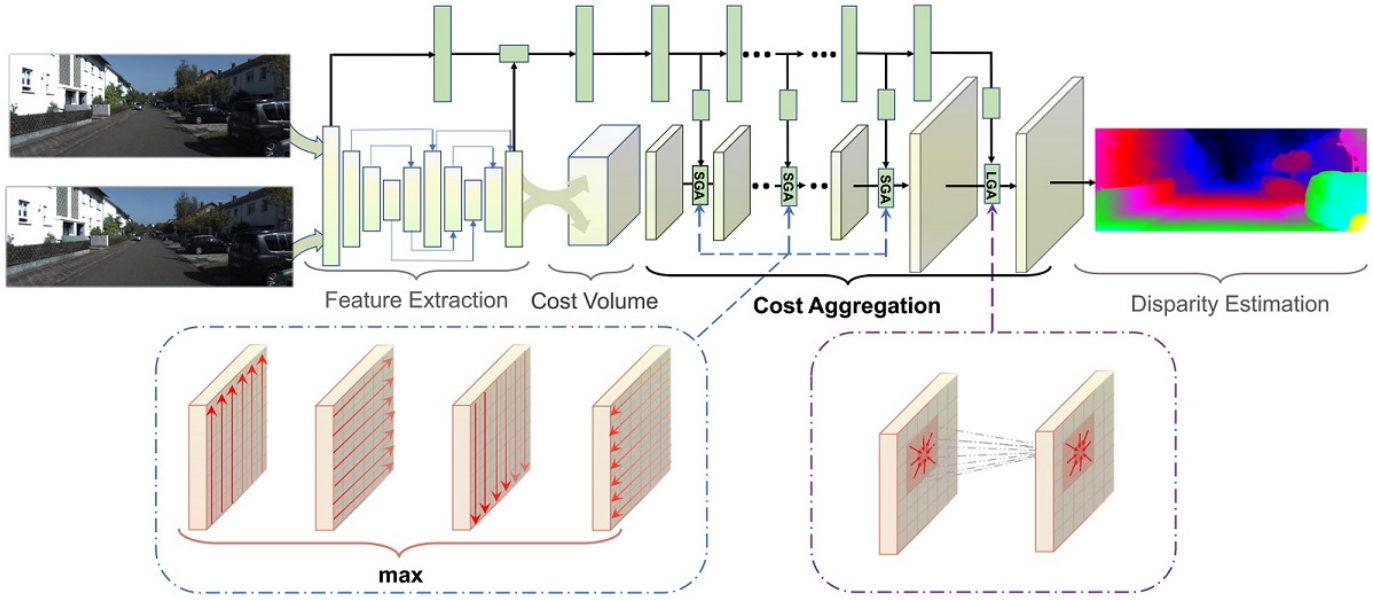
Depth for whole image:

There are some outliers at the middle part, some of the pixels are recognized to be far away here.

b. The model I choose is Guided Aggregation Net, code is from: `https://github.com/feihuzhang/GANet`
Here is the result:

The quality is much better using their model, and the speed is approximately the same.

c. Work flow for guided aggregation net:
First some terms explained based on my understanding:
Semi-Global Matching (SGM): according to my understanding SGM is to compare the cost along multiple directions and sum the cost over same direction. I think this is to find all pixels on same depth level.
Semi-Global Aggregation (SGA), introduced in Zhang et al's paper "GA-Net: Guided Aggregation Net for End-to-end Stereo Matching": A differentiable approximation of SGM
Local Guided Aggregation (LGA): Since down sampling and up sampling are widely used in the model, a LGA layer is added and aid in to recover blurred edges in image back to thin edge.
With all these terms explained, here is the model diagram:



First stack the two images together and pass to the model, 10 layers of CNN are used for feature extraction, then extracted features are fed into a 4D cost volume, then it enters a cost aggregation block with 3 SGA layers and 1 LGA layer, each SGA layer is calculated from 4 directions and the maximum one is preserved, after 3 SGA layers it's fed into LGA for thin edge recovery. The green part on top is guidance subnet that is used to generate weights for SGA and LGA.

d. Below is the function to find the corresponding coordinates of top left corner and bottom right corner on right image:

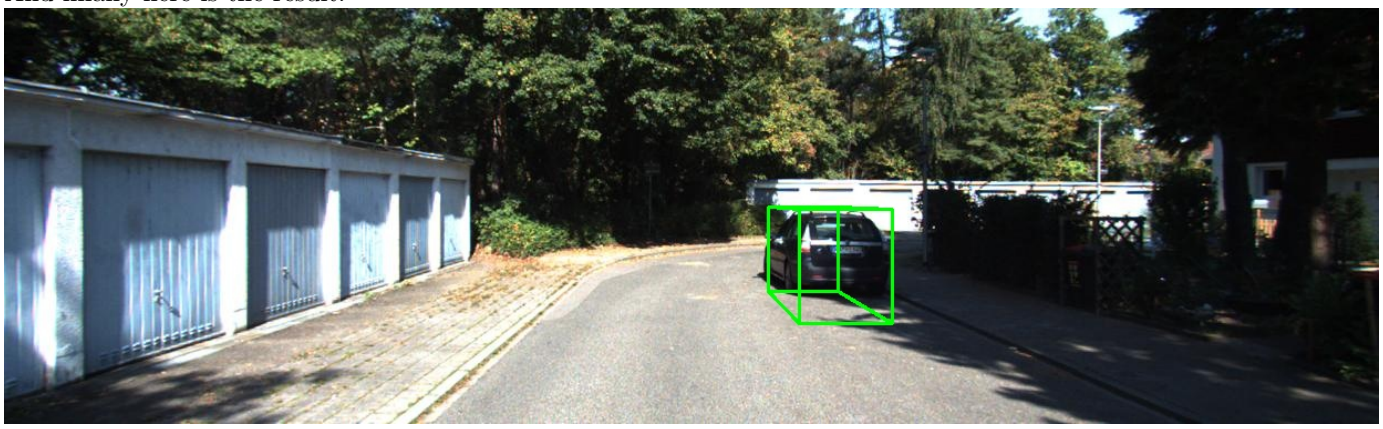
```
def find_cor_pixels(left_x1=x1l, left_y1=y1l, left_x2=x2l, left_y2=y2l,
                    img1=left_color, img2=right_color, patch_size=14):
    height, width = img1.shape[0], img1.shape[1]
    right_y1 = left_y1
    right_y2 = left_y2
    target_patch1 = img1[left_y1 - patch_size: left_y1 +
                         patch_size + 1, left_x1 - patch_size: left_x1 + patch_size + 1]
    target_patch2 = img1[left_y2 - patch_size: left_y2 +
                         patch_size + 1, left_x2 - patch_size: left_x2 + patch_size + 1]
    best_score1 = None
    best_ind1 = None
    best_score2 = None
    best_ind2 = None
    for x in range(patch_size, width - patch_size):
        source_patch1 = img2[right_y1 - patch_size: right_y1 +
                             patch_size + 1, x - patch_size: x + patch_size + 1]
        source_patch2 = img2[right_y2 - patch_size: right_y2 +
                             patch_size + 1, x - patch_size: x + patch_size + 1]
        score1 = ssd(source_patch1, target_patch1)
        if best_score1 is None or score1 < best_score1:
            best_score1 = score1
            best_ind1 = x
        score2 = ssd(source_patch2, target_patch2)
        if best_score2 is None or score2 < best_score2:
            best_score2 = score2
            best_ind2 = x
    x1, x2 = best_ind1 - patch_size, best_ind2
    y1, y2 = right_y1, right_y2
    return x1, y1, x2, y2
```

Then here is the code for finding coordinates and draw a 3d box:

```
def calculate_3d(depth, img, x1l, y1l, x2l, y2l, x1r, y1r, x2r, y2r, threshold = 10, px = px, py =
     py, f = f):
    Z = depth[y1l:y2l + 1, x1l:x2l + 1]
    x = np.ones((Z.shape[0], 1)) * np.arange(x1l, x2l + 1)
    y = (np.ones((Z.shape[1], 1)) * np.arange(y1l, y2l + 1)).T
    X = Z * (x - px) / f
    Y = Z * (y - py) / f

    X0 = X[math.floor(X.shape[0]/2), math.floor(X.shape[1]/2)]
    Y0 = Y[math.floor(Y.shape[0]/2), math.floor(Y.shape[1]/2)]
    Z0 = Z[math.floor(Z.shape[0]/2), math.floor(Z.shape[1]/2)]

    dis = ((X - X0) ** 2 + (Y - Y0)** 2 + (Z - Z0) ** 2) ** 0.5
    Xs, Ys, Zs = X[dis < threshold], Y[dis < threshold], Z[dis < threshold]
    xmin, xmax = np.min(Xs), np.max(Xs)
    ymin, ymax = np.min(Ys), np.max(Ys)
    zmin, zmax = np.min(Zs), np.max(Zs)

    # Plane thats closer to camera (zmin)
    # Plane on the left (xmin)
    # plane on the top (ymin)
    # Plane that's far from camera (zmax)
    # Plane on the right (xmax)
    # plane on the bottom (ymax)
    x1, y1 = convert_3d_to_2d(xmin, zmin, px), convert_3d_to_2d(ymin, zmin, py)
    x2, y2 = convert_3d_to_2d(xmax, zmin, px), convert_3d_to_2d(ymin, zmin, py)
    x3, y3 = convert_3d_to_2d(xmin, zmin, px), convert_3d_to_2d(ymax, zmin, py)
    x4, y4 = convert_3d_to_2d(xmax, zmin, px), convert_3d_to_2d(ymax, zmin, py)

    x5, y5 = convert_3d_to_2d(xmin, zmax, px), convert_3d_to_2d(ymin, zmax, py)
    x6, y6 = convert_3d_to_2d(xmax, zmax, px), convert_3d_to_2d(ymin, zmax, py)
    x7, y7 = convert_3d_to_2d(xmin, zmax, px), convert_3d_to_2d(ymax, zmax, py)
    x8, y8 = convert_3d_to_2d(xmax, zmax, px), convert_3d_to_2d(ymax, zmax, py)

    p1, p2, p3, p4, p5, p6, p7, p8 = (x1, y1), (x2, y2), (x3, y3), (x4, y4), (x5, y5),(x6, y6), (
        x7, y7), (x8, y8)
    stereo = draw_3d(left_color, p1, p2, p3,p4,p5,p6,p7,p8)
    return stereo

def draw_3d(img, p1, p2, p3, p4, p5, p6, p7, p8):
  temp = np.copy(img)
  cv.line(temp, p1, p2, (0, 255, 0), 2)
  cv.line(temp, p1, p3, (0, 255, 0), 2)
  cv.line(temp, p1, p5, (0, 255, 0), 2)
  cv.line(temp, p2, p4, (0, 255, 0), 2)
  cv.line(temp, p2, p6, (0, 255, 0), 2)
  cv.line(temp, p3, p4, (0, 255, 0), 2)
  cv.line(temp, p3, p7, (0, 255, 0), 2)
  cv.line(temp, p4, p8, (0, 255, 0), 2)
  cv.line(temp, p5, p6, (0, 255, 0), 2)
  cv.line(temp, p5, p7, (0, 255, 0), 2)
  cv.line(temp, p6, p8, (0, 255, 0), 2)
  cv.line(temp, p7, p8, (0, 255, 0), 2)
  return temp

def convert_3d_to_2d(coordinate, z, p, f = f):
  return int(f * coordinate / z + p)

left_color = load_color_image('./000020_left.jpg')
```

```
threshold = 10
box = calculate_3d(depth, left_color, x1l, y1l, x2l, y2l, x1r, y1r, x2r, y2r, threshold =
    threshold)
cv.imwrite(f"./basdfa.jpg", box)
```

And finally here is the result:



3 Fundamental Matrix

a. Visualization of (I1, I2) and (I1, I3) pair:

```
def sift(img, nfeatures=10, ct=0.04, et=5, sigma=20):
    print("Finding Key Points ... ")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    sift = cv2.xfeatures2d.SIFT_create(
        contrastThreshold=ct, edgeThreshold=et, sigma=sigma)
    kp, des = sift.detectAndCompute(gray, None)
    img = cv2.drawKeypoints(
        gray, kp, img, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    print("Done finding key points")
    return kp, des, img

def sift_color(img, nfeatures=100, ct=0.04, et=10, sigma=5):
    print("Finding Key Points ... ")
    sift = cv2.xfeatures2d.SIFT_create(
        contrastThreshold=ct, edgeThreshold=et, sigma=sigma)
    kp, des = sift.detectAndCompute(img, None)
    img = cv2.drawKeypoints(
        img, kp, img, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    print("Done finding key points")
    return kp, des, img

def distance(des1, des2):
    # L2 Norm
    dis = des1 - des2
    l2norm = np.linalg.norm(dis, 2)

    # L1 Norm
    l1norm = np.linalg.norm(dis, 1)

    # L3 Norm
    l3norm = np.linalg.norm(dis, 3)

    return l2norm

def match(kp1, des1, kp2, des2, max_n=12):
```

```
    lkp1 = len(kp1)
    lkp2 = len(kp2)
    point_list = []
    for i in range(lkp1):
        min_dis = None
        for j in range(lkp2):
            d1 = des1[i]
            d2 = des2[j]
            dis = distance(d1, d2)
            if (min_dis == None) or (dis < min_dis[2]):
                min_dis = (kp1[i], kp2[j], dis)
        point_list.append(min_dis)
    point_list.sort(key=itemgetter(2))
    return point_list[:min(len(point_list), max_n)]

def match_images(img1, img2, color=False):
    if color:
        kp1, des1, img1_sift = sift_color(img1)
        kp2, des2, img2_sift = sift_color(img2)
    else:
        kp1, des1, img1_sift = sift(img1)
        kp2, des2, img2_sift = sift(img2)
    match_points = match(kp1, des1, kp2, des2)
    con = concatenate(img1, img2)
    h_shift = img1_sift.shape[1]
    for point in match_points:
        r0 = int(point[0].pt[1])
        c0 = int(point[0].pt[0])
        r1 = int(point[1].pt[1])
        c1 = int(point[1].pt[0]) + h_shift
        new_img = draw_square(con, r0, c0, 10, np.array([0, 0, 255]))
        new_img = draw_square(con, r1, c1, 10, np.array([0, 0, 255]))
        rr, cc, val = line_aa(r0, c0, r1, c1)
        new_img[rr, cc] = np.array([0, 255, 0])
    return new_img
```

And result:

b.

```
def fundamental(points1, points2):
    width = points1.shape[1]
    m = np.zeros((width, 9))
    for n in range(width):
        xrn = points1[0, n]
        xln = points2[0, n]
        yrn = points1[1, n]
        yln = points2[1, n]
        m[n] = np.array([xrn * xln, xrn * yln, xrn, yrn *
                         xln, yrn * yln, yrn, xln, yln, 1])
    U, S, V = np.linalg.svd(m)
    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]

f_i1i2 = fundamental(pts1_i1i2, pts2_i1i2)
print(f_i1i2)
f_i1i3 = fundamental(pts1_i1i3, pts2_i1i3)
print(f_i1i3)

Output:
F for i1 and i2
[[-1.12968084e-07 -1.68069138e-07 -3.96386120e-11]
 [-5.54160300e-08 -8.24456264e-08 -4.19827724e-11]
 [-2.33128965e-11 -5.21139899e-11 1.00000000e+00]]

F for i1 and i3
[[ 1.29674358e-05 -1.95807021e-05 -6.16654364e-08]
 [ 3.76012366e-06 -5.67775019e-06 -7.24720849e-08]
 [-6.56700352e-08 -7.19518447e-08 1.00000000e+00]]
```

c. Used OpenCV's method to implement:

```
def epipolar(img1, img2):
    print("finding kps")
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)
    print("done")

    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)
    good = []
    pts1 = []
    pts2 = []
    # ratio test as per Lowe's paper
    for i, (m, n) in enumerate(matches):
        if m.distance < 0.8*n.distance:
            good.append(m)
            pts2.append(kp2[m.trainIdx].pt)
            pts1.append(kp1[m.queryIdx].pt)
    pts1 = np.int32(pts1)
    pts2 = np.int32(pts2)
    F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_LMEDS)
    # We select only inlier points
```

```
    pts1 = pts1[mask.ravel() == 1][:8]
    pts2 = pts2[mask.ravel() == 1][:8]

    # Find epilines corresponding to points in right image (second image) and
    # drawing its lines on left image
    lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1, 1, 2), 2, F)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines(img1, img2, lines1, pts1, pts2)

    # Find epilines corresponding to points in left image (first image) and
    # drawing its lines on right image
    lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1, 1, 2), 1, F)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines(img2, img1, lines2, pts2, pts1)

    return img5, img3
```

result for i1i2:

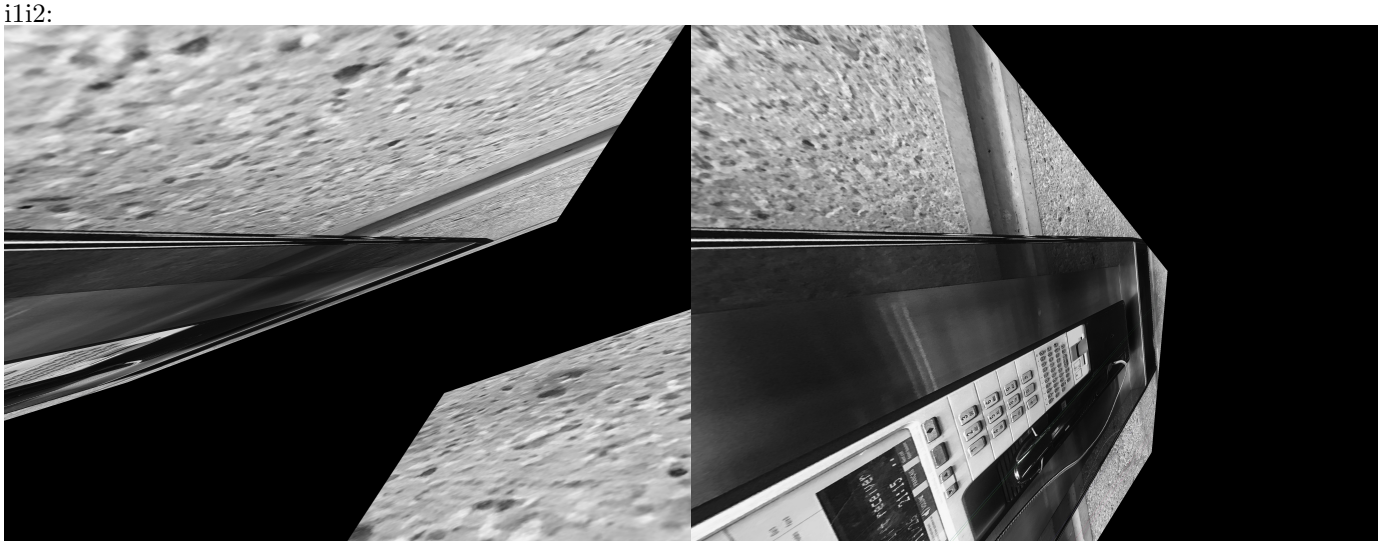
result for i1i3:

d.

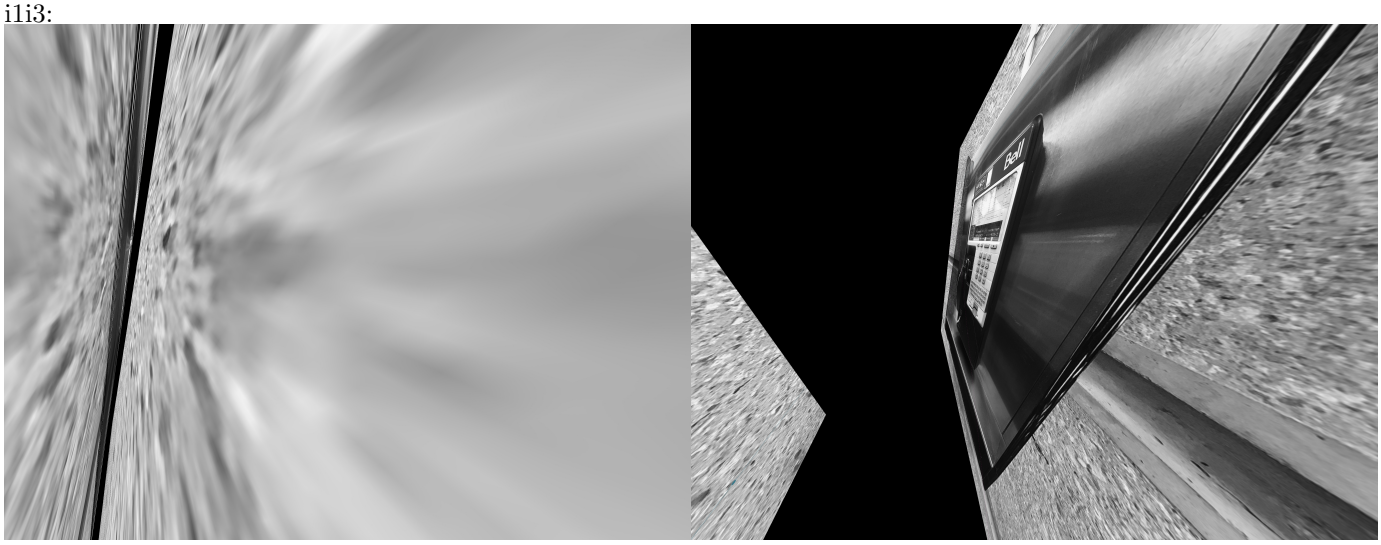
```
def rectify(img1, img2):
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)
    good = []
    pts1 = []
    pts2 = []
    # ratio test as per Lowe's paper
    for i, (m, n) in enumerate(matches):
        if m.distance < 0.8*n.distance:
            good.append(m)
            pts2.append(kp2[m.trainIdx].pt)
            pts1.append(kp1[m.queryIdx].pt)
    pts1 = np.int32(pts1)[:8]
    pts2 = np.int32(pts2)[:8]
    F = fundamental(pts1,pts2)
    # F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_LMEDS)
    # We select only inlier points

    res, H1, H2 = cv2.stereoRectifyUncalibrated(
        pts1, pts2, F, img1.shape[:2], threshold=15)
    left = cv2.warpPerspective(img1, H1, img1.shape[:2])
    right = cv2.warpPerspective(img2, H2, img2.shape[:2])
```

```
    return left, right
```

result looks like this, which is not very well matched:
i1i2:



i1i3:



e.

```
f12':
[[ 2.27351076e-07 -5.04321243e-07 -2.17969615e-04]
 [ 5.00556310e-07 2.34934407e-07 -1.45605855e-03]
 [-6.20657458e-04 8.18685316e-04 1.00000000e+00]]

f13':
[[ 3.95884427e-08 4.87071745e-07 -1.87284237e-03]
 [-5.90830328e-08 -6.92988723e-08 -9.55458613e-03]
 [ 9.40004913e-04 9.14397735e-03 1.00000000e+00]]
```

No they are not the same, they are not very similar either, this could because the 8 point chosen for my fundamental matrix is not the 8 best points but randomly chosen, also they are not very evenly distributed (most of them are at the center of picture).

f. Rectify using fundamental matrix computed by opencv:

```
def epipolar(img1, img2):
    print("finding kps")
    kp1, des1 = sift.detectAndCompute(img1, None)
```

```
    kp2, des2 = sift.detectAndCompute(img2, None)
    print("done")

    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1, des2, k=2)
    good = []
    pts1 = []
    pts2 = []
    # ratio test as per Lowe's paper
    for i, (m, n) in enumerate(matches):
        if m.distance < 0.8*n.distance:
            good.append(m)
            pts2.append(kp2[m.trainIdx].pt)
            pts1.append(kp1[m.queryIdx].pt)
    pts1 = np.int32(pts1)[:8]
    pts2 = np.int32(pts2)[:8]
    F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_LMEDS)
    # We select only inlier points

    # Find epilines corresponding to points in right image (second image) and
    # drawing its lines on left image
    lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1, 1, 2), 2, F)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines(img1, img2, lines1, pts1, pts2)

    # Find epilines corresponding to points in left image (first image) and
    # drawing its lines on right image
    lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1, 1, 2), 1, F)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines(img2, img1, lines2, pts2, pts1)

    return img5, img3
```

result looks like this:
i1i2:

i1i3:

this result is a lot better than my result, possibly because the key point selection is more careful.