گزارش پروژه درس بینایی کامپیوتر

علی عسگری

بخش اول)

در ابتدا تصاویر را از حافظه می خوانیم:

```
img1 = cv2.imread('images/building1.jpg')
img2 = cv2.imread('images/building2.jpg')
```

سپس برای بدست اوردن keypoint ها و محاسبه descriptor ها از orb استفاده میکنیم و همچنین برای بدست اوردن match ها بین آن ها از brute force matching استفاده کرده ایم بنابراین یک شی از آن ها می سازیم:

```
orb = cv2.ORB_create()
bf = cv2.BFMatcher()
```

سپس keypoint ها و descriptor های هر دو تصویر را محاسبه می کنیم و در keypoints ذخیره می کنیم:

```
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)
keypoints = [kp1, kp2]
```

حال باید match بین keypoint ها را بدست آوریم برای اینکار برای هر keypoint دو بهترین match را بدست می آوریم و با سنجش نسبت آن ها سعی می کنیم بهترین keypoint ها را نگه نداریم:

```
matches = bf.knnMatch(des1, des2, k=2)
# apply ratio test to find better matches
good = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good.append(m)
```

حال با استفاده از این match ها و استفاده از الگوریتم ransac ماتریس هوموگرافی بین دو تصویر را بدست می آوریم:

```
for match in good:
    (x1, y1) = keypoints[0][match.queryIdx].pt
    (x2, y2) = keypoints[1][match.trainIdx].pt
    correspondenceList.append([x1, y1, x2, y2])

corrs = np.matrix(correspondenceList)
# find homography between two images using ransac
H, inliers = ransac(corrs, estimation_thresh)
```

و تابع هوموگرافی بدست امده بین دو تصویر برابر است با:

```
Final homography:
[[ 1.27433014e+00 -1.47126459e-02 -3.76154550e+02]
 [ 1.34352030e-01  1.21501391e+00 -6.76636165e+01]
 [ 3.64538580e-04  1.07892974e-04  1.00000000e+00]]
```

در شکل زیر 10تا keypoint متناظر را نیز نشان داده ایم: (فایل در پیوست نیز وجود دارد)

```
matchImg = cv2.drawMatches(img1, kp1, img2, kp2, good[:10], None,
                           flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

می توانیم دو تصویر را به یکدیگر نیز stitch کنیم:

```
stitcher = cv2.Stitcher.create()
status, stitched_image = stitcher.stitch([img1, img2])
if status == cv2.STITCHER_OK:
    cv2.imshow('stitching', stitched_image)
```

بخش دوم)

ابتدا تصویر را از حافظه می خوانیم و طول و عرض آن را بدست می آوریم:

```
src = cv2.imread('images/room.jpg')
height, width = src.shape[:2]
```

نقاط متناظر با چهار گوشه تلوزیون (که میخواهیم تصویر را در آن محدوده قرار دهیم) را پیدا می کنیم و همچنین نقاط متناظر با 4 گوشه تصویر را نیز پیدا میکنیم:

```
pts1 = np.float32([[0, 0], [width, 0], [0, height], [width, height]])
# 4 Points corresponding to corners of the television
pts2 = np.float32([ [540, 300], [688, 295], [540, 385], [690, 390] ])
```

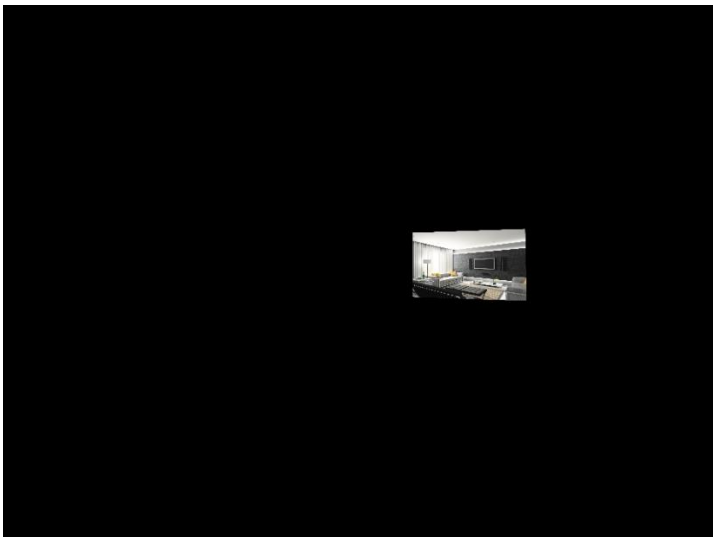
حال ماتریس هوموگرافی مرتبط با این 4 نقطه متناظر را پیدا میکنیم:

```
correspondences = []
for i in range(4):
    x1, y1 = pts1[i]
    x2, y2 = pts2[i]
    correspondences.append([x1, y1, x2, y2])

correspondences = np.matrix(correspondences)
# find homography between two images using ransac
H = calculateHomography(correspondences)
```

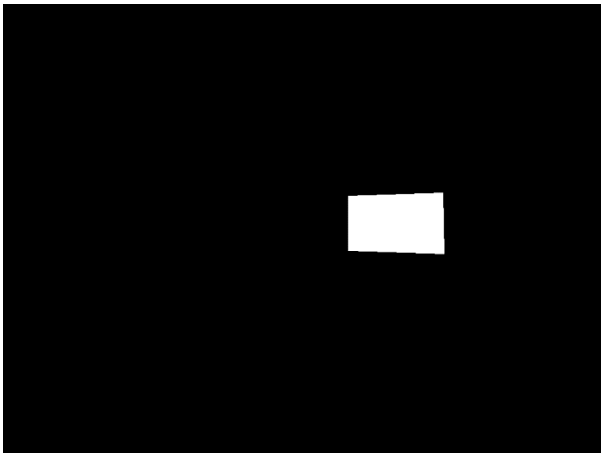
حال با استفاده از ماتریس به دست آمده تصویرمان را warp می کنیم:

```
img_reg = cv2.warpPerspective(src, H, (width, height))
```



حال باید یک mask بسازیم و تعیین کنیم که کدام بخش از تصویر جدیدی که میخواهیم بسیازیم تصویر بالایی باشد و چه بخشی تصویر اصلی:

```
mask = np.zeros(src.shape, dtype=np.uint8)
roi_corners = np.int32(pts2)
channel_count = src.shape[2]
ignore_mask_color = (255,) * channel_count
# determine the region image must place there
cv2.fillConvexPoly(mask,
                   roi_corners[[0, 1, 3, 2]],
                   ignore_mask_color)
```



اما ما معکوس این حالت را میخواهیم تا وقتی با تصویر اصلی آن را AND کردیم تصویر اصلی همه جا به غیر از داخل تلوزیون قرار بگیرد:

```
mask = cv2.bitwise_not(mask)
```

این تصویر دقیقا معکوس تصویر بالا است (جای سیاه و سفید عوض شده است)

حال تصویر اصلی را با این mask باید AND کنیم:

```
masked_image = cv2.bitwise_and(src, mask)
```

در تصویر نشان داده شده داخل تلوزیون کاملا خالی است و حال باید تصویر warp شده را از طریق OR کردن آن با تصویر بالایی در تصویر قرار دهیم تا تصویر نهایی حاصل شود:

```
final = cv2.bitwise_or(img_reg, masked_image)
```

بخش سوم)

ابتدا ویدیو و تصویر را از حافظه می خوانیم:

```
vid = cv2.VideoCapture('images/Video1.avi')
building = cv2.imread('images/building2.jpg', 0)
```

فریم اول از ویدیو را میخوانیم و آن را به سطح خاکستری تبدیل می کنیم (زیرا الگوریتم KLT و goodFeaturesToTrack تصاویر با سطح خاکستری می پذیرند):

```
_, frame1 = vid.read()
frame1_gray = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
```

حال باید چهار نقطه ای که میخواهیم تصویرمان در آن محل قرار بگیرد را در فریم اول مشخص کنیم. ترتیب انتخاب نقاط چهارگانه: 1. بالا چپ 2. بالا راست 3. پایین راست 4. پایین چپ

```
pts = np.array(select_region(VIDEO_PATH))
```

حال باید تصویری که میخواهیم augment کنیم را به اندازه فریم اول در بیاوریم و سپس آن را نسبت به چهار نقطه انتخابی warp کنیم تا در فریم اول جایگیری مناسب داشته باشد برای ان کار باید تبدیل بین چهار گوشه تصویر و چهار نقطه انتخابی را بدست آوریم:

```
height, width = frame1_gray.shape
building = cv2.resize(building, (width, height))
pts1 = np.array([[0, 0], [width, 0], [width, height], [0, height]])
transform = cv2.getPerspectiveTransform(np.float32(pts1),
np.float32(pts))
transform = cv2.getPerspectiveTransform(np.float32(pts1),
np.float32(pts))
```

حال یک mask می سازیم برای این که تعیین کنیم فقط در محدوده انتخاب شده بین چهار نقطه به دنبال یافتن ویژگی هستیم:

```
mask_frame = np.zeros_like(frame1_gray)
mask_frame[pts[1][1]: pts[2][1], pts[0][0]: pts[1][0]] = 255
pts = np.float32(pts.reshape((-1, 1, 2)))
```

حال ویژگی ها (گوشه ها) را در فریم اول و در محدوده انتخابی پیدا میکنیم:

```
feature_params = dict(maxCorners=250,
                      qualityLevel=0.3,
                      minDistance=7,
                      blockSize=7)
old_points = cv2.goodFeaturesToTrack(frame1_gray, mask=mask_frame,
**feature_params)
```

حال نقاط پیدا شده در فریم اول را در متغیری به نام temp قرار می دهیم. ما میخواهیم ماتریس هوموگرافی را بین این نقاط و نقاط رهگیری شده در هر فریم پیدا کنیم بنابراین باید آن را ذخیره کنیم:

```
temp = old_points
```

حال تعداد فریم های ویدئو را پیدا می کنیم زیرا میخواهیم یک حلقه for به تعداد فریم های ویدئو داشته باشیم زیرا در هر فریم باید رهگیری ویژگی ها و قرار دادن تصویر در ویدئو را انجام دهیم:

```
length = int(vid.get(cv2.CAP_PROP_FRAME_COUNT))
```

حال در داخل حلقه هر فریم را میخوانیم و آن را به سطح خاکستری تبدیل می کنیم و ویژگی ها را در آن فریم رهگیری می کنیم:

```
 _, new_frame = vid.read()
 new_frame_gray = cv2.cvtColor(new_frame, cv2.COLOR_BGR2GRAY)
 new_points, st, err = cv2.calcOpticalFlowPyrLK(old_frame_gray,
 new_frame_gray, old_points, None, **lk_params)
```

که در آن **lk_params برابر است با:

```
lk_params = dict(winSize=(17, 17),
 maxLevel=7,
 criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
```

حال از بین نقاط داده شده برای رهگیری فقط آن هایی را استفاده خواهیم کرد که در فریم جدید یافت شده اند یا به عبارت دیگر st آن ها برابر 1 است:

```
 good_new = new_points[st == 1]
 good_old = temp[st == 1]
```

حال از طریق این نقاط و الگوریتم ransac ماتریس هوموگرافی را پیدا میکنیم:

```
    correspondenceList = []
    for i in range(len(good_old)):
        x1, y1 = good_old[i]
        x2, y2 = good_new[i]
        correspondenceList.append([x1, y1, x2, y2])


    corrs = np.matrix(correspondenceList)
    # find Homography between 2 frames
    finalH, inliers = ransac(corrs, 1, 10, verbose=False)
```

حال باید فریمی که در این دور از حلقه جدید محسوب می شود را به عنوان فریم قدیم ثبت کنیم و همچنین نقاط جدید را به عنوان نقاط قدیم ثبت کنیم برای استفاده در دور بعدی حلقه:

```
    old_frame_gray = new_frame_gray
    old_points = new_points
```

حال با استفاده از ماتریس هوموگرافی بدست آمده تصویری را که میخواستیم aument کنیم را warp می کنیم تا مطابق با فریم جدید شود:

```
    img_warp = cv2.warpPerspective(building, finalH,
    (new_frame_gray.shape[1], new_frame_gray.shape[0]))
```

حال مجدد با استفاده از ماتریس هوموگرافی بدست آمده نقاط انتخاب شده را نیز warp میکنیم تا محدوده جدید انتخاب شده در فریم اول را نیز بدست آوریم:

```
new_pts = cv2.perspectiveTransform(pts, finalH)
```

حال یک mask می سازیم و مانند قبل بخشی را که میخواهیم تصویر را در آن augment کنیم انتخاب میکنیم:

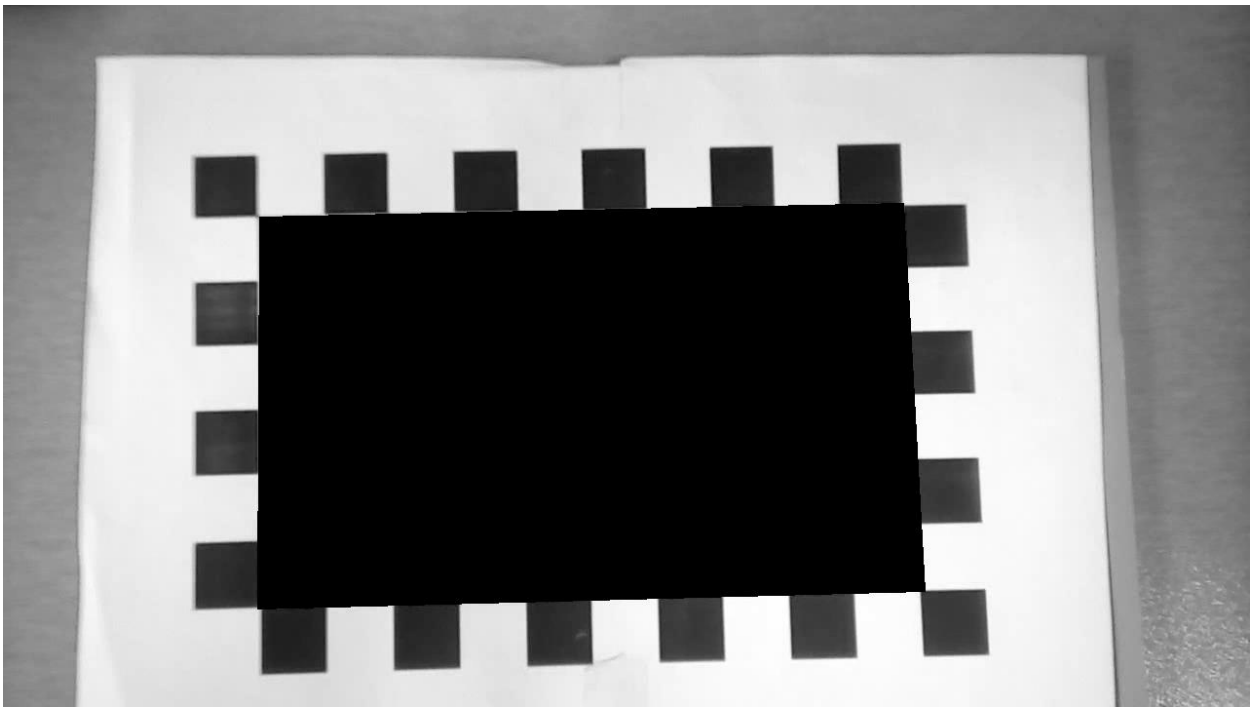
```
mask = np.zeros((new_frame_gray.shape[0], new_frame_gray.shape[1]),
np.uint8)
cv2.fillPoly(mask, [np.int32(new_pts)], (255, 255, 255))
```

البته همانند قبل ما معکوس mask فعلی را میخواهیم:

```
mask_inv = cv2.bitwise_not(mask)
```

حال فریم حال حاضر را با mask به دست امده AND میکنیم:

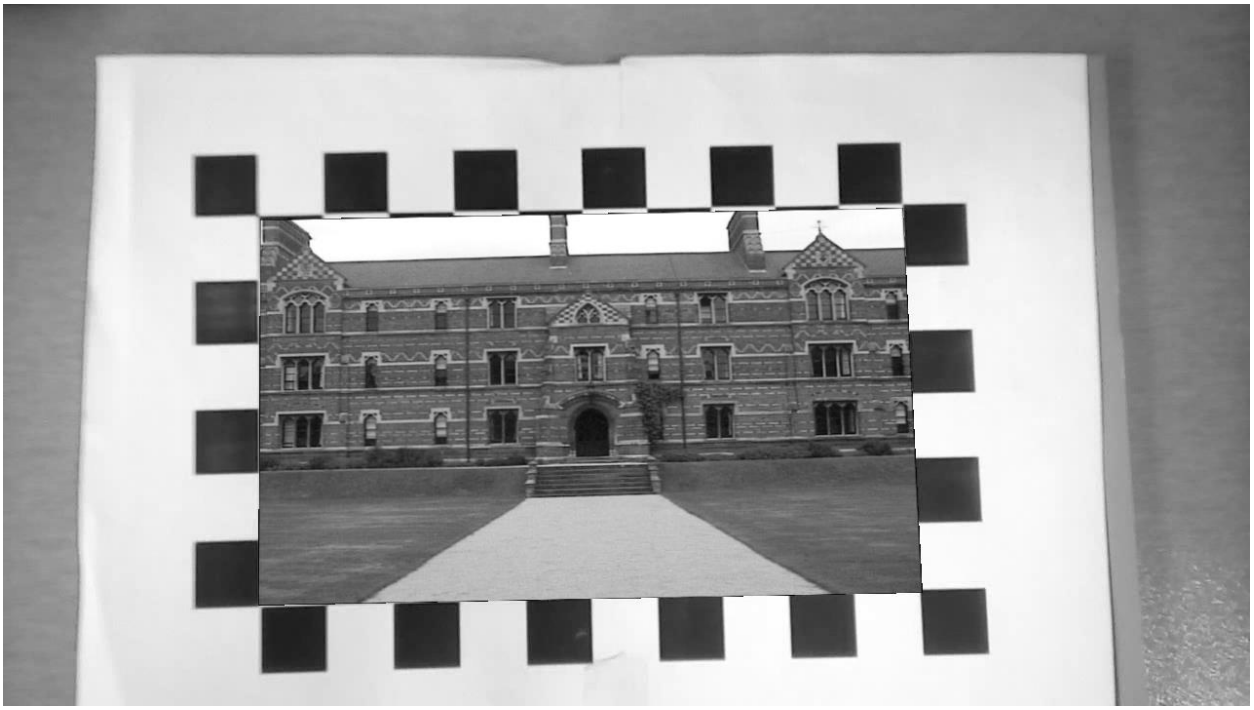
```
img_aug = cv2.bitwise_and(img_aug, mask_inv)
```



حال باید تصویر بالا را با تصویر building که آن را با توجه به ماتریس هوموگرافی بدست آمده بین دو فریم warp کردیم را یا هم ادغام کنیم:

```
img_aug = cv2.bitwise_or(img_warp, img_aug)
```

و نتیجه نهایی خواهد بود:



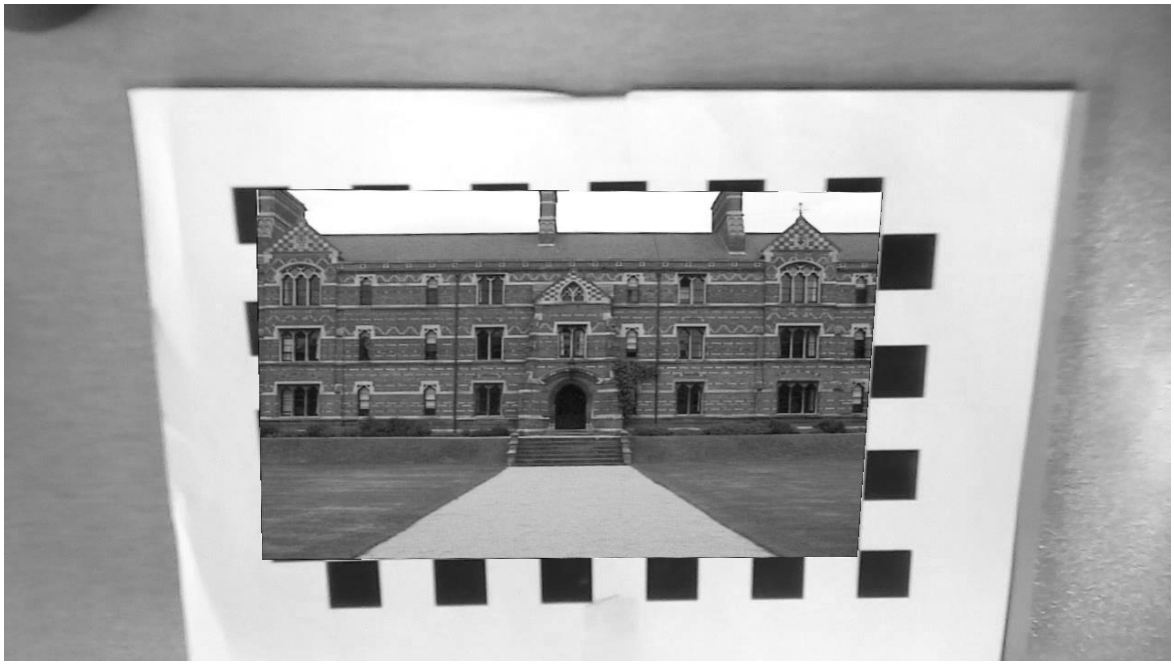
** برای نمایش ویژگی های رهگیری شده هم از این بخش از کد استفاده کرده ایم. هر ویژگی موجود در آن فریم را به عنوان یک دایره با شعاع 2 و ضخامت 5 و رنگ مشکی نمایش داده ایم:

```
IMAGE = new_frame_gray.copy()
for x, y in good_new:
    IMAGE = cv2.circle(IMAGE, (np.int32(x), np.int32(y)), 2,
    (0, 0, 0), 5)
    cv2.imshow('tracking features', IMAGE)
```

در فولدر تصاویر 5 فریم متوالی رهگیری شده نشان داده شده است(فریم های 20 تا 24)

مشکلاتی باعث می شود که تصویر به درستی در محدوده فریم انتخابی قرار نگیرد که احتمالا یکی از دلایل آن تخمین نادقیق ماتریس هوموگرافی است. مثلا در فریم نمایش داده شده تعداد inlier های متناظر با فریم نسبت به تعداد کل عدد کمی است (35 به 128):

```
Iter 0 Corr size:  128  NumInliers:  27 Max inliers:  27
Iter 1 Corr size:  128  NumInliers:  9 Max inliers:  27
Iter 2 Corr size:  128  NumInliers:  31 Max inliers:  31
Iter 3 Corr size:  128  NumInliers:  4 Max inliers:  31
Iter 4 Corr size:  128  NumInliers:  15 Max inliers:  31
Iter 5 Corr size:  128  NumInliers:  21 Max inliers:  31
Iter 6 Corr size:  128  NumInliers:  35 Max inliers:  35
Iter 7 Corr size:  128  NumInliers:  27 Max inliers:  35
Iter 8 Corr size:  128  NumInliers:  12 Max inliers:  35
Iter 9 Corr size:  128  NumInliers:  16 Max inliers:  35
Iter 10 Corr size:  128  NumInliers:  22 Max inliers:  35
Iter 11 Corr size:  128  NumInliers:  4 Max inliers:  35
Iter 12 Corr size:  128  NumInliers:  4 Max inliers:  35
Iter 13 Corr size:  128  NumInliers:  18 Max inliers:  35
Iter 14 Corr size:  128  NumInliers:  7 Max inliers:  35
```



یک راه حل که به تخمین دقیقتر این موضوع کمک می کند افزایش تعداد سطوح pyramid در الگوریتم KLT است که با افزایش پارامتر MaxLevel از 2 به 7 در تابع calcOpticalFlowPyrLK دقت به وضوح افزایش یافت.

توضیح توابع فایل utils

تابع calculateHomography:

تابع هوموگرافی را با توجه به آن چه در فایل pdf آمده بود محاسبه کرده ایم:

برای هر زوج نقطه متناظر ابتدا نقطه اول را در متغیر p1 ذخیره می کنیم و با اضافه کردن 1 به بعد سوم آن، آن را به homogenous تبدیل می کنیم. همین کار را برای نقطه دوم انجام می دهیم و در متغیر p2 ذخیره می کنیم:

```
for corr in correspondences:
    p1 = np.matrix([corr.item(0), corr.item(1), 1])
    p2 = np.matrix([corr.item(2), corr.item(3), 1])
```

سپس با توجه به آن چه در فایل pdf آمده بود بردار های a1 و a2 را می سازیم:

$$\begin{aligned} \mathbf{a}_x &= (-x_1, -y_1, -1, 0, 0, 0, x'_2 x_1, x'_2 y_1, x'_2)^T \\ \mathbf{a}_y &= (0, 0, 0, -x_1, -y_1, -1, y'_2 x_1, y'_2 y_1, y'_2)^T . \end{aligned}$$

```
a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2), 0, 0, 0,
      p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.item(2),
      p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
```

و این کار را برای تمام نقاط match انجام می دهیم و آن ها را در یک لیست قرار می دهیم و آن لیست را تبدیل به یک ماتریس میکنیم:

```
aList.append(a1)
aList.append(a2)

matrixA = np.matrix(aList)
```

حال svd این ماتریس را محاسبه می کنیم:

```
u, s, v = np.linalg.svd(matrixA)
```

کوچکترین ستون ماتریس v متناظر با کوچکترین singular value برابر ماتریس h است. البته برای نتیجه بهتر ماتریس h را نرمالیزه نیز میکنیم:

```
h = np.reshape(v[8], (3, 3))
h = (1/h.item(8)) * h
```

تابع geometricDistance:

این تابع یک زوج نقطه متناظر و تابع هوموگرافی را به عنوان ورودی می گیرد سپس نقطه اول را با توجه به ماتریس هوموگرافی تبدیل می کند به نقطه متناظرش. فاصله یا norm بین نقطه تبدیل یافته و نقطه اصلی میزان خطای ماست:

نقطه اول را میگیریم و آن را به فضای همگن می بریم:

```
p1 = np.transpose(np.matrix([correspondence[0].item(0),
correspondence[0].item(1), 1]))
```

نقطه اول را با استفاده از ماتریس هوموگرافی تبدیل به نقطه متناظرش میکنیم و آن را نرمالیزه نیز می کنیم:

```
estimatep2 = np.dot(h, p1)
estimatep2 = (1/estimatep2.item(2))*estimatep2
```

نقطه دوم را نیز به فضای همگن می بریم

```
p2 = np.transpose(np.matrix([correspondence[0].item(2),
correspondence[0].item(3), 1]))
```

نرم فاصله هندسی این دو میزان خطای ماست:

```
error = p2 - estimatep2
np.linalg.norm(error)
```

توضیح تابع ransac:

به ازای تعداد تکرار هایمان عملیات های زیر را انجام می دهیم:

ابتدا 4 زوج نقطه متناظر را به صورت تصادفی پیدا انتخاب می کنیم و آن ها را به صورت stack می کنیم تا یک ماتریس ساخته شود:

```
corr1 = corr[random.randrange(0, len(corr))]
corr2 = corr[random.randrange(0, len(corr))]
corr3 = corr[random.randrange(0, len(corr))]
corr4 = corr[random.randrange(0, len(corr))]
randomFour = np.vstack((corr1, corr2, corr3, corr4))
```

سپس تابع هوموگرافی را با استفاده از این 4 زوج نقطه متناظر پیدا می کنیم:

```
h = calculateHomography(randomFour)
```

سپس باید تعداد inlier ها را پیدا کنیم، برای این کار برای تمام زوج نقاط متناظر نقطه اول را با استفاده از ماتریس هوموگرافی تبدیل به نقطه دوم متناظرش می کنیم و فاصله هندسی بین آن دو را می سنجیم اگر این فاصله کمتر از یک حد آستانه بود آن زوج نقطه را به عنوان inlier به حساب می آوریم و در لیست inliers ذخیره میکنیم و در نهایت ماتریس هوموگرافی با بیشترین تعداد نقاط inlier را به عنوان پاسخ بر می گردانیم:

```
for i in range(len(corr)):
  d = geometricDistance(corr[i], h)
  if d < 5:
      inliers.append(corr[i])
  if len(inliers) > len(maxInliers):
      maxInliers = inliers
      finalH = h
```