Assignment #5

HARNEET SINGH - 400110275

COMPENG 4TN4: Image Processing

March 13, 2022

Theory

1. Harris Corner Detection

For the given image, blurring step is not applied (assuming no noise in the image).

- As a first step, image derivatives are found in both x and y directions. Using the left and top pixel as the anchor pixels for horizontal and vertical kernel respectively, we can find the gradients by convolving the given kernels with the image. Result of Image gradient is shown below (assuming top left non zero pixel as (1, 1)):

$$I_x = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 255 & 0 & 0 & -255 & 0 \\ 255 & 0 & 0 & -255 & 0 \\ 255 & 0 & 0 & -255 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$, Sample Calculation, I_{11} = -1*255 + 255*1 = 0

$$I_y = \begin{bmatrix} 0 & 255 & 255 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -255 & -255 & -255 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$, Sample Calculation, I_{01} = -1*0 + 255*1 = 255

- Square of derivatives i.e. I_x^2 and I_y^2:

$$I_x^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 65025 & 0 & 0 & 65025 & 0 \\ 65025 & 0 & 0 & 65025 & 0 \\ 65025 & 0 & 0 & 65025 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$I_y^2 = \begin{bmatrix} 0 & 65025 & 65025 & 65025 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 65025 & 65025 & 65025 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- Computing the second moment matrix $M = \sum \begin{bmatrix} I_x^2 & I_x.I_y \\ I_x.I_y & I_y^2 \end{bmatrix}$ for each pixel using the values computed in the previous steps. Assuming top left corner of the image is (0, 0) and only non-zero pixels are shown below.

$$M_{11} = \begin{bmatrix} 65025. & 0. \\ 0. & 65025. \end{bmatrix} \quad M_{12} = \begin{bmatrix} 0. & 0. \\ 0. & 130050. \end{bmatrix} \quad M_{13} = \begin{bmatrix} 65025. & 0. \\ 0. & 130050. \end{bmatrix}$$

$$M_{21} = \begin{bmatrix} 130050. & 0. \\ 0. & 0. \end{bmatrix} \quad M_{22} = \begin{bmatrix} 0. & 0. \\ 0. & 0. \end{bmatrix} \quad M_{23} = \begin{bmatrix} 130050. & 0. \\ 0. & 0. \end{bmatrix}$$

$$M_{31} = \begin{bmatrix} 130050. & 0. \\ 0. & 65025. \end{bmatrix} \quad M_{32} = \begin{bmatrix} 0. & 0. \\ 0. & 130050. \end{bmatrix} \quad M_{33} = \begin{bmatrix} 130050. & 65025. \\ 65025. & 130050. \end{bmatrix}$$

- Compute the eigen values using $\lambda = \pm\frac{1}{2}[(h_{11} + h_{22}) \pm \sqrt{4h_{12}h_{21} + (h_{11} - h_{22})^2}]$

$\lambda_{11} = 65025, 65025$ $\quad \lambda_{12} = 0, 130050$ $\quad \lambda_{13} = 65025, 130050$

$\lambda_{21} = 0, 130050$ $\quad \lambda_{22} = 0, 0$ $\quad \lambda_{23} = 0, 130050$

$\lambda_{31} = 65025, 130050$ $\quad \lambda_{32} = 0, 130050$ $\quad \lambda_{33} = 65025, 195075$

Another way of finding the eigen value is shown below (sample calculation shown for M_{33}):

$M_{33}\vec{v} = \lambda\vec{v} \quad \Rightarrow (M - \lambda I)\vec{v} = \vec{0}$

$det(M - \lambda I) = \vec{0} \quad \Rightarrow \begin{vmatrix} 130050 - \lambda I & 65025 \\ 65025 & 130050 - \lambda I \end{vmatrix} = \vec{0}$

$(130050 - \lambda)^2 - 65025^2 = 0 \qquad 130050 - \lambda = \pm 65025$

$\lambda = 65025, 195075$

Sample result matches the code outcome as shown above.

- Using $R = det(M) - \alpha trace(M)^2 = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$ cornerness score as a thresholding function (sample calculation shown below for M_{11} using $\alpha = 0.04$):

  $R_{11} = 3551730525 = (65025 * 65025) - 0.04 * (65025 + 65025)^2 \qquad R_{12} = -676520100 \qquad R_{13} = 6934331025$

  $R_{21} = -676520100 \qquad R_{22} = 0 \qquad R_{23} = -676520100$

  $R_{31} = 6934331025 \qquad R_{32} = -676520100 \qquad R_{33} = 9978671475$

- $f(x) = \begin{cases} corner, \text{ if } R > threshold, \text{ using threshold} = 1000 \\ edge, \text{ if } R << 0 \\ Flat, \text{ if } R \cong 0 \end{cases}$

  $R_{11} = Corner \qquad R_{12} = Edge \qquad R_{13} = Corner$

  $R_{21} = Edge \qquad R_{22} = Flat \qquad R_{23} = Edge$

  $R_{31} = Corner \qquad R_{32} = Edge \qquad R_{33} = Corner$

  Corners in the given image $= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Following Python code uses the same technique to compute corner values for all the non-zero pixels in the given image:
NOTE: Zero-padding is not applied to the given binary image.

```
import cv2
import numpy as np
import math

# creating the image
img = np.array([[0, 0, 0, 0, 0],
                [0, 255, 255, 255, 0],
                [0, 255, 255, 255, 0],
                [0, 255, 255, 255, 0],
                [0, 0, 0, 0, 0]])
imgShape = int(img.shape[0])
resultImg = np.zeros_like(img)

# creating kernels for x and y directions
kernelX = np.array([[-1, 1]])
kernelY = np.copy(kernelX)
kernelY = np.transpose(kernelY)
kernelX_h, kernelX_w = kernelX.shape
kernelY_h, kernelY_w = kernelY.shape


# function to extract gradients using the kernel (convolving kernel over the image)
def gradExtraction(kernel, ker_h, ker_w):
    filteredImg = np.zeros_like(img)

    for k in range(imgShape - ker_h + 1):
        for p in range(imgShape - ker_w + 1):
            subMatrix = np.zeros((ker_h, ker_w))
            subMatrix = img[k:k + ker_h, p:p + ker_w]
            multipliedValue = np.multiply(subMatrix, kernel)
            sumAllElements = np.sum(multipliedValue)
```

```
            filteredImg[k, p] = sumAllElements
    return filteredImg


# used to find max and min lambda values
def findMinMax(num1, num2):
    if num1 > num2:
        return num1, num2
    else:
        return num2, num1


# following variables used to calculate M, R and eigen values
I_x = gradExtraction(kernelX, kernelX_h, kernelX_w)
I_y = gradExtraction(kernelY, kernelY_h, kernelY_w)
# print(f'{I_x=}')
# print(f'{I_y=}')

I_xSquared = np.square(I_x)
I_ySquared = np.square(I_y)
# print(f'{I_xSquared=}')
# print(f'{I_ySquared=}')

Ix_Iy = np.multiply(I_x, I_y)

M = np.zeros((9, 2, 2))
i = 0
alpha = 0.04

# going through each non-zero pixel at a time to get lambda max and min, and cornerness score
for h in range(1, 4):
    for w in range(1, 4):
        sum_I_xSquared = int(np.sum(I_xSquared[h-1:h+1, w-1:w+1]))
        sum_I_ySquared = int(np.sum(I_ySquared[h-1:h+1, w-1:w+1]))
        sum_Ix_Iy = int(np.sum(Ix_Iy[h-1:h+1, w-1:w+1]))
        M[i] = [[sum_I_xSquared,    sum_Ix_Iy],
                [sum_Ix_Iy,         sum_I_ySquared]]
        # print(f'{i}{M[i]=}')

        sqrtTerm = 4*sum_Ix_Iy*sum_Ix_Iy + ((sum_I_xSquared-sum_I_ySquared)**2)
        lambda_1 = (1/2) * ((sum_I_xSquared+sum_I_ySquared) + math.sqrt(sqrtTerm))
        lambda_2 = (1/2) * ((sum_I_xSquared+sum_I_ySquared) - math.sqrt(sqrtTerm))

        lambdaMax, lambdaMin = findMinMax(lambda_1, lambda_2)
        # print(f'{i}: {lambdaMin=},\t\t {lambdaMax=}')

        # detect corners if above certain threshold:
        # cornerness score:
        R = lambda_1 * lambda_2 - alpha * (lambda_1 + lambda_2) ** 2
        # print(f'{R=}')

        # generally, we can apply a normalization function to the M matrix
        # in this case, since we need true lambda values, I've omitted the normalization function
        # threshold is computed based on the trend between all non-zero values
        # higher value could have been chosen but a value of 1000 works as well
        # (essentially a large positive value will be a good candidate for threshold)
        # Note that I am assuming top-left corner is (0 , 0) in the image
        if R > 1000:
            # print(f'M{i}: \n{M[i]} with {R=}, {lambdaMax=} and {lambdaMin=}. Corner at location ({h}, {w})\n')
            resultImg[h, w] = 255

```

```
        # for comparison with my lambda values:
        eigVal, eigVec = np.linalg.eig(M[i])

        i = i + 1

print(f'Result Image with corners is \n{resultImg}')
```

2. Scale selection, LoG

$$LoG = \nabla^2 G = (\frac{x^2 + y^2 - 2\sigma^2}{\sigma^4}).e^{-\frac{x^2+y^2}{2\sigma^2}}$$

At maximum response, the max value is at the center of the Laplacian response distribution. Therefore, the edges of the circle will be zero-crossing at some distance because the distribution is in the shape of a bell curve and its value reduce as we move away from the center.
We can say that zeros of the Laplacian are aligned with the circle, when $\nabla^2 G = 0$

$$\Rightarrow (\frac{x^2 + y^2 - 2\sigma^2}{\sigma^4}).e^{-\frac{x^2+y^2}{2\sigma^2}} = 0$$

$$\Rightarrow (\frac{x^2 + y^2 - 2\sigma^2}{\sigma^4}) = 0$$

$$\because e^{-\frac{x^2+y^2}{2\sigma^2}} \neq 0$$

Also, we know that the equation of a circle located at origin is $x^2 + y^2 = r^2$

$$\Rightarrow (\frac{r^2 - 2\sigma^2}{\sigma^4}) = 0$$

$$\Rightarrow r^2 - 2\sigma^2 = 0$$

$$\Rightarrow r^2 = 2\sigma^2$$

$$\Rightarrow \sigma^2 = \frac{r^2}{2}$$

$$\Rightarrow \sigma = \sqrt{\frac{r^2}{2}} = \frac{r}{\sqrt{2}}$$

$$\boxed{\sigma = \frac{r}{\sqrt{2}}} \tag{1}$$

3. RANSAC

N = number of iterations,
p = probability that at least 1 set of random samples do not include an outlier (usually equal to 0.99)
u = probability that any selected data point is an inlier
m = required number of samples for each iteration

As we know, u = probability of an inlier in a sample,
$u....u = u^m$ = probability that all m samples are inliers
$1 - u^m$ = probability that one or more points in the sample are outliers
$(1 - u^m)^N$ = probability that all N samples are outliers (i.e. probability that N samples are contaminated)
$1 - (1 - u^m)^N = p$, probability that at least one set of sample was not contaminated i.e. at least one set of sample (each set of m points) is composed of only inliers.

$$\boxed{p = 1 - (1 - u^m)^N} \tag{2}$$

$\Rightarrow 1 - p = (1 - u^m)^N$
$\therefore$, take log on both sides: $log(1 - p) = log((1 - u^m)^N)$ $\Rightarrow log(1 - p) = N.log(1 - u^m)$

$$\boxed{\Rightarrow N = \frac{log(1 - p)}{log(1 - u^m)}} \tag{3}$$

Given p = 0.99, u = 0.70, m = 2:

$$\Rightarrow N = \frac{log(1 - 0.99)}{log(1 - 0.7^2)} = \frac{-2}{-0.292)} = 6.83$$

$$\boxed{N = 7}$$

Therefore, at least 7 samples are required.

Implementation

1. Change point of view

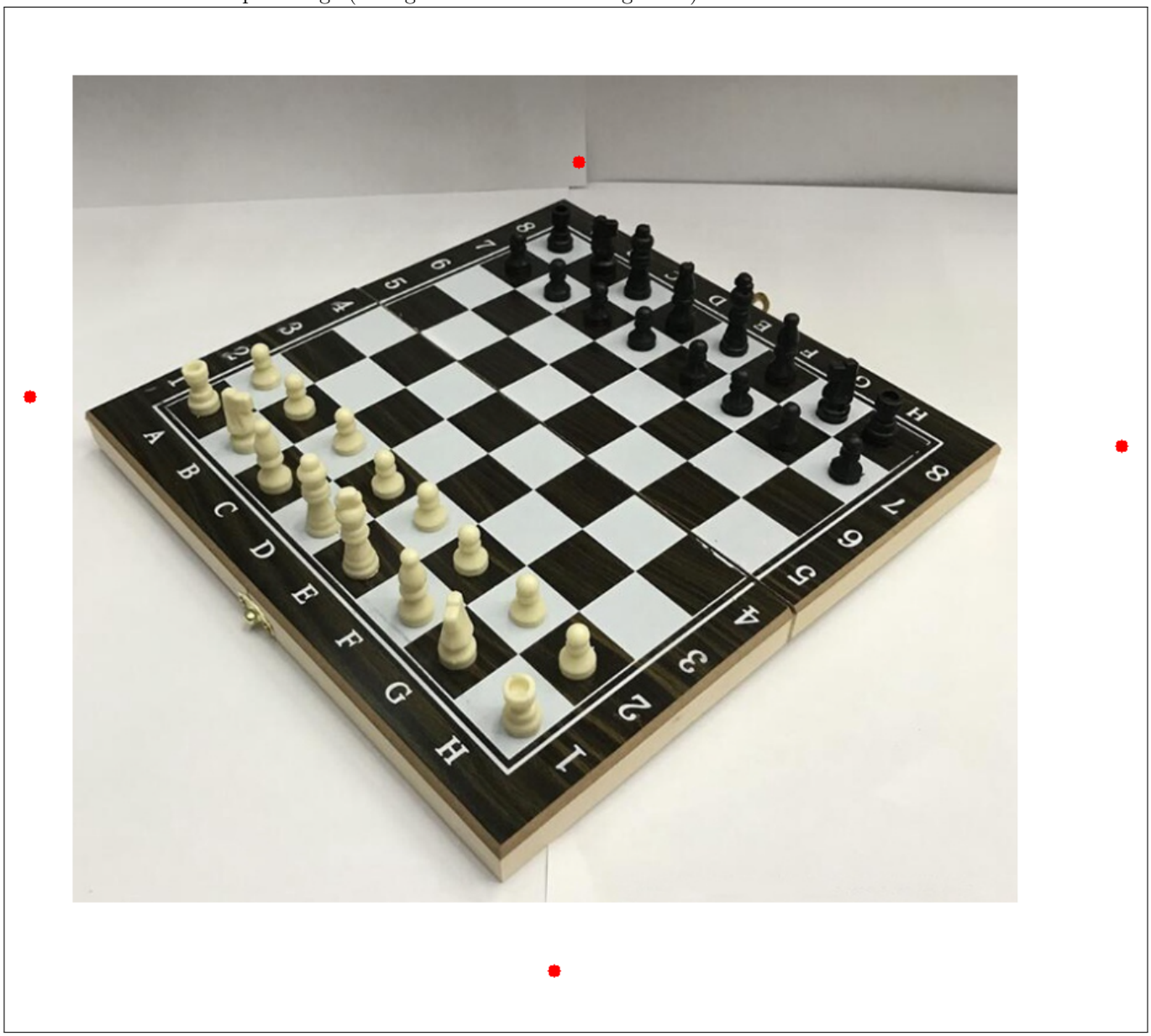
Four points are required to change the perspective of the image, each point represents the corner of a rectangle that will be transformed into the final result.

Python Script:

```python
import cv2
import numpy as np

# read given image
img = cv2.imread("chess.png")
print(f'Given image size is {img.shape}')
height, width, _ = img.shape

# creating enlarged image of the given image for white background requirement
imgEnlarge = np.full((height+150, width+150, 3), 255, dtype='uint8')
imgEnlarge[50:50+height, 50:50+width] = img

# find the co-ordinates of the chess board to perform warp
# used trial and error to find the correct pixel values (and Paint app)
topLeftPt, topRightPt = [460, 120], [900, 350]
bottomLeftPt, bottomRightPt = [15, 310], [440, 775]

# using the points in image A to transform the image
inputPts = np.float32([topLeftPt, topRightPt, bottomLeftPt, bottomRightPt])

# creating output points to tell function where to warp the image
outputPts = np.float32([[0, 0], [width, 0], [0, height], [width, height]])

# use cv2's perspective transform function
transformationMat = cv2.getPerspectiveTransform(inputPts, outputPts)
outImg = cv2.warpPerspective(imgEnlarge, transformationMat, (width, height))

# drawing points on the image to show the corners
for x, y in inputPts:
    cv2.circle(imgEnlarge, (int(x), int(y)), 5, (0, 0, 255), cv2.FILLED)

cv2.imshow("chess image", imgEnlarge)
cv2.imshow("transformed image", outImg)
# cv2.imwrite("impl_1_input_points.png", imgEnlarge)
# cv2.imwrite("impl_1_result.png", outImg)
cv2.waitKey(0)
```

Result:

Following image shows the point that have been utilized to warp the image as the final result image. Note the location of the red dots on the input image (enlarged to add white background).



Warped Image shown below:

Note that border has been added to highlight the corners and edges of the image, it is not part of the result image.

2. Visualize Matched points

Using ORB (similar to SIFT algorithm) in the implementation. In the result, I am displaying the top 20 matches between the images to avoid confusion between the connections as it becomes difficult to trace a connection when more matches are shown in the result. Also, note that all the top 20 matches are correct, but as we increase the matches it may not be the case. Care should be taken, because ORB may create false positive matches between two images.

Python Script:

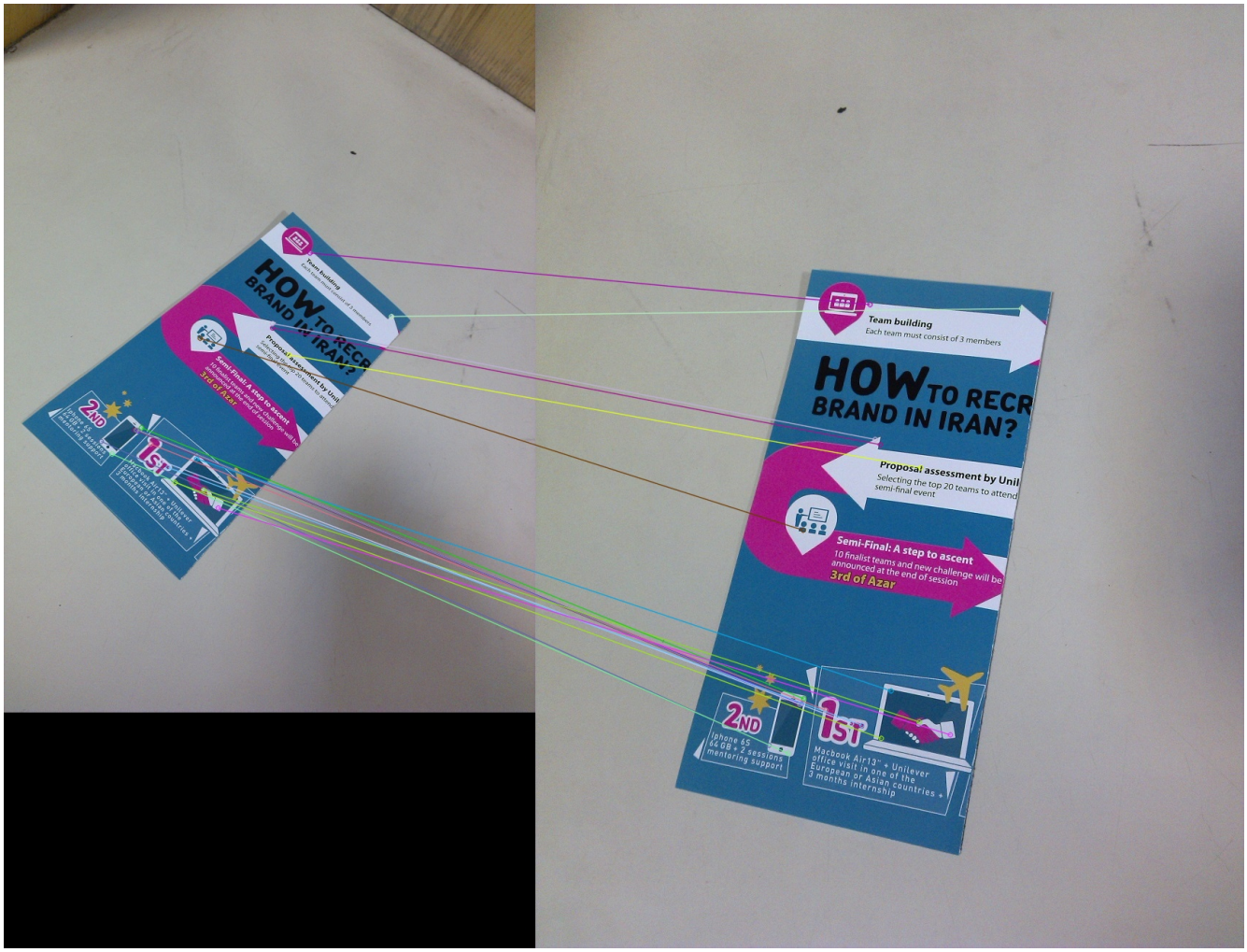
```
import cv2
import numpy as np

# read given images
img1Color = cv2.imread("image1_1.jpg")
img2Color = cv2.imread("image1_2.jpg")
img1 = cv2.cvtColor(img1Color, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2Color, cv2.COLOR_BGR2GRAY)

# Initiate ORB detector
# ORB is similar to SIFT algorithm to detect features and keypoints
orb = cv2.ORB_create()

# find keypoints and descriptors in both images
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

# use brute force method to match the keypoints
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des1, des2)
print(f'Total number of matches is {len(matches) = }')

# sort the matched based on the distance between the matches of the images
# shorter distance represents a better match, therefore it will allow us
# to pick top n matches
matches = sorted(matches, key=lambda x: x.distance)

# iterate through the match's distance
print("Distance between matches is:")
for m in matches:
    print(m.distance, end=' ')

# displaying top 20 matches on the images
matchedImages = cv2.drawMatches(img1Color, kp1, img2Color, kp2, matches[:20], None, flags=2)

# cv2.imshow("image 1", img1)
# cv2.imshow("image 2", img2)
cv2.imshow("matched", matchedImages)
cv2.imwrite("impl_2_result.jpg", matchedImages)
cv2.waitKey(0)
```

Result:

Total number of matches between the two images = 192

Distance between matches is (in ascending order):

15.0, 16.0, 17.0, 19.0, 19.0, 21.0, 23.0, 23.0, 23.0, 23.0, 25.0, 25.0, 26.0, 26.0, 27.0, 28.0, 28.0, 28.0, 28.0, 28.0, 29.0, 29.0, 29.0, 29.0, 29.0, 29.0, 29.0, 30.0, 30.0, 30.0, 31.0, 31.0, 31.0, 31.0, 32.0, 32.0, 32.0, 32.0, 32.0, 32.0, 32.0, 32.0, 32.0, 33.0, 33.0, 33.0, 33.0, 33.0, 33.0, 33.0, 34.0, 34.0, 34.0, 34.0, 34.0, 34.0, 35.0, 35.0, 35.0, 35.0, 35.0, 35.0, 35.0, 35.0, 36.0, 36.0, 36.0, 36.0, 36.0, 37.0, 37.0, 37.0, 37.0, 37.0, 37.0, 38.0, 38.0, 38.0, 38.0, 38.0, 38.0, 38.0, 39.0, 39.0, 39.0, 39.0, 39.0, 39.0, 39.0, 39.0, 40.0, 40.0, 40.0, 40.0, 40.0, 41.0, 41.0, 41.0, 42.0, 42.0, 42.0, 43.0, 43.0, 43.0, 43.0, 43.0, 44.0, 44.0, 44.0, 44.0, 44.0, 45.0, 45.0, 45.0, 46.0, 46.0, 46.0, 46.0, 46.0, 46.0, 46.0, 47.0, 48.0, 49.0, 49.0, 50.0, 50.0, 50.0, 50.0, 51.0, 51.0, 51.0, 51.0, 52.0, 53.0, 53.0, 53.0, 53.0, 53.0, 54.0, 54.0, 54.0, 55.0, 55.0, 55.0, 56.0, 56.0, 56.0, 56.0, 56.0, 57.0, 57.0, 57.0, 57.0, 57.0, 57.0, 58.0, 58.0, 58.0, 58.0, 59.0, 59.0, 60.0, 61.0, 61.0, 61.0, 62.0, 62.0, 62.0, 64.0, 64.0, 64.0, 65.0, 66.0, 66.0, 68.0, 69.0, 69.0, 69.0, 70.0, 70.0, 70.0, 71.0, 71.0, 71.0, 72.0, 72.0, 73.0, 74.0, 74.0, 76.0, 78.0



Note, the result only shows the top 20 matches between the images.

3. Solving a puzzle

Python Script:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# read given images
img21Color = cv2.imread("image2_1.jpg")
img22Color = cv2.imread("image2_2.jpg")
img23Color = cv2.imread("image2_3.jpg")
img24Color = cv2.imread("image2_4.jpg")
img1 = cv2.cvtColor(img21Color, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img22Color, cv2.COLOR_BGR2GRAY)
img3 = cv2.cvtColor(img23Color, cv2.COLOR_BGR2GRAY)
img4 = cv2.cvtColor(img24Color, cv2.COLOR_BGR2GRAY)


# this function will use SIFT to find keypoints and descriptors
def detectAndDescribe(img):
    descriptor = cv2.SIFT_create()

    # get keypoints and descriptors
    kps, features = descriptor.detectAndCompute(img, None)
    return kps, features


# getting the keypoints and features of all the images
kps1, features1 = detectAndDescribe(img1)
kps2, features2 = detectAndDescribe(img2)
kps3, features3 = detectAndDescribe(img3)
kps4, features4 = detectAndDescribe(img4)
# print(f'feature shape of feature 1 is {features1.shape}')
# print(f'{features1=}')


# function is used to display keypoints on the image (shows gradient: strength and orientation)
def drawKeyPoints(img, kps):
    img = cv2.drawKeypoints(img, kps, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    return img

# calling functions to draw gradients for all images
img1_kps = drawKeyPoints(img1, kps1)
img2_kps = drawKeyPoints(img2, kps2)
img3_kps = drawKeyPoints(img3, kps3)
img4_kps = drawKeyPoints(img4, kps4)

# plotting keypoints on the images
# plt.subplot(221), plt.imshow(img1_kps, cmap='gray', vmin=0, vmax=255)
# plt.title('img1_kps'), plt.xticks([]), plt.yticks([])
# plt.subplot(222), plt.imshow(img2_kps, cmap='gray', vmin=0, vmax=255)
# plt.title('img2_kps'), plt.xticks([]), plt.yticks([])
# plt.subplot(223), plt.imshow(img3_kps, cmap='gray', vmin=0, vmax=255)
# plt.title('img3_kps'), plt.xticks([]), plt.yticks([])
# plt.subplot(224), plt.imshow(img4_kps, cmap='gray', vmin=0, vmax=255)
# plt.title('img4_kps'), plt.xticks([]), plt.yticks([])
# plt.show()

# creating matcher object to be used in matching features of two images
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

```

```
59
60  # function matches the features between two images and
61  # computes distance between the features
62  def bruteForceMatcher(feat1, feat2):
63      # Match descriptors.
64      matches = bf.match(feat1, feat2)
65      # Sort them in the order of their distance.
66      matches = sorted(matches, key=lambda x: x.distance)
67
68      return matches
69
70  # calling function to match features between all pairs of images
71  matchesImg12 = bruteForceMatcher(features1, features2)
72  matchesImg13 = bruteForceMatcher(features1, features3)
73  matchesImg14 = bruteForceMatcher(features1, features4)
74  matchesImg23 = bruteForceMatcher(features2, features3)
75  matchesImg24 = bruteForceMatcher(features2, features4)
76  matchesImg34 = bruteForceMatcher(features3, features4)
77
78
79  # function is used to plot the matched points on two images
80  def drawMatchedImages(matches, image1, kp1, image2, kp2):
81      # Draw first 100 matches.
82      matchedImg = cv2.drawMatches(image1, kp1, image2, kp2, matches[:100],
83                                   None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
84      plt.figure(figsize=(20, 10))
85      plt.imshow(matchedImg)
86      plt.show()
87
88
89  # calling function to visualize matching points between all images in pairs
90  # drawMatchedImages(matchesImg12, img1, kps1, img2, kps2)
91  # drawMatchedImages(matchesImg13, img1, kps1, img3, kps3)
92  # drawMatchedImages(matchesImg14, img1, kps1, img4, kps4)
93  # drawMatchedImages(matchesImg23, img2, kps2, img3, kps3)
94  # drawMatchedImages(matchesImg24, img2, kps2, img4, kps4)
95  # drawMatchedImages(matchesImg34, img3, kps3, img4, kps4)
96
97
98  # function gets the homography matrix to be able to transform images over each other
99  def getHomography(kpsA, kpsB, featuresA, featuresB, matches, reprojThresh):
100     # convert the keypoints to numpy arrays
101     kpsA = np.float32([kp.pt for kp in kpsA])
102     kpsB = np.float32([kp.pt for kp in kpsB])
103
104     if len(matches) > 4:
105
106         # construct the two sets of points
107         ptsA = np.float32([kpsA[m.queryIdx] for m in matches])
108         ptsB = np.float32([kpsB[m.trainIdx] for m in matches])
109
110         # estimate the homography between the sets of points
111         H, stat = cv2.findHomography(ptsA, ptsB, cv2.RANSAC, reprojThresh)
112
113         return matches, H, stat
114     else:
115         return None
116
117 # getting homography matrix for each pair of images
118 # ideally, to reduce computation, we would only need to compute this step 3 times but
119 # to see the matches between all the images, I am getting H matrix for all the pairs
120 matches12, HomographyMat12, status12 = getHomography(kps1, kps2, features1, features2, matchesImg12, 4)
```

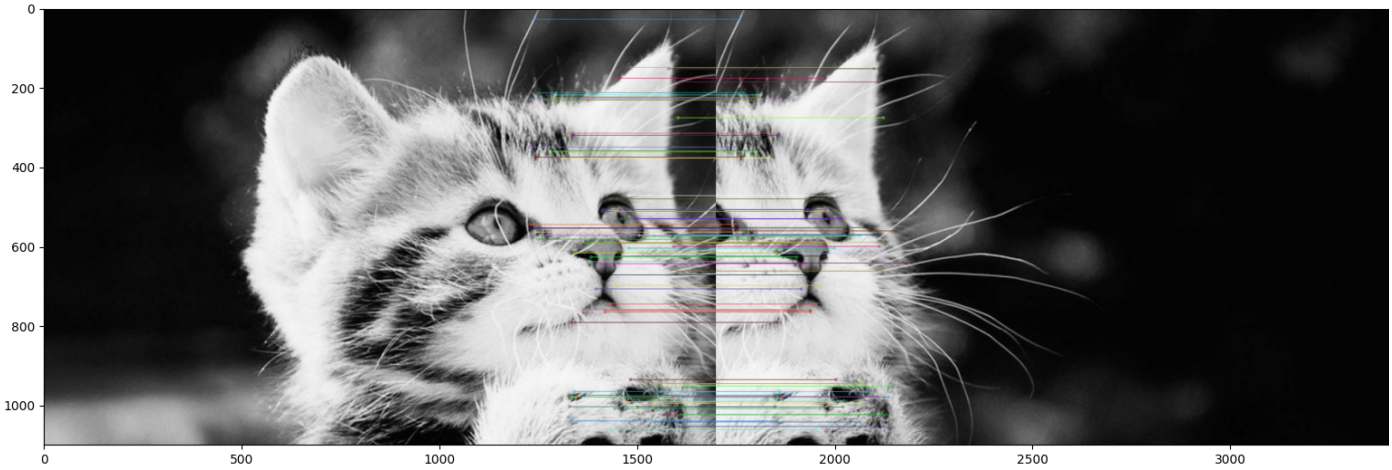
```
matches13, HomographyMat13, status13 = getHomography(kps1, kps3, features1, features3, matchesImg13, 4)
matches14, HomographyMat14, status14 = getHomography(kps1, kps4, features1, features4, matchesImg14, 4)
matches23, HomographyMat23, status23 = getHomography(kps2, kps3, features2, features3, matchesImg23, 4)
matches24, HomographyMat24, status24 = getHomography(kps2, kps4, features2, features4, matchesImg24, 4)
matches34, HomographyMat34, status34 = getHomography(kps3, kps4, features3, features4, matchesImg34, 4)
# print(f'Homography Matrix_12 is: \n{HomographyMat12}')
# print(f'Homography Matrix_13 is: \n{HomographyMat13}')
# print(f'Homography Matrix_14 is: \n{HomographyMat14}')
# print(f'Homography Matrix_23 is: \n{HomographyMat23}')
# print(f'Homography Matrix_24 is: \n{HomographyMat24}')
# print(f'Homography Matrix_34 is: \n{HomographyMat34}')


# using H matrix, this function is warping two images together to create panaroma
def warpImages(image1, image2, H_Mat):
    rows1, cols1 = image1.shape[:2]
    rows2, cols2 = image2.shape[:2]

    list_of_points_1 = np.float32([[0, 0], [0, rows1], [cols1, rows1], [cols1, 0]]).reshape(-1, 1, 2)
    temp_points = np.float32([[0, 0], [0, rows2], [cols2, rows2], [cols2, 0]]).reshape(-1, 1, 2)

    # When we have established a homography we need to warp perspective
    # Change field of view
    list_of_points_2 = cv2.perspectiveTransform(temp_points, H_Mat)

    list_of_points = np.concatenate((list_of_points_1, list_of_points_2), axis=0)

    [x_min, y_min] = np.int32(list_of_points.min(axis=0).ravel() - 0.5)
    [x_max, y_max] = np.int32(list_of_points.max(axis=0).ravel() + 0.5)
    translation_dist = [-x_min, -y_min]

    H_translation = np.array([[1, 0, translation_dist[0]], [0, 1, translation_dist[1]], [0, 0, 1]])
    # print(H_translation)

    outputImg = cv2.warpPerspective(image1, H_translation.dot(H_Mat), (x_max - x_min, y_max - y_min))
    outputImg[translation_dist[1]:rows1 + translation_dist[1], translation_dist[0]:cols1 + translation_dist[0]] = image2

    return outputImg

# calling warpImages to warp each pair of image (ideally, we would only need 2 but I wanted to see the results)
warpImages12 = warpImages(img1, img2, HomographyMat12)
warpImages13 = warpImages(img1, img3, HomographyMat13)
warpImages14 = warpImages(img1, img4, HomographyMat14)
warpImages23 = warpImages(img2, img3, HomographyMat23)
warpImages24 = warpImages(img2, img4, HomographyMat24)
warpImages34 = warpImages(img3, img4, HomographyMat34)

# plt.subplot(231), plt.imshow(warpImages12, cmap='gray', vmin=0, vmax=255)
# plt.subplot(232), plt.imshow(warpImages13, cmap='gray', vmin=0, vmax=255)
# plt.subplot(233), plt.imshow(warpImages14, cmap='gray', vmin=0, vmax=255)
# plt.subplot(234), plt.imshow(warpImages23, cmap='gray', vmin=0, vmax=255)
# plt.subplot(235), plt.imshow(warpImages24, cmap='gray', vmin=0, vmax=255)
# plt.subplot(236), plt.imshow(warpImages34, cmap='gray', vmin=0, vmax=255)
# plt.show()

# finally, creating full image using top 2 warped images and bottom 2 warped images
kps12, features12 = detectAndDescribe(warpImages12)
kps34, features34 = detectAndDescribe(warpImages34)
matchesImg1234 = bruteForceMatcher(features12, features34)
matches1234, HomographyMat1234, status1234 = getHomography(kps12, kps34, features12, features34, matchesImg1234, 4)
warpImages1234 = warpImages(warpImages12, warpImages34, HomographyMat1234)
plt.imshow(warpImages1234, cmap='gray', vmin=0, vmax=255)
```

```
plt.show()

# cv2.imshow("image 1", img1)
# cv2.imshow("image 2", img2)
# cv2.imshow("image 3", img3)
# cv2.imshow("image 4", img4)
# cv2.waitKey(0)
```

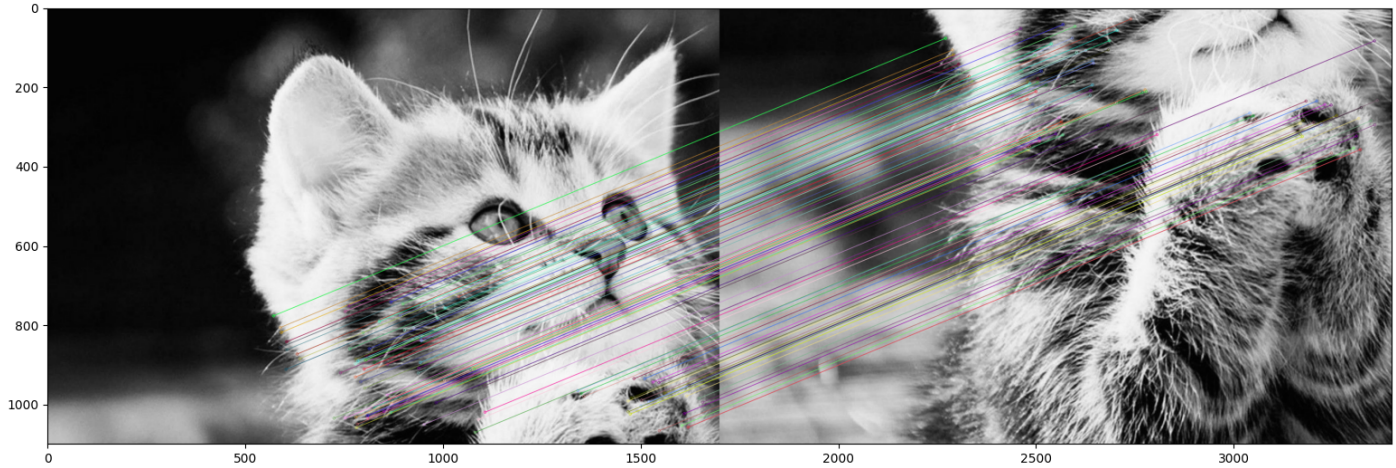
Result:

Keypoints in the images (shows gradient strength and orientation):

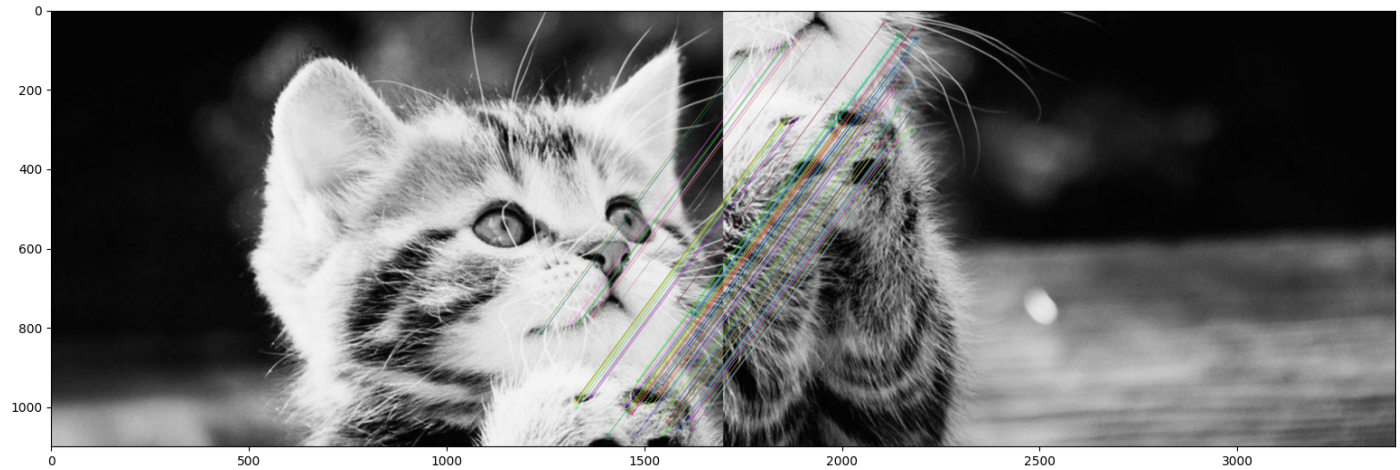
Matches between image 1 and image 2:



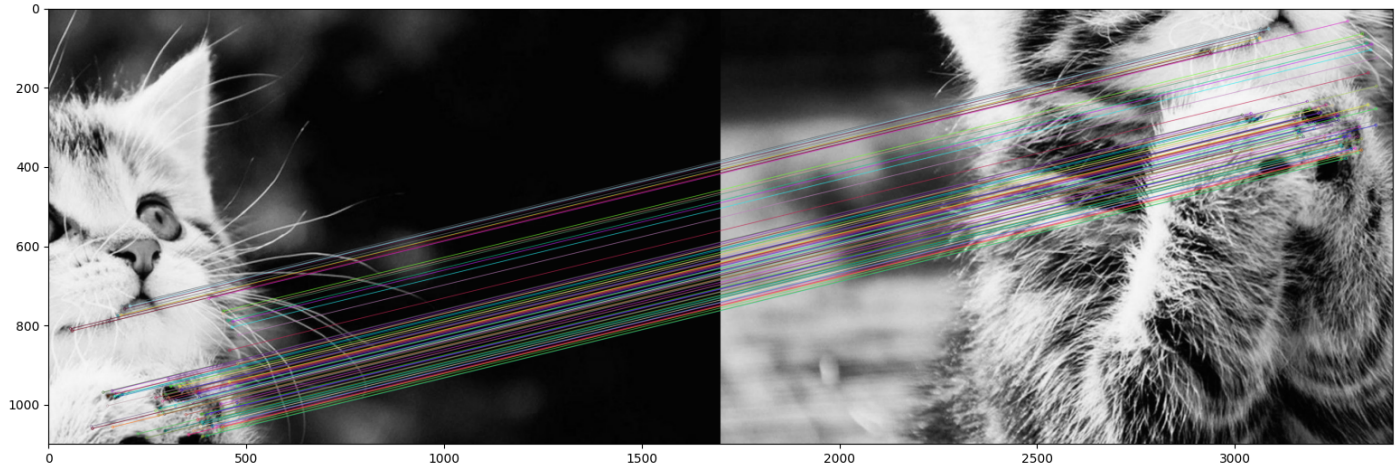
Matches between image 1 and image 3:



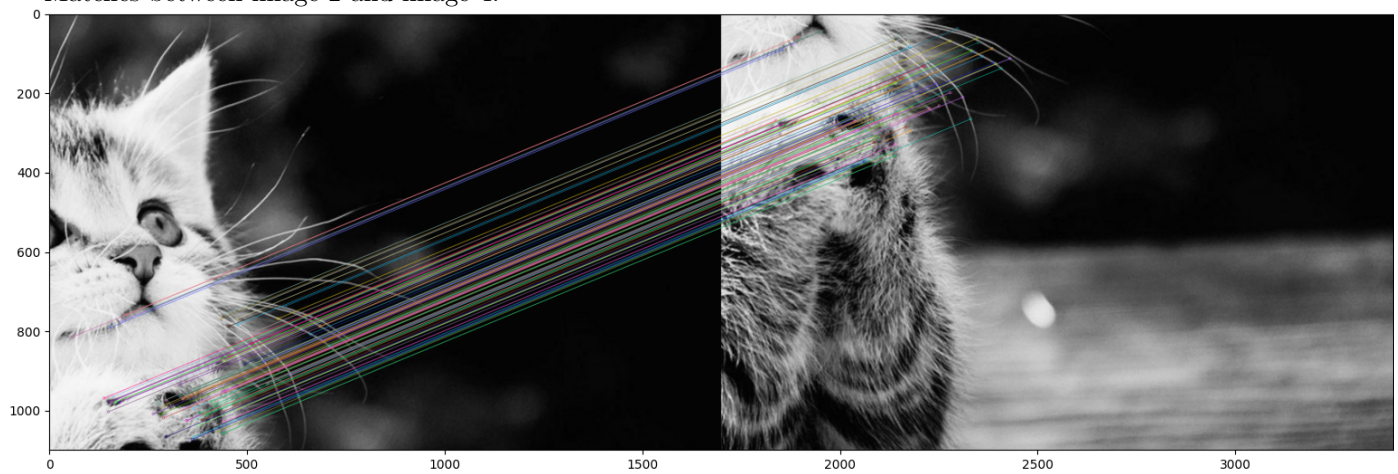
Matches between image 1 and image 4:



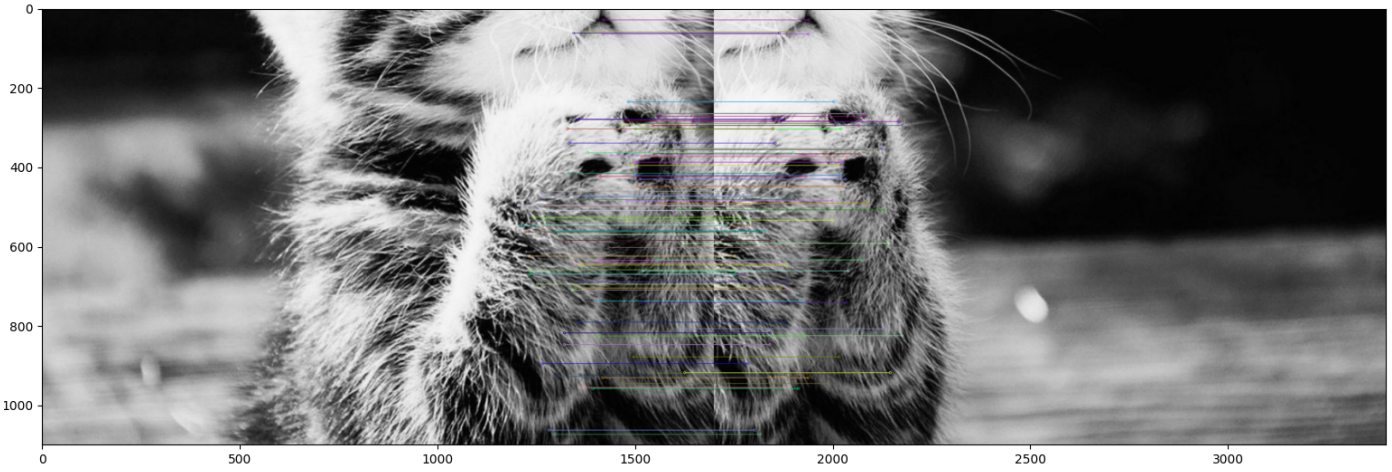
Matches between image 2 and image 3:



Matches between image 2 and image 4:



Matches between image 3 and image 4:



Homography Matrix (H):

$$H_{12} = \begin{bmatrix} 1.00006463e+00 & -2.53351371e-06 & -1.18007342e+03 \\ 3.88840132e-05 & 1.00004471e+00 & -4.72390490e-02 \\ 4.74556029e-08 & -1.32746089e-08 & 1.00000000e+00 \end{bmatrix}$$

$$H_{13} = \begin{bmatrix} 9.99972680e-01 & -4.39309980e-05 & 3.75483174e-02 \\ -1.13062094e-05 & 9.99981073e-01 & -6.99976699e+02 \\ 8.59973193e-09 & -4.17591350e-08 & 1.00000000e+00 \end{bmatrix}$$

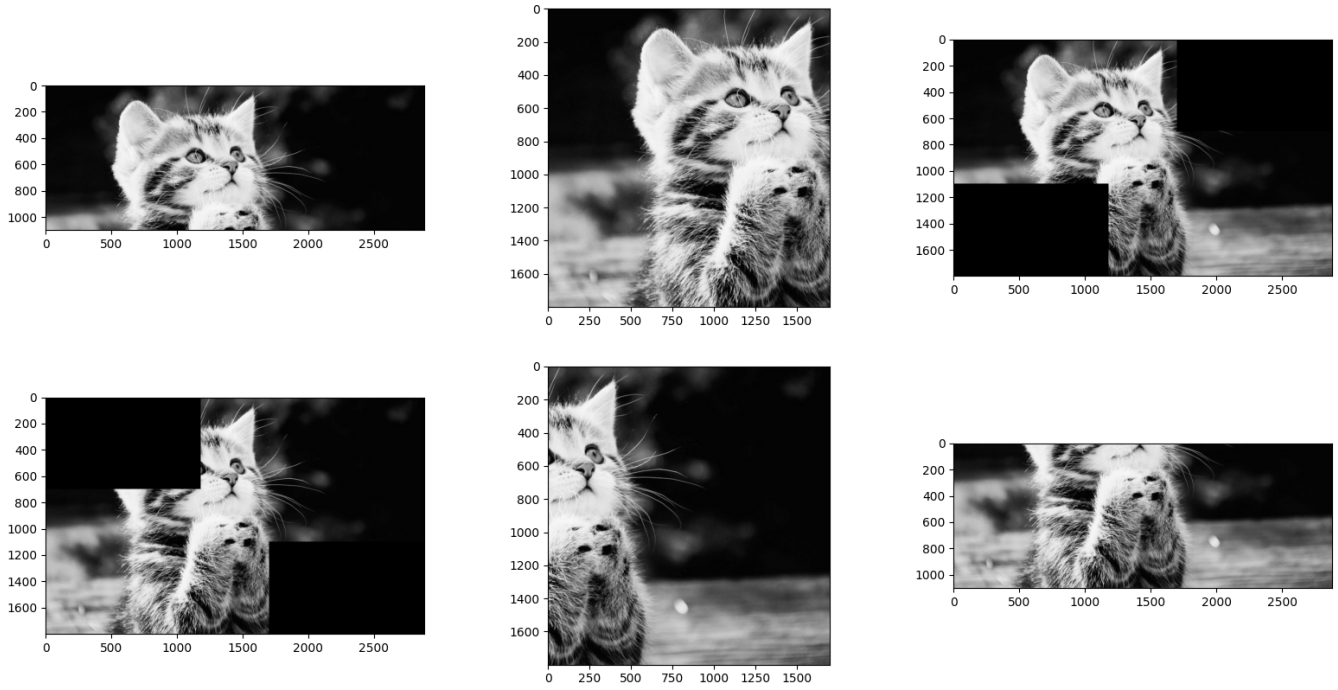
$$H_{14} = \begin{bmatrix} 9.99485738e-01 & -1.30823957e-04 & -1.17926697e+03 \\ -3.32953056e-05 & 9.99430180e-01 & -6.99543274e+02 \\ -4.03583389e-08 & -4.67207395e-07 & 1.00000000e+00 \end{bmatrix}$$

$$H_{23} = \begin{bmatrix} 1.00002346e+00 & -5.73905826e-04 & 1.18011905e+03 \\ -9.31654268e-06 & 9.99700882e-01 & -6.99784090e+02 \\ 2.17604428e-07 & -3.68362200e-07 & 1.00000000e+00 \end{bmatrix}$$

$$H_{24} = \begin{bmatrix} 9.99626071e-01 & -1.54443223e-04 & 1.48956797e-01 \\ -1.67312653e-05 & 9.99584978e-01 & -6.99697162e+02 \\ 2.91136716e-08 & -3.97028779e-07 & 1.00000000e+00 \end{bmatrix}$$

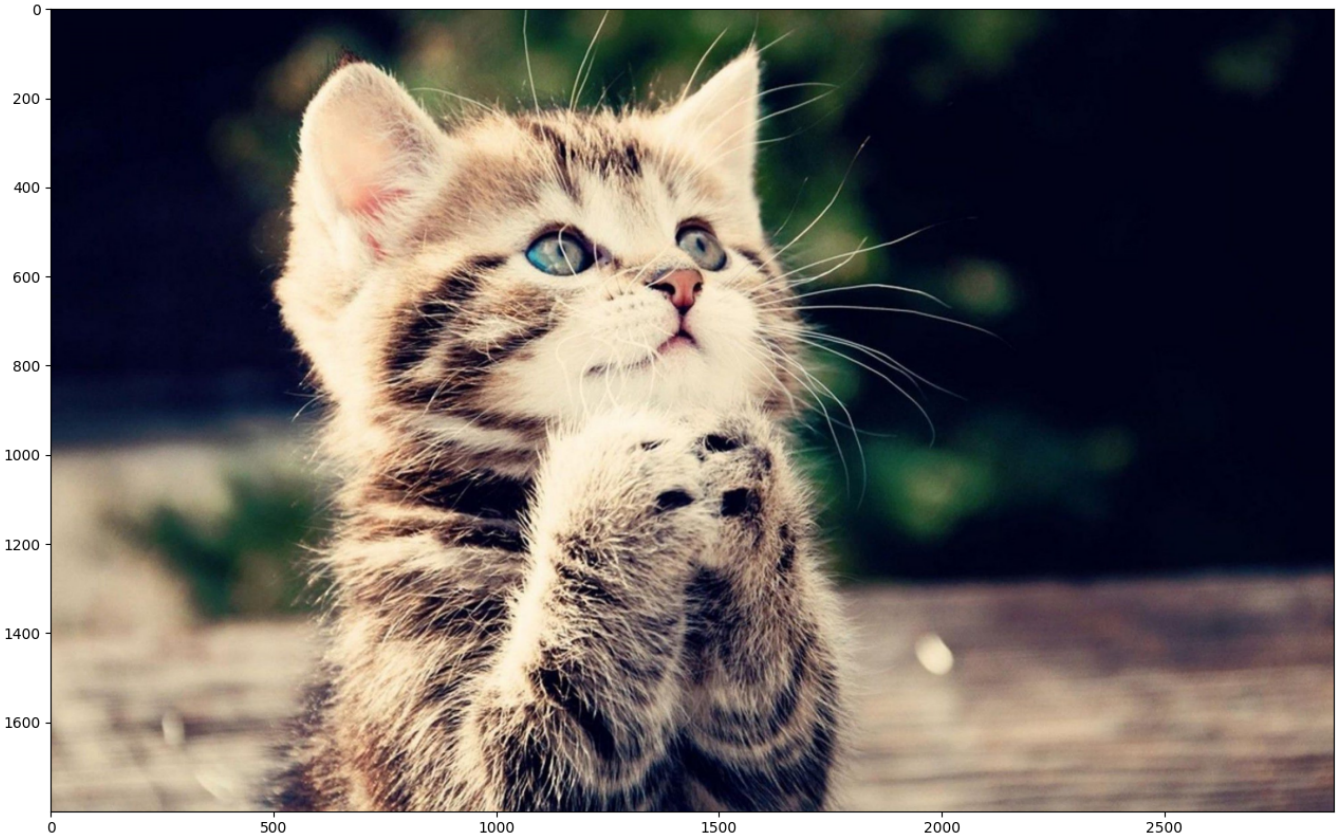
$$H_{34} = \begin{bmatrix} 1.00000916e+00 & -3.51708906e-07 & -1.18001038e+03 \\ 7.09275628e-06 & 1.00002836e+00 & -1.77596052e-02 \\ -1.42634066e-09 & 2.93407570e-08 & 1.00000000e+00 \end{bmatrix}$$

Following shows the transformation between each pair of image:



Following shows the final result between warped images (panaroma) of top two images and bottom two images:
In grayscale:

In color:


0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500

Bibliography

[1] Introduction to sift (scale-invariant feature transform)). https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html.

[2] Opencv projects – how to extract features from the image in python? https://datahacker.rs/004-opencv-projects-how-to-extract-features-from-the-image-in-python/.

[3] Orb (oriented fast and rotated brief). https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html.