Assignment 2

Index No.: 200022X

Name: AMANA M.A.N.

GitHub: [Nusrath-Amana/EN3160-Image-Processing-and-Machine-Vision (github.com)]

Question1

It computes the Laplacian of Gaussian images with successively increasing standard deviation and stacks them up in a cube.

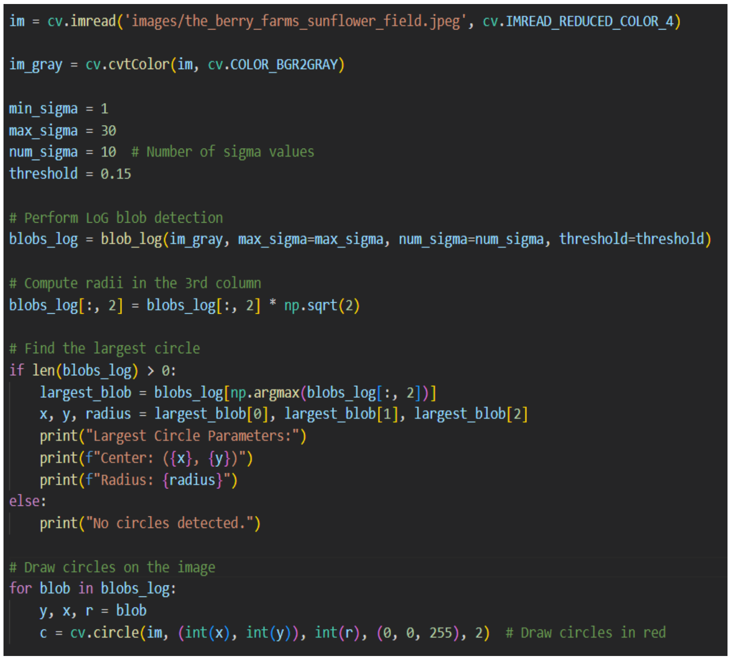
- Sigma is ranged between (1, 30) with 10 sigma values.

Blobs are local maximas in this cube. It searches for local extrema in the LoG response across these different scales to identify and locate blobs of varying sizes in the image.

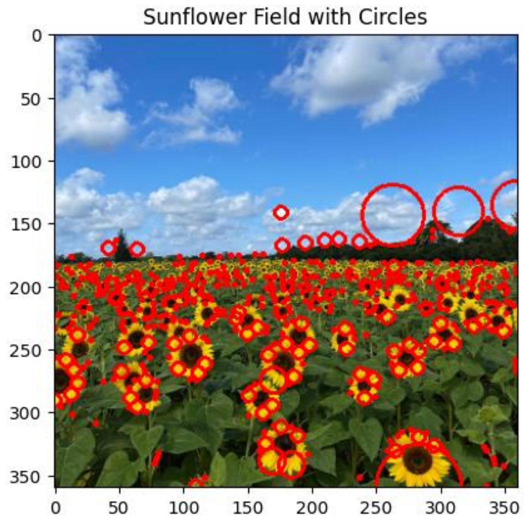
- threshold filters out weak blob responses and retains stronger blobs.
- To get maximum response, the zeros of the Laplacian have to be aligned with the circle
  The Laplacian is given by (up to scale):

$$(x^2 + y^2 - 2\sigma^2) \exp(- (x^2 + y^2) / 2\sigma^2)$$

Therefore, the maximum response occurs at $r = \sqrt{2}\,\sigma$



```python
im = cv.imread('images/the_berry_farms_sunflower_field.jpeg', cv.IMREAD_REDUCED_COLOR_4)

im_gray = cv.cvtColor(im, cv.COLOR_BGR2GRAY)

min_sigma = 1
max_sigma = 30
num_sigma = 10  # Number of sigma values
threshold = 0.15

# Perform LoG blob detection
blobs_log = blob_log(im_gray, max_sigma=max_sigma, num_sigma=num_sigma, threshold=threshold)

# Compute radii in the 3rd column
blobs_log[:, 2] = blobs_log[:, 2] * np.sqrt(2)

# Find the largest circle
if len(blobs_log) > 0:
    largest_blob = blobs_log[np.argmax(blobs_log[:, 2])]
    x, y, radius = largest_blob[0], largest_blob[1], largest_blob[2]
    print("Largest Circle Parameters:")
    print(f"Center: ({x}, {y})")
    print(f"Radius: {radius}")
else:
    print("No circles detected.")

# Draw circles on the image
for blob in blobs_log:
    y, x, r = blob
    c = cv.circle(im, (int(x), int(y)), int(r), (0, 0, 255), 2)  # Draw circles in red
```



Largest Circle Parameters:

- Center: (359.0, 283.0)
- Radius: 33.31258613589958

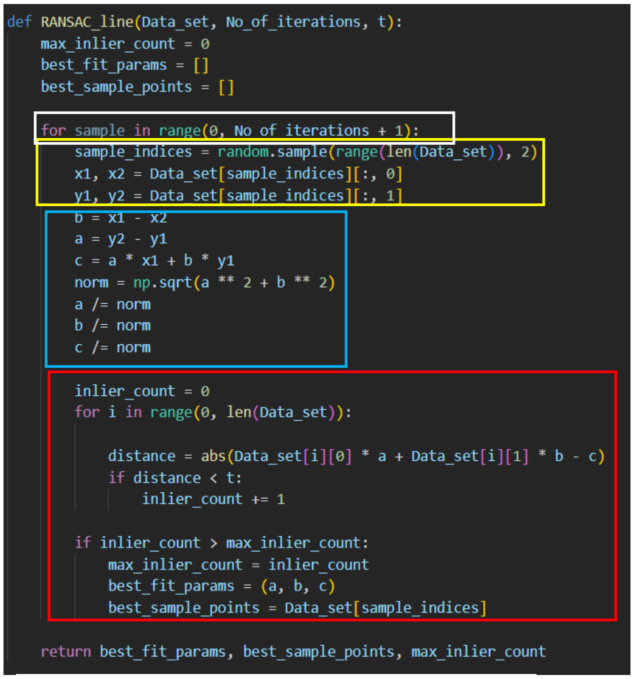
Question 2

The RANSAC algorithm is implemented with 4 steps.

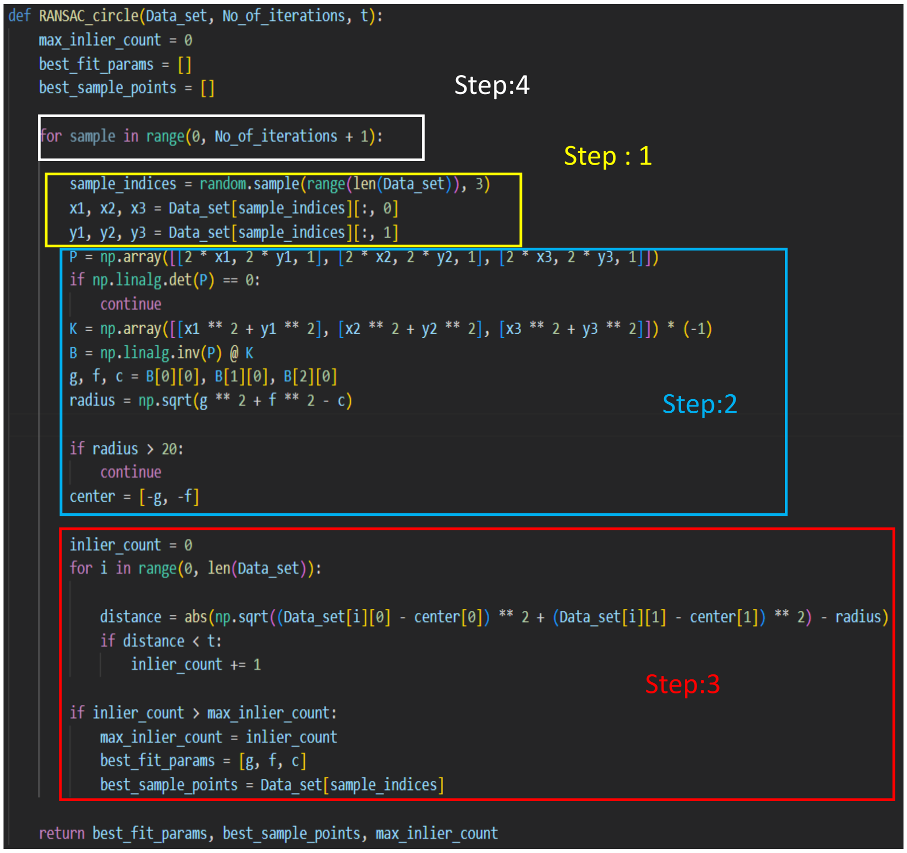
1. Randomly choose a small initial subset of points (2 points for a line and 4 points for a circle)
2. Fit a model to that subset and find model parameters.
   - Line : a , b , c : line equation is a.x + b.y = c
   - Circle : centre (-g,-f) and radius
3. Find the inliers by comparing the error with a threshold value; the model with maximum inliers is considered the best model.
   - Fpr line, error-the normal distance to the estimated line
   - For circle , error -radial error

➢ Both threshold values are set to 1, to achieve most fitted model.

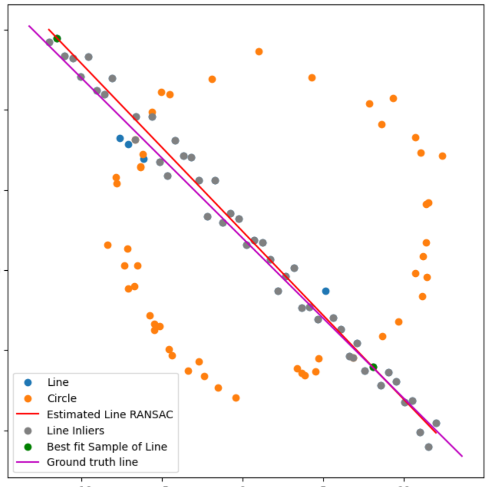
4. Do this many times and choose the model with the most inliers ; 10,000 iterations



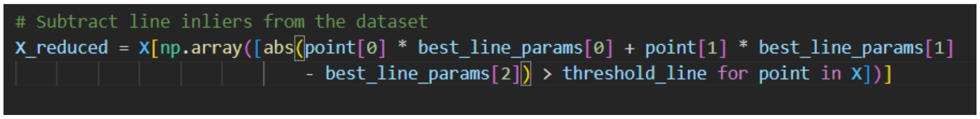
```python
def RANSAC_line(Data_set, No_of_iterations, t):
    max_inlier_count = 0
    best_fit_params = []
    best_sample_points = []

    for sample in range(0, No_of_iterations + 1):
        sample_indices = random.sample(range(len(Data_set)), 2)
        x1, x2 = Data_set[sample_indices][:, 0]
        y1, y2 = Data_set[sample_indices][:, 1]
        b = x1 - x2
        a = y2 - y1
        c = a * x1 + b * y1
        norm = np.sqrt(a ** 2 + b ** 2)
        a /= norm
        b /= norm
        c /= norm

        inlier_count = 0
        for i in range(0, len(Data_set)):

            distance = abs(Data_set[i][0] * a + Data_set[i][1] * b - c)
            if distance < t:
                inlier_count += 1

        if inlier_count > max_inlier_count:
            max_inlier_count = inlier_count
            best_fit_params = (a, b, c)
            best_sample_points = Data_set[sample_indices]

    return best_fit_params, best_sample_points, max_inlier_count
```



```python
def RANSAC_circle(Data_set, No_of_iterations, t):
    max_inlier_count = 0
    best_fit_params = []
    best_sample_points = []

    for sample in range(0, No_of_iterations + 1):

        sample_indices = random.sample(range(len(Data_set)), 3)
        x1, x2, x3 = Data_set[sample_indices][:, 0]
        y1, y2, y3 = Data_set[sample_indices][:, 1]
        P = np.array([[2 * x1, 2 * y1, 1], [2 * x2, 2 * y2, 1], [2 * x3, 2 * y3, 1]])
        if np.linalg.det(P) == 0:
            continue
        K = np.array([[x1 ** 2 + y1 ** 2], [x2 ** 2 + y2 ** 2], [x3 ** 2 + y3 ** 2]]) * (-1)
        B = np.linalg.inv(P) @ K
        g, f, c = B[0][0], B[1][0], B[2][0]
        radius = np.sqrt(g ** 2 + f ** 2 - c)

        if radius > 20:
            continue
        center = [-g, -f]

        inlier_count = 0
        for i in range(0, len(Data_set)):

            distance = abs(np.sqrt((Data_set[i][0] - center[0]) ** 2 + (Data_set[i][1] - center[1]) ** 2) - radius)
            if distance < t:
                inlier_count += 1

        if inlier_count > max_inlier_count:
            max_inlier_count = inlier_count
            best_fit_params = [g, f, c]
            best_sample_points = Data_set[sample_indices]

    return best_fit_params, best_sample_points, max_inlier_count
```

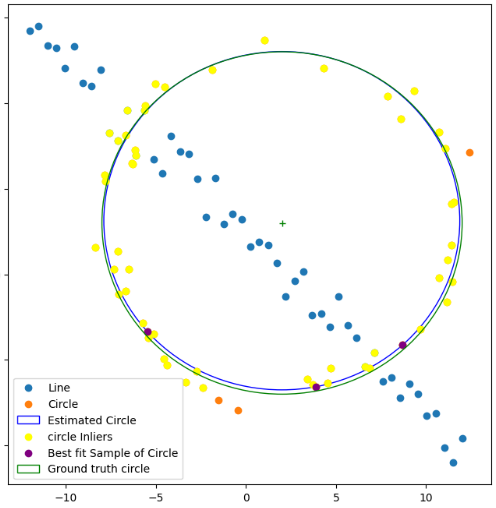


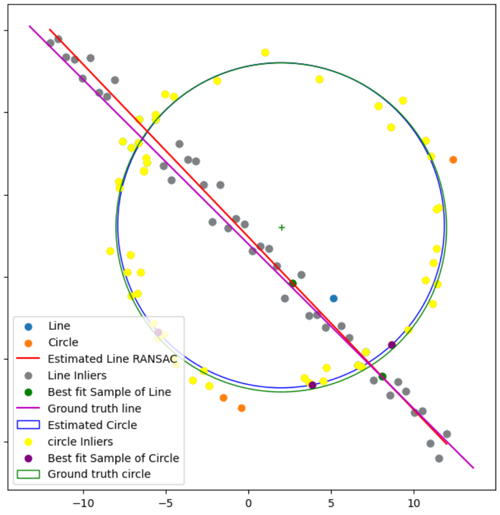
Before obtaining circle, the consensus of the best line are subtracted.



```python
# Subtract line inliers from the dataset
X_reduced = X[np.array([abs(point[0] * best_line_params[0] + point[1] * best_line_params[1]
                            - best_line_params[2]) > threshold_line for point in X])]
```

Since Ground truth line is obtained with only the line samples, while RANSAC line is obtained with both line and circle samples there is a little difference between Ground truth and RANSAC



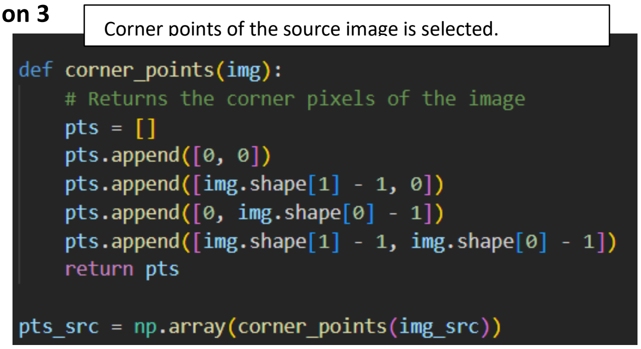


if we fit the circle first,

line also can be detected as a part of circle with large radius.

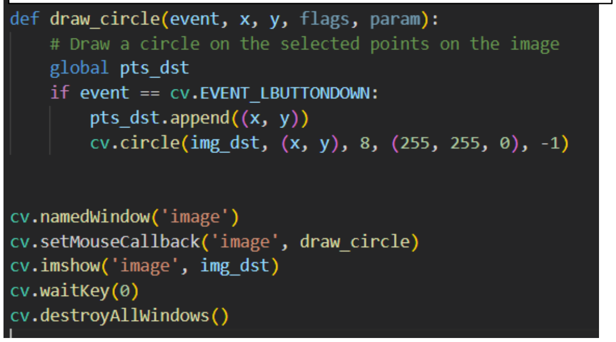
Question 3

Corner points of the source image is selected.



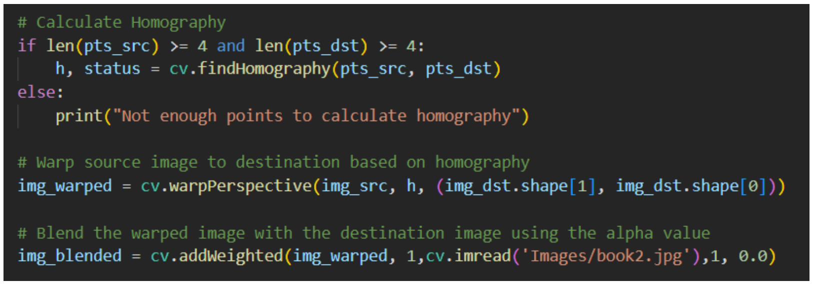
```python
def corner_points(img):
    # Returns the corner pixels of the image
    pts = []
    pts.append([0, 0])
    pts.append([img.shape[1] - 1, 0])
    pts.append([0, img.shape[0] - 1])
    pts.append([img.shape[1] - 1, img.shape[0] - 1])
    return pts

pts_src = np.array(corner_points(img_src))
```

4 points of destination image are selected by mouse click.

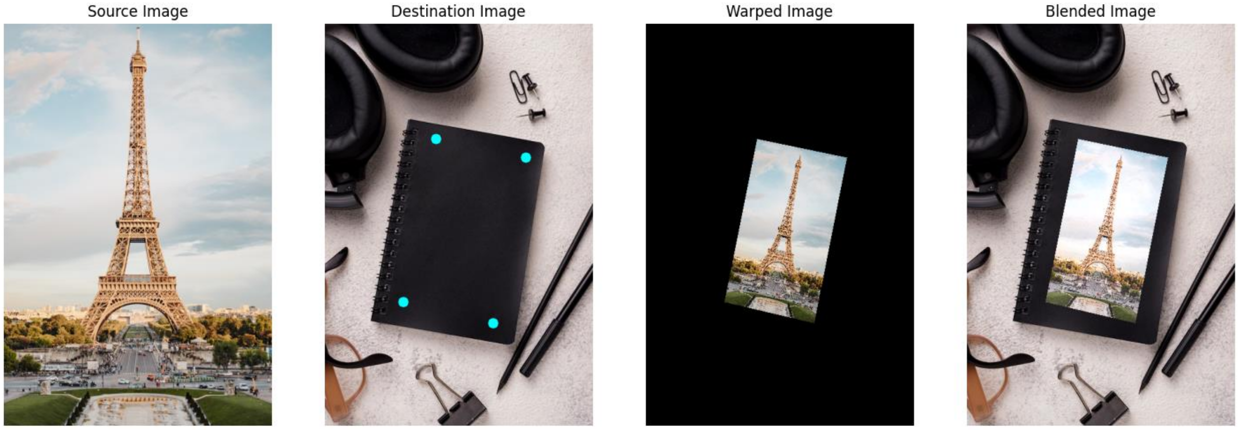


```python
def draw_circle(event, x, y, flags, param):
    # Draw a circle on the selected points on the image
    global pts_dst
    if event == cv.EVENT_LBUTTONDOWN:
        pts_dst.append((x, y))
        cv.circle(img_dst, (x, y), 8, (255, 255, 0), -1)


cv.namedWindow('image')
cv.setMouseCallback('image', draw_circle)
cv.imshow('image', img_dst)
cv.waitKey(0)
cv.destroyAllWindows()
```

Homography was created between 2 images and wrapped and blended.



```python
# Calculate Homography
if len(pts_src) >= 4 and len(pts_dst) >= 4:
    h, status = cv.findHomography(pts_src, pts_dst)
else:
    print("Not enough points to calculate homography")

# Warp source image to destination based on homography
img_warped = cv.warpPerspective(img_src, h, (img_dst.shape[1], img_dst.shape[0]))

# Blend the warped image with the destination image using the alpha value
img_blended = cv.addWeighted(img_warped, 1,cv.imread('Images/book2.jpg'),1, 0.0)
```



Source Image

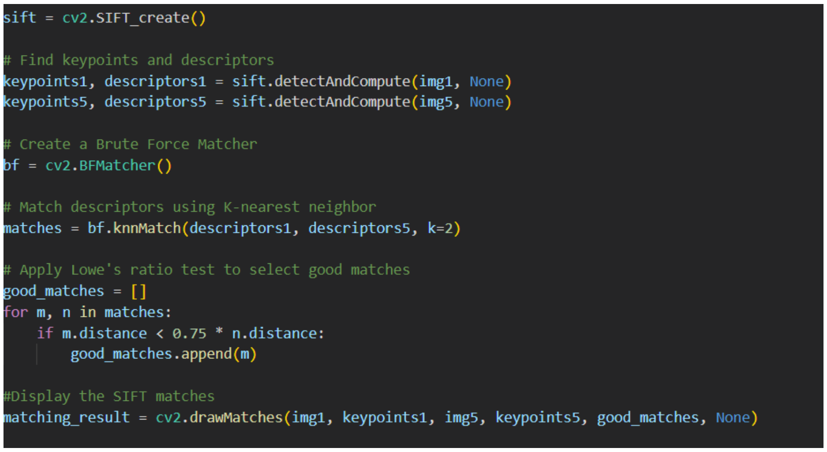
Destination Image

Warped Image

Blended Image

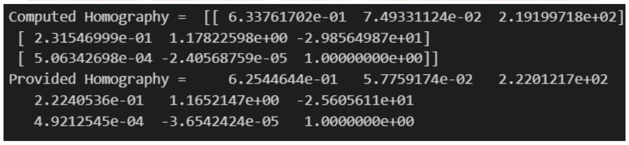
A picture of the Eiffel Tower is used as the source image and a book with a black cover as the destination image, resulting in a book cover featuring the Eiffel Tower This technique can be used to make creative images.

Question 4



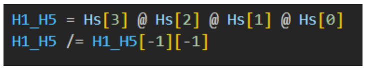
```python
sift = cv2.SIFT_create()

# Find keypoints and descriptors
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints5, descriptors5 = sift.detectAndCompute(img5, None)

# Create a Brute Force Matcher
bf = cv2.BFMatcher()

# Match descriptors using K-nearest neighbor
matches = bf.knnMatch(descriptors1, descriptors5, k=2)

# Apply Lowe's ratio test to select good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good_matches.append(m)

#Display the SIFT matches
matching_result = cv2.drawMatches(img1, keypoints1, img5, keypoints5, good_matches, None)
```

SIFT object named sift is used for detecting keypoints and computing descriptors.. The knnMatch function was used to compute the matching between the features with k = 2. Then ratio test is applied to filter out good matches.here rtio is set to 0.75



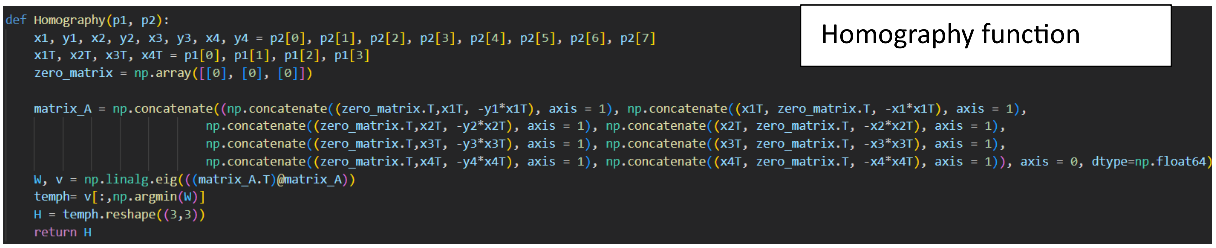
```
Computed Homography =  [[ 6.33761702e-01  7.49331124e-02  2.19199718e+02]
 [ 2.31546999e-01  1.17822598e+00 -2.98564987e+01]
 [ 5.06342698e-04 -2.40568759e-05  1.00000000e+00]]
Provided Homography =      6.2544644e-01   5.7759174e-02   2.2201217e+02
   2.2240536e-01   1.1652147e+00  -2.5605611e+01
   4.9212545e-04  -3.6542424e-05   1.0000000e+00
```

When clearly observed there are many inaccurate mathes in SIFT feature. Therefore, the approach used was to calculate homographies for img1-img2, img2-img3, img3-img4 and img4-img5 sequentially and multiplying each homography to obtain the final homography between img1 and img5.



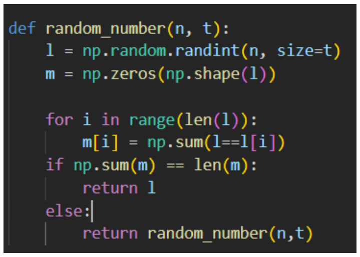
```
H1_H5 = Hs[3] @ Hs[2] @ Hs[1] @ Hs[0]
H1_H5 /= H1_H5[-1][-1]
```

Calculated homography and provided homography are almost same.



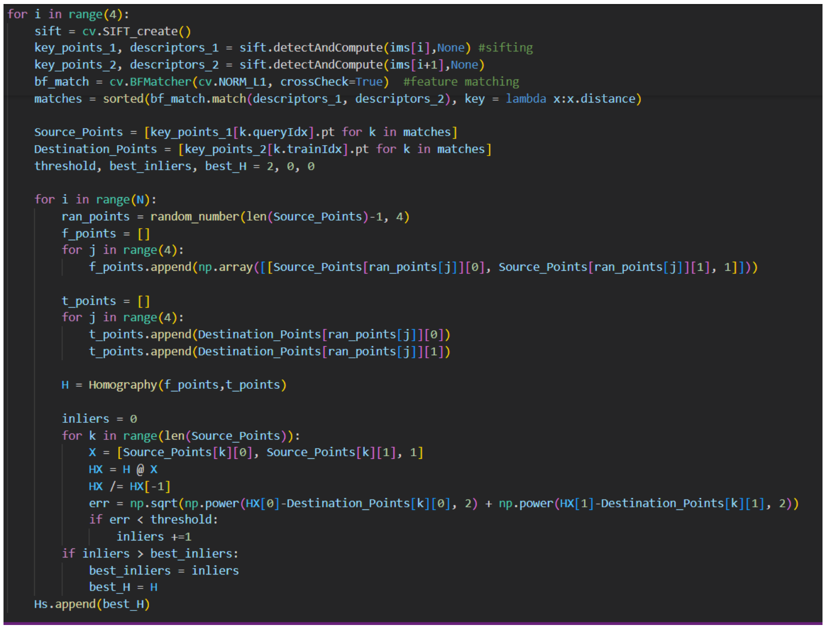
Homography function

```
def Homography(p1, p2):
    x1, y1, x2, y2, x3, y3, x4, y4 = p2[0], p2[1], p2[2], p2[3], p2[4], p2[5], p2[6], p2[7]
    x1T, x2T, x3T, x4T = p1[0], p1[1], p1[2], p1[3]
    zero_matrix = np.array([[0], [0], [0]])

    matrix_A = np.concatenate((np.concatenate((zero_matrix.T,x1T, -y1*x1T), axis = 1), np.concatenate((x1T, zero_matrix.T, -x1*x1T), axis = 1),
                               np.concatenate((zero_matrix.T,x2T, -y2*x2T), axis = 1), np.concatenate((x2T, zero_matrix.T, -x2*x2T), axis = 1),
                               np.concatenate((zero_matrix.T,x3T, -y3*x3T), axis = 1), np.concatenate((x3T, zero_matrix.T, -x3*x3T), axis = 1),
                               np.concatenate((zero_matrix.T,x4T, -y4*x4T), axis = 1), np.concatenate((x4T, zero_matrix.T, -x4*x4T), axis = 1)), axis = 0, dtype=np.float64)
    W, v = np.linalg.eig(((matrix_A.T)@matrix_A))
    temph= v[:,np.argmin(W)]
    H = temph.reshape((3,3))
    return H
```

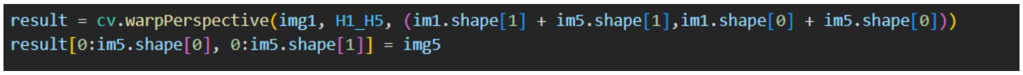


```
def random_number(n, t):
    l = np.random.randint(n, size=t)
    m = np.zeros(np.shape(l))

    for i in range(len(l)):
        m[i] = np.sum(l==l[i])
    if np.sum(m) == len(m):
        return l
    else:
        return random_number(n,t)
```

RANSAC algorithm



```
for i in range(4):
    sift = cv.SIFT_create()
    key_points_1, descriptors_1 = sift.detectAndCompute(ims[i],None) #sifting
    key_points_2, descriptors_2 = sift.detectAndCompute(ims[i+1],None)
    bf_match = cv.BFMatcher(cv.NORM_L1, crossCheck=True)  #feature matching
    matches = sorted(bf_match.match(descriptors_1, descriptors_2), key = lambda x:x.distance)

    Source_Points = [key_points_1[k.queryIdx].pt for k in matches]
    Destination_Points = [key_points_2[k.trainIdx].pt for k in matches]
    threshold, best_inliers, best_H = 2, 0, 0

    for i in range(N):
        ran_points = random_number(len(Source_Points)-1, 4)
        f_points = []
        for j in range(4):
            f_points.append(np.array([[Source_Points[ran_points[j]][0], Source_Points[ran_points[j]][1], 1]]))

        t_points = []
        for j in range(4):
            t_points.append(Destination_Points[ran_points[j]][0])
            t_points.append(Destination_Points[ran_points[j]][1])

        H = Homography(f_points,t_points)

        inliers = 0
        for k in range(len(Source_Points)):
            X = [Source_Points[k][0], Source_Points[k][1], 1]
            HX = H @ X
            HX /= HX[-1]
            err = np.sqrt(np.power(HX[0]-Destination_Points[k][0], 2) + np.power(HX[1]-Destination_Points[k][1], 2))
            if err < threshold:
                inliers +=1
        if inliers > best_inliers:
            best_inliers = inliers
            best_H = H
    Hs.append(best_H)
```



```
result = cv.warpPerspective(img1, H1_H5, (im1.shape[1] + im5.shape[1],im1.shape[0] + im5.shape[0]))
result[0:im5.shape[0], 0:im5.shape[1]] = img5
```