Tharindu O.K.D.
190622R
GitHub Repository (click here)

Question 01

```python
def RANSAC_circle(points, threshold):
    n = points.shape[0] # number of points
    best_estimate = None; best_sample = None
    max_inliers = []
    p = 0.99; e = 0.5; N = np.log(1 - p) / np.log(1 - e**(n/10))
    for i in range(int(N)):
        random_index = np.random.randint(n, size=3) # get 3 random indexes
        pt1, pt2, pt3 = points[random_index] # get 3 points using random indexes
        A = np.array([[pt2[0] - pt1[0], pt2[1] - pt1[1]], [pt3[0] - pt2[0], pt3[1] - pt2[1]]])
        B = np.array([[pt2[0]**2 - pt1[0]**2 + pt2[1]**2 - pt1[1]**2], [pt3[0]**2 - pt2[0]**2 + pt3[1]**2 - pt2[1]**2]])
        C = np.linalg.pinv(A) @ B / 2
        a, b = C.flatten() # center cordinates of the selected circle
        r = np.sqrt((a - pt1[0])**2 + (b - pt1[1])**2) # calculating radius
        inliers = []
        for x, y in points: # to check each point with calculated circle.
            distance = np.sqrt((x - a)**2 + (y - b)**2)
            if (np.abs(distance - r) < threshold): inliers.append([x,y])# check wheather the point is lied within the threshold value
        if (len(inliers) > len(max_inliers)): best_estimate = (a, b, r); best_sample = random_index; max_inliers = inliers
    return best_estimate, max_inliers, best_sample
best_estimate, inliers, best_sample = RANSAC_circle(X, 1.96*sigma) # get the best estimate circle parameters
a, b, r = best_estimate[0], best_estimate[1], best_estimate[2]; inliers = np.array(inliers)
P = np.concatenate((inliers, np.ones((inliers.shape[0], 1))), axis=1) # Least Square
Q = inliers[:,[0]]**2 + inliers[:, [1]]**2
B = np.linalg.pinv(P.T @ P) @ P.T @ Q
a0, b0, c = B.flatten(); a0, b0 = a0/2, b0/2 # parameters of the best fitting circle
r0 = np.sqrt(a0**2 + b0**2 + c)
```

Figure 1: Question 01 Code

Our objective for this is to robustly fit a circle to our point set. This can be done using the RANSAC algorithm to find the best inlier set and then fit a circle to this inlier set. Figure 1 shows the code snipped of the RANSAC function. Approach of this method is shown below.

1. First, we calculate the number of iterations N required to get a better estimation. We choose N so that, with probability p, at least one random sample is free from outliers. It is given by, $N = \frac{\log(1-p)}{\log(1-(1-e)^n)}$. Here e is outlier ratio and n is a number of points. Here I took $e = 0.5$ and $p = 0.99$ so that 50% of points will fall into inliers with a 99% probability.
2. We randomly select 3 points from the given data set. Then we calculate the fitting circle using the cartesian equation of the circle $(x-a)^2 + (y-b)^2 = r^2$. Substituting 3 point coordinates and putting it into matrix form, we get

$$\begin{pmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_2 & y_3 - y_2 \end{pmatrix} \begin{pmatrix} 2a \\ 2b \end{pmatrix} = \begin{pmatrix} x_2^2 - x_1^2 & y_2^2 - y_1^2 \\ x_3^2 - x_2^2 & y_3^2 - y_2^2 \end{pmatrix}$$

   Then we can find center coordinates (a, b) and radius(r) by solving above matrix equation.
3. Then determine the inlier set $= \{(x_i, y_i) \in points \text{ s.t } |\sqrt{(x_i - a)^2 + (y_i - b)^2} - r| < threshold\}$
4. If $|inliers| > |max\_inliers|$, we select it as the best estimate. Go to step 2 until iterate N times.

First, we get the parameters of the best fitting circle using the above $RANSAC\_circle$ function. Since the sample noise is distributed as zero-mean Gaussian, we pass the threshold value as 1.96σ so that will give a $\approx 95\%$ probability of capturing all inliers. After finding the best inliers, we find the best fitting circle for those inliers using the Least Square approach as follows.

$$\begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{pmatrix} \begin{pmatrix} 2a_0 \\ 2b_0 \\ c \end{pmatrix} = \begin{pmatrix} x_1^2 & y_1^2 \\ x_2^2 & y_2^2 \\ \vdots & \vdots \\ x_n^2 & y_n^2 \end{pmatrix}$$

$$PB = Q$$

$$B = (P^T P)^{-1} P^T Q$$

Using the B matrix, we can find a_0, b_0, and r_0 which are the parameters of the best-fitting circle.

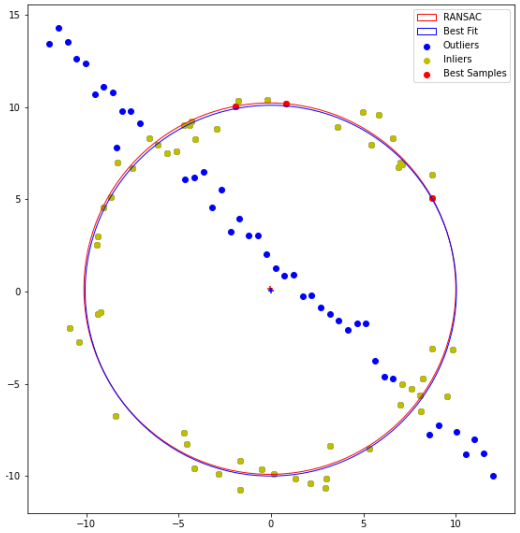

Figure 2: Output

Question 02

```python
def find_hormograpy(src_pts, dst_pts):
    n = src_pts.shape[0] # get the number of matches
    A = np.empty((2*n, 9))
    for i in range(n): # appending corresponding value to A matrix
        A[2*i] = [src_pts[i][0], src_pts[i][1], 1, 0, 0, 0, -dst_pts[i][0]*src_pts[i][0], -dst_pts[i][0]*src_pts[i][1], -dst_pts[i][0]]
        A[2*i+1] = [0, 0, 0, src_pts[i][0], src_pts[i][1], 1, -dst_pts[i][1]*src_pts[i][0], -dst_pts[i][1]*src_pts[i][1], -dst_pts[i][1]]
    W, V = np.linalg.eig(A.T @ A) # get the eigen values and eigen vectors of the A.T @ A
    ev_corresponding_to_smalest_ev = V[:, np.argmin(W)] # get the eigenvector associated with smallest eigenvalue
    return ev_corresponding_to_smalest_ev.reshape((3, 3)) # return the 3x3 homography matrix
Y = np.empty((0,2))
def click_event(event, x, y, flags, params): # define the clicking event
    global Y
    if event == cv.EVENT_LBUTTONDOWN and len(Y) < 4:
        Y = np.vstack((Y, [x, y])); font = cv.FONT_HERSHEY_SIMPLEX # draw small circle on click point
        cv.circle(img,(x,y), 5,(0,0,255),-1); cv.putText(img, "P{}".format(len(Y)), (x+10,y+10), font, 1, (255, 255, 255), 2); cv.imshow('image', img)
img = cv.imread('002.jpg') # get background image
img_copy = img.copy() # keep a copy of the img to blend the flag
cv.imshow('image', img); cv.setMouseCallback('image', click_event); cv.waitKey(0); cv.destroyAllWindows()
uk_flag = cv.imread('uk_flag.png') # get the flag
shape = uk_flag.shape
X = np.array([[0, 0], [shape[1]-1, 0], [0, shape[0]-1], [shape[1]-1, shape[0]-1]]) # get the four corner's cordinates of the flag
H = find_hormograpy(X, Y) # find the homography corresponding to X to Y
uk_transformed = cv.warpPerspective(uk_flag, H, (img.shape[1], img.shape[0])) # warp the flag onto according to calculated H
blend_img = cv.addWeighted(img_copy, 1, uk_transformed, 0.4, 0) # Add the backgroud image with the transformed image with 1:0.4 ratio
```

Figure 3: Question 02 Code

The objective of this question is to superimpose the flag into the given background image.

1. First, we find the coordinates of the selected points (Y) using mousing clicking and find the coordinates of the four corners(X) of the flag. Then we feed these source points (X) and destination points (Y) to the *find_homography* function as shown in Figure 3.
2. Although, we only need four correspondences to find the homography matrix, in this function, if we give more than four points it will calculate the homography matrix using the least square approach ($\because$ this will be required for question 03). This will not be a problem because if we give the exact four correspondences, it will return the exact homography matrix. In the *find_homography* function, we put each coordinate into matrix form as shown below.

$$\begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_s^{(i)} & y_s^{(i)} & 1 & 0 & 0 & 0 & -x_d^{(i)}x_s^{(i)} & -x_d^{(i)}y_s^{(i)} & x_d^{(i)} \\ 0 & 0 & 0 & x_s^{(i)} & y_s^{(i)} & 1 & -y_d^{(i)}x_s^{(i)} & -y_d^{(i)}y_s^{(i)} & y_d^{(i)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} = \begin{pmatrix} \vdots \\ 0 \\ \vdots \end{pmatrix}$$

$$Ah = 0$$

Since H is defined up to a scale value, we can solve this as a Constrained Least Square problem. i.e. h is the eigenvector associated with the smallest eigenvalue of the matrix $A^T A$. Then we can find the H matrix using h vector.

3. After finding the homography matrix, we can use the OpenCV warpPerspective method to warp the flag into selected points. To blend the two images, I have used the OpenCV addWeighted method which will add the two images with a given ratio.

Figure 4 shows selected points in the order and the resulting image for the UK flag and Sri Lanka flag.

Figure 4: Output

Question 03

```python
def ransac_homography(src_pts, dst_pts):
    best_inliers = []; best_H = None; n = len(src_pts)
    for i in range(100):
        random_index = np.random.randint(n, size=4); random_src_pts = src_pts[random_index]; random_dst_pts = dst_pts[random_index]
        H = find_hormograpy(random_src_pts, random_dst_pts) # find the homography matrix using above random 4 points
        inliers = []
        for j in range(n):
            est = H @ np.hstack((src_pts[j], 1)); est = (1/est[2])*est[:2] # get the mapping point for calculated H
            error = np.linalg.norm(est - dst_pts[j]) # get the norm of the difference vector
            if error < 10: inliers.append(j)
        if len(inliers) > len(best_inliers):
            best_inliers = inliers; best_H = H
        if (len(best_inliers) > n*0.5):break # if there are 50% inliers, we take it as the best estimate
    return best_inliers, H
while len(max_inliers) < 50:
    pre_H = np.array([ [1, 0, 0], [0 , 1,  0], [ 0, 0, 1]], dtype=np.float64); pre_H = 1e-4 * np.random.randn(3,3) * pre_H # generate random homography
    pre_img1 = cv.warpPerspective(img1, pre_H,(1000,1000)) # apply random homography to img1
    keypoint_1, descriptor_1 = sift.detectAndCompute(pre_img1, None); keypoint_2, descriptor_2 = sift.detectAndCompute(img2, None) # get keypoints.
    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5); search_params = dict(checks=100)
    flann = cv.FlannBasedMatcher(index_params,search_params); matches = flann.knnMatch(descriptor_1,descriptor_2,k=2)
    good = []
    for m,n in matches:
        if m.distance < 0.75*n.distance: good.append(m) # get the best matching features
    src_pts = np.float32([ keypoint_1[m.queryIdx].pt for m in good ]); dst_pts = np.float32([ keypoint_2[m.trainIdx].pt for m in good ])
    if (len(src_pts)>4): # if the matches are less than 4 we cannot find a homography.
        inl, h = ransac_homography(src_pts, dst_pts) # get the best estimate homography for given pre_img
        if (len(inl) > len(max_inliers)):
            max_inliers = inl; best_H = pre_H; sr = src_pts[max_inliers]; ds = dst_pts[max_inliers]
H_ = find_hormograpy(sr, ds) # find the best fitting homography
H = H_ @ best_H # get the effective homography
img3 = cv.warpPerspective(img1, H, (img2.shape[1], img2.shape[0])) # appling warp perspective with effective homography
print(H/H[2][2])
```

Figure 5: Code

Figure 6: Match SIFT Features

I have used a FLANN-based matcher to match SIFT features rather than using a Brutal-Force matcher because it gives very accurate and more matches. You can see that in Figure 6, even though we used the FLANN-based matcher, we only get a fewer number of SIFT matches and lots of them are incorrect. This is happen because img5 is taken more parallel to the wall. Therefore, the img5 is more compact in the horizontal

direction than img1. Therefore, lots of features are missed and incorrect. So, if we directly calculate the homography using these matches, we will get a wrong homography matrix.

To overcome this problem, I have used the following method.

1. First I apply a random pre-homography transformation to img1. Since we do not need any translation, I force it to be zero and also force vector v of the homography matrix to be zero.
2. Apply the warp perspective transformation to img1 with the above random pre-homography transformation. Then match the SIFT features of the resultant pre-transformed image and img5.
3. Then apply the ransac_homography function with these matching points. It first randomly picks 4 correspondences and finds the corresponding homography matrix using the find_homography function defined in question 02. Then apply the homography to each source point and calculate the vector norm with the destination point. If the norm value is less than the threshold value, we append this to the inlier set. After iterating 100 times, it returns the best homography matrix with inliers. I constrained the iteration time to 100 because some random warp perspective may not have any matches.
4. Repeat the above 3 steps until we find the pre-homography which gives more than 50 inliers. Then calculate the best fitting homography using the find_homography function (As mentioned in question 02, this function is capable of finding homography using the least square method).
5. After finding the best homography for the pre-transformed image to img5, we can calculate the effective homography using the above pre-homography and calculated homography.

$$img1 \xrightarrow{H_{21}} pre\_transformed\_image \xrightarrow{H_{32}} img5$$

$$img1 \xrightarrow{H_{31}=H_{32}H_{21}} img5$$

$$calculatedHomography = \begin{pmatrix} 6.14979996e-01 & 5.93985823e-02 & 2.21696291e+02 \\ 2.15344442e-01 & 1.15345300e+00 & -2.28382302e+01 \\ 4.74622983e-04 & -4.59124680e-05 & 1.00000000e+00 \end{pmatrix}$$

$$givenHomography = \begin{pmatrix} 6.2544644e-01 & 5.7759174e-02 & 2.2201217e+02 \\ 2.2240536e-01 & 1.1652147e+00 & -2.5605611e+01 \\ 4.9212545e-04 & -3.6542424e-05 & 1.0000000e+00 \end{pmatrix}$$

As you can see, calculated homography and given homography are almost the same. Therefore we can confirm that this method performs well rather than directly matching SIFT features. After finding the warp perspective image, I have used 3 methods to blend the image. First, I directly add the resultant image to img5 with 50-50% weight. As you can see the common area is more highlighted. In the second method, I cut the perspective of the img1 from img5 and then add the two. This gives a better result but we can still see the seams of the joining lines. Lastly, rather than directly cutting the perspective, I applied a distance transformed mask to the resultant image and then add the two images. You can see this gives a better result than the other two methods.

Figure 7: Output