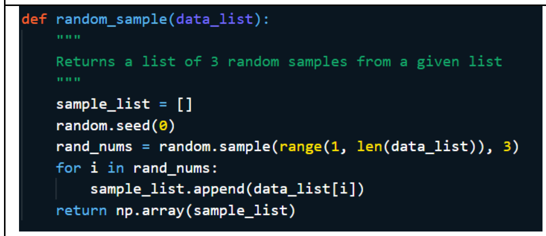
EN 2550 – Assignment 2

Index No.: 190696U
Name: C.H.W. Wijegunawardana
GitHub: https://github.com/chathuni1999/EN2550.git

Question 1

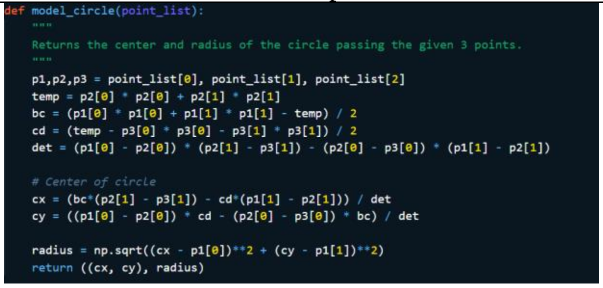
The algorithm followed to compute the RANSAC approximation of the circle that fits a set of points is as follows.

Step 1 - Create samples of three points from the dataset.



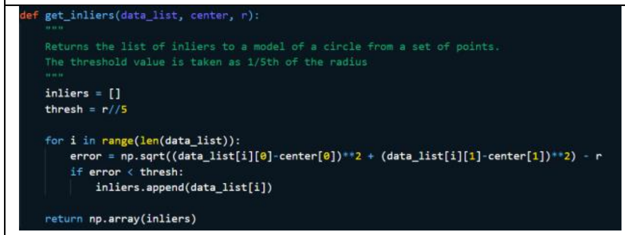
```python
def random_sample(data_list):
    """
    Returns a list of 3 random samples from a given list
    """
    sample_list = []
    random.seed(0)
    rand_nums = random.sample(range(1, len(data_list)), 3)
    for i in rand_nums:
        sample_list.append(data_list[i])
    return np.array(sample_list)
```

Step 2 - Calculate the center and radius of the circle for the selected three points.



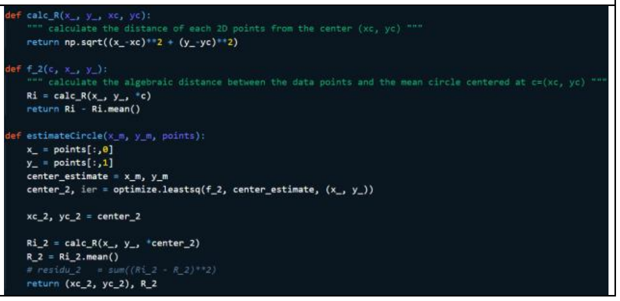
```python
def model_circle(point_list):
    """
    Returns the center and radius of the circle passing the given 3 points.
    """
    p1,p2,p3 = point_list[0], point_list[1], point_list[2]
    temp = p2[0] * p2[0] + p2[1] * p2[1]
    bc = (p1[0] * p1[0] + p1[1] * p1[1] - temp) / 2
    cd = (temp - p3[0] * p3[0] - p3[1] * p3[1]) / 2
    det = (p1[0] - p2[0]) * (p2[1] - p3[1]) - (p2[0] - p3[0]) * (p1[1] - p2[1])

    # Center of circle
    cx = (bc*(p2[1] - p3[1]) - cd*(p1[1] - p2[1])) / det
    cy = ((p1[0] - p2[0]) * cd - (p2[0] - p3[0]) * bc) / det

    radius = np.sqrt((cx - p1[0])**2 + (cy - p1[1])**2)
    return ((cx, cy), radius)
```

Step 3 – Get the set of inliers for that circle with a threshold of radius/5.
This threshold was selected by trial and error to get the best fit.



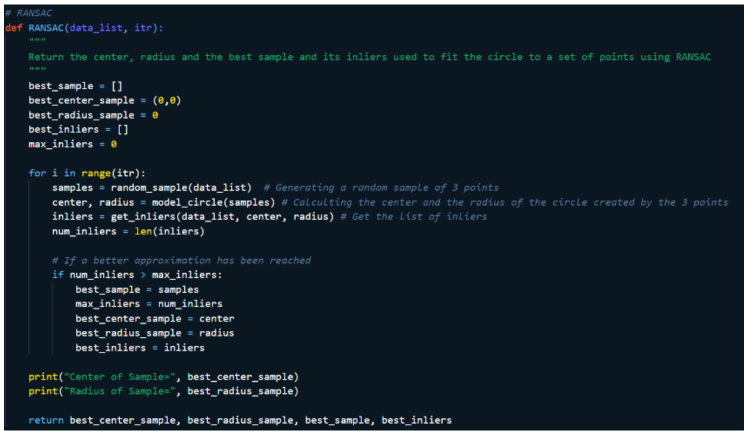
```python
def get_inliers(data_list, center, r):
    """
    Returns the list of inliers to a model of a circle from a set of points.
    The threshold value is taken as 1/5th of the radius
    """
    inliers = []
    thresh = r//5

    for i in range(len(data_list)):
        error = np.sqrt((data_list[i][0]-center[0])**2 + (data_list[i][1]-center[1])**2) - r
        if error < thresh:
            inliers.append(data_list[i])

    return np.array(inliers)
```

Step 4 – Iterate for a fixed number of times (10,000 in this implementation).
Find the sample with the maximum number of inliers.
Estimate the circle for the selected set of inliers.



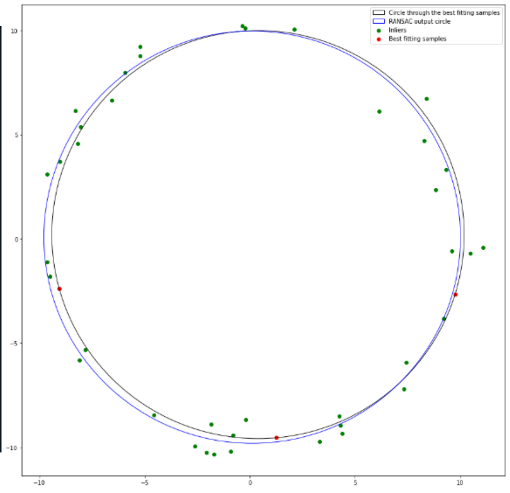
```python
def calc_R(x_, y_, xc, yc):
    """ calculate the distance of each 2D points from the center (xc, yc) """
    return np.sqrt((x_-xc)**2 + (y_-yc)**2)

def f_2(c, x_, y_):
    """ calculate the algebraic distance between the data points and the mean circle centered at c=(xc, yc) """
    Ri = calc_R(x_, y_, *c)
    return Ri - Ri.mean()

def estimateCircle(x_m, y_m, points):
    x_ = points[:,0]
    y_ = points[:,1]
    center_estimate = x_m, y_m
    center_2, ier = optimize.leastsq(f_2, center_estimate, (x_, y_))

    xc_2, yc_2 = center_2

    Ri_2 = calc_R(x_, y_, *center_2)
    R_2 = Ri_2.mean()
    # residu_2   = sum((Ri_2 - R_2)**2)
    return (xc_2, yc_2), R_2
```

Complete Code:

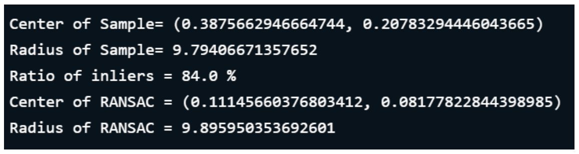


```python
# RANSAC
def RANSAC(data_list, itr):
    """
    Return the center, radius and the best sample and its inliers used to fit the circle to a set of points using RANSAC
    """
    best_sample = []
    best_center_sample = (0,0)
    best_radius_sample = 0
    best_inliers = []
    max_inliers = 0

    for i in range(itr):
        samples = random_sample(data_list)  # Generating a random sample of 3 points
        center, radius = model_circle(samples) # Calculting the center and the radius of the circle created by the 3 points
        inliers = get_inliers(data_list, center, radius) # Get the list of inliers
        num_inliers = len(inliers)

        # If a better approximation has been reached
        if num_inliers > max_inliers:
            best_sample = samples
            max_inliers = num_inliers
            best_center_sample = center
            best_radius_sample = radius
            best_inliers = inliers

    print("Center of Sample=", best_center_sample)
    print("Radius of Sample=", best_radius_sample)

    return best_center_sample, best_radius_sample, best_sample, best_inliers
```

Fitted Circle:



Results:

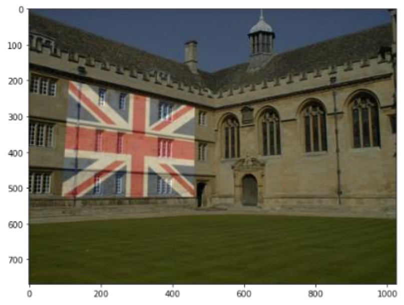


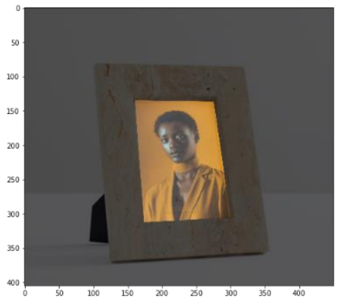
```
Center of Sample= (0.3875662946664744, 0.20783294446043665)
Radius of Sample= 9.79406671357652
Ratio of inliers = 84.0 %
Center of RANSAC = (0.11145660376803412, 0.08177822844398985)
Radius of RANSAC = 9.895950353692601
```

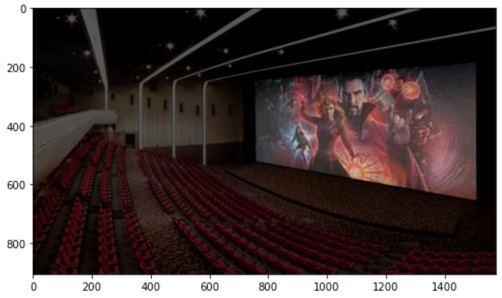
Observations:
RANSAC is a powerful algorithm that can estimate a circle or a line that fits a given set of points. A very satisfactory fit could be achieved for the provided dataset using the basic form of the RANSAC algorithm. This fitting could be improved by implementing the adoptive method where the threshold will be adjusted in each iteration. The centers and radii of a circle created by three points and a circle that estimates the set of inliers were based on algebraic methods and implemented using the *scipy.optimize* package.

Question 2

The following shows the results of blending a set of images with suitable other images.

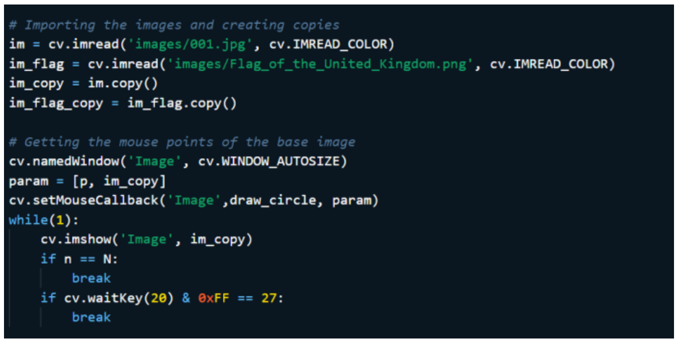






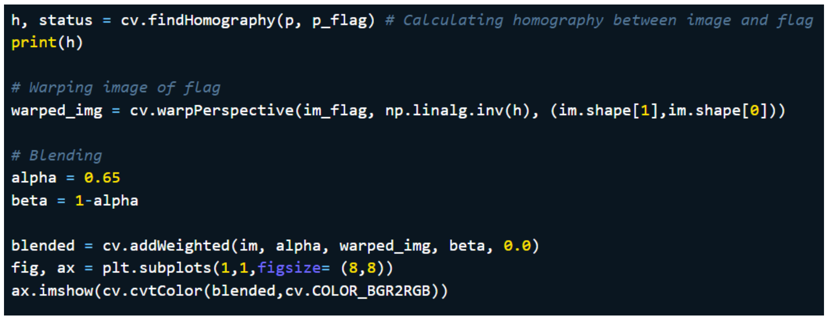
The mouse-points that should mapped together in both images were selected by the user through the following code.

The flag was positioned to appear like it has been hanged on the wall and not blocking any of the main doorways. Additional two sets of images were selected by own choice to demonstrate how two images could be combined to construct a creative output image. The first combination matches a portrait image with a photo frame while the second combination matches a movie poster with a theater screen.



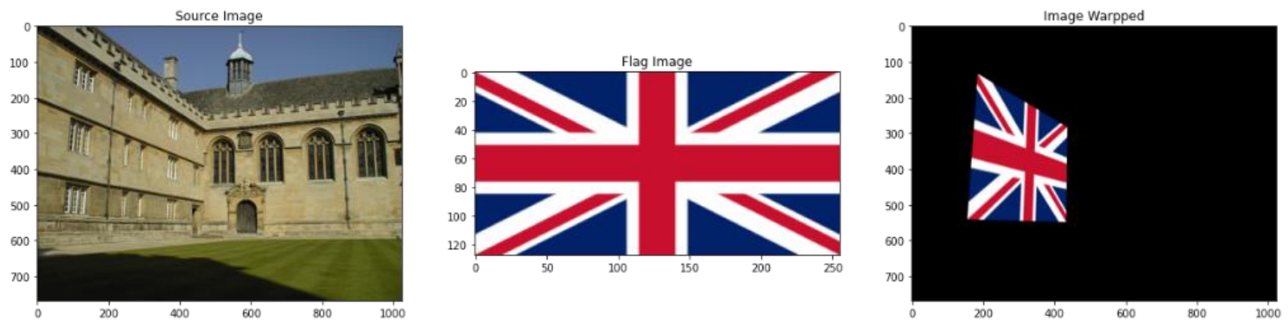
```python
# Importing the images and creating copies
im = cv.imread('images/001.jpg', cv.IMREAD_COLOR)
im_flag = cv.imread('images/Flag_of_the_United_Kingdom.png', cv.IMREAD_COLOR)
im_copy = im.copy()
im_flag_copy = im_flag.copy()

# Getting the mouse points of the base image
cv.namedWindow('Image', cv.WINDOW_AUTOSIZE)
param = [p, im_copy]
cv.setMouseCallback('Image',draw_circle, param)
while(1):
    cv.imshow('Image', im_copy)
    if n == N:
        break
    if cv.waitKey(20) & 0xFF == 27:
        break
```

Then, the homography was created between the images using *findHomography* function the OpenCV package, the flag image was warped and blended. The alpha value was adjusted for each image to provide a natural look to the images.



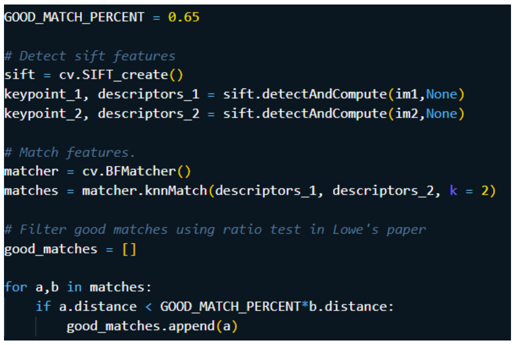
```
h, status = cv.findHomography(p, p_flag) # Calculating homography between image and flag
print(h)

# Warping image of flag
warped_img = cv.warpPerspective(im_flag, np.linalg.inv(h), (im.shape[1],im.shape[0]))

# Blending
alpha = 0.65
beta = 1-alpha

blended = cv.addWeighted(im, alpha, warped_img, beta, 0.0)
fig, ax = plt.subplots(1,1,figsize= (8,8))
ax.imshow(cv.cvtColor(blended,cv.COLOR_BGR2RGB))
```

The intermediate result of the above process is illustrated below.

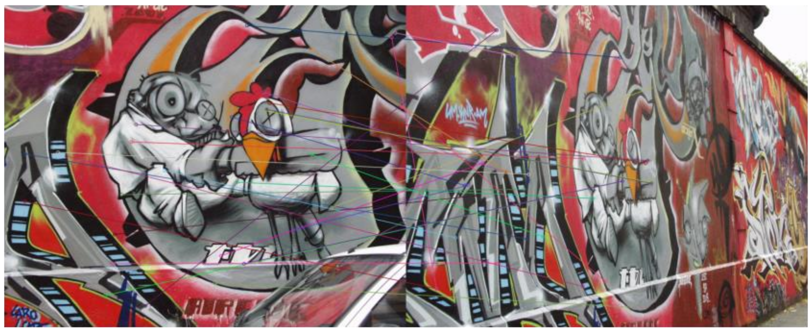


Question 3

a) The features of img1 and img5 of the graffiti images were detected using SIFT feature detection. The *knnMatch* function was used to compute the matching between the features with k = 2. Then, a variable was set to filter out the best matches based on the ratio test according to Lowe's paper. For this implementation, the ratio was selected through testing to be 0.65. The features were mapped using the above function for img1 and img5.

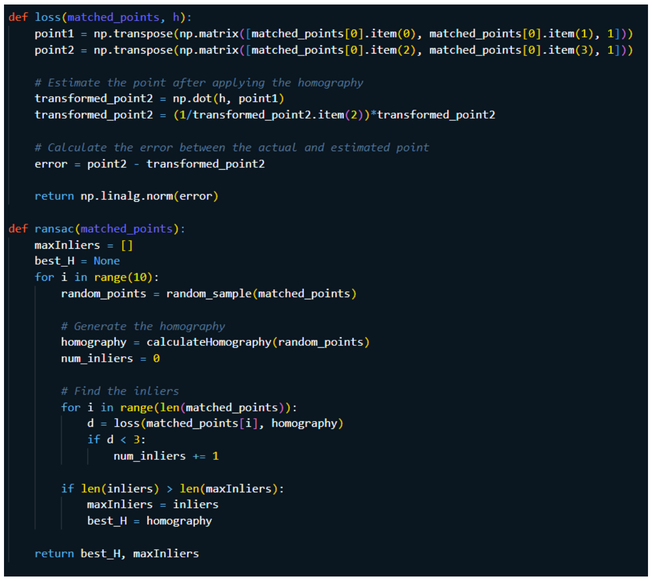


```
GOOD_MATCH_PERCENT = 0.65

# Detect sift features
sift = cv.SIFT_create()
keypoint_1, descriptors_1 = sift.detectAndCompute(im1,None)
keypoint_2, descriptors_2 = sift.detectAndCompute(im2,None)

# Match features.
matcher = cv.BFMatcher()
matches = matcher.knnMatch(descriptors_1, descriptors_2, k = 2)

# Filter good matches using ratio test in Lowe's paper
good_matches = []

for a,b in matches:
    if a.distance < GOOD_MATCH_PERCENT*b.distance:
        good_matches.append(a)
```



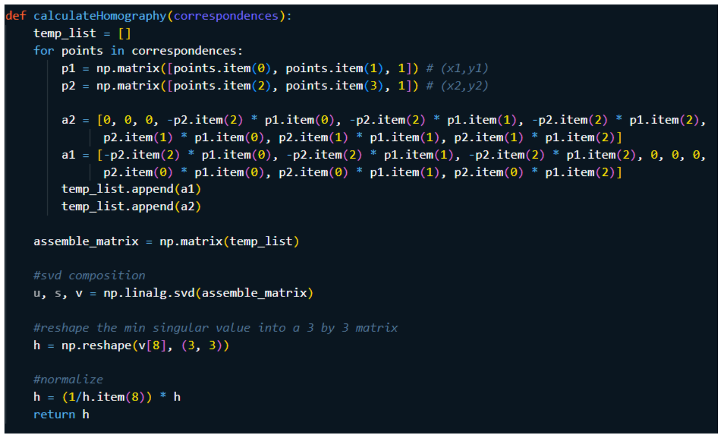
b) The above RANSAC code in Q1 was modified to generate the best homography between two images. The loss in estimation of homography was calculated by the distance between the estimated 2nd point and the actual 2nd point in the matched correspondences. The algorithm was run for 1000 iterations. It was realized that the mapped SIFT features between img1 and img5 are heavily inaccurate. The number of matched features was very low. 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.

The RANSAC code:



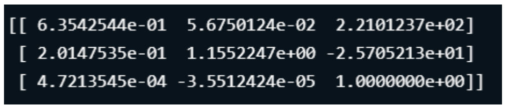
```
def loss(matched_points, h):
    point1 = np.transpose(np.matrix([matched_points[0].item(0), matched_points[0].item(1), 1]))
    point2 = np.transpose(np.matrix([matched_points[0].item(2), matched_points[0].item(3), 1]))

    # Estimate the point after applying the homography
    transformed_point2 = np.dot(h, point1)
    transformed_point2 = (1/transformed_point2.item(2))*transformed_point2

    # Calculate the error between the actual and estimated point
    error = point2 - transformed_point2

    return np.linalg.norm(error)

def ransac(matched_points):
    maxInliers = []
    best_H = None
    for i in range(10):
        random_points = random_sample(matched_points)

        # Generate the homography
        homography = calculateHomography(random_points)
        num_inliers = 0

        # Find the inliers
        for i in range(len(matched_points)):
            d = loss(matched_points[i], homography)
            if d < 3:
                num_inliers += 1

        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            best_H = homography

    return best_H, maxInliers
```

The homography code:



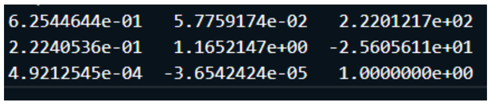
```
def calculateHomography(correspondences):
    temp_list = []
    for points in correspondences:
        p1 = np.matrix([points.item(0), points.item(1), 1]) # (x1,y1)
        p2 = np.matrix([points.item(2), points.item(3), 1]) # (x2,y2)

        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)]
        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)]
        temp_list.append(a1)
        temp_list.append(a2)

    assemble_matrix = np.matrix(temp_list)

    #svd composition
    u, s, v = np.linalg.svd(assemble_matrix)

    #reshape the min singular value into a 3 by 3 matrix
    h = np.reshape(v[8], (3, 3))

    #normalize
    h = (1/h.item(8)) * h
    return h
```

The final homography between img1 and img5:



```
[[ 6.3542544e-01  5.6750124e-02  2.2101237e+02]
 [ 2.0147535e-01  1.1552247e+00 -2.5705213e+01]
 [ 4.7213545e-04 -3.5512424e-05  1.0000000e+00]]
```

Homography in the dataset:



```
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
```

c) Img1 and img5 were stitched together using the above homography generated. Img5 was positioned over the warped img1. It was challenging to filter out the best matches as the mapped features between img1 and img5 were scarce. This could be solved by using other feature mapping techniques other than SIFT.

Stitched image: