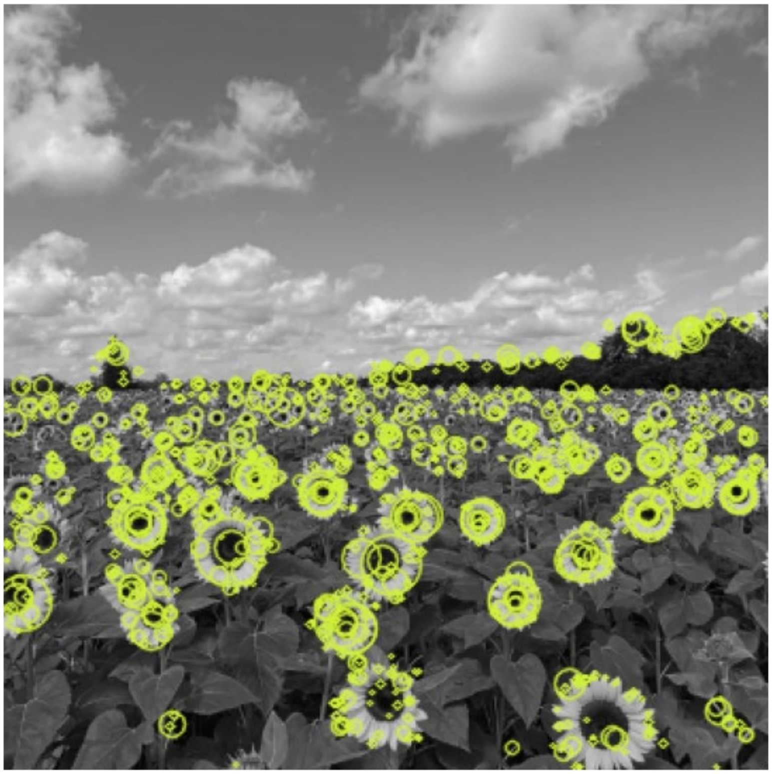
200687M Vishwamith P.G.H Assignment 2

Q1)

Blob Detected Image Circles



Largest Circle Sigma value range: (0.1,2.8)

Center : (106, 257)

Radius : 16

Sigma : 2.8

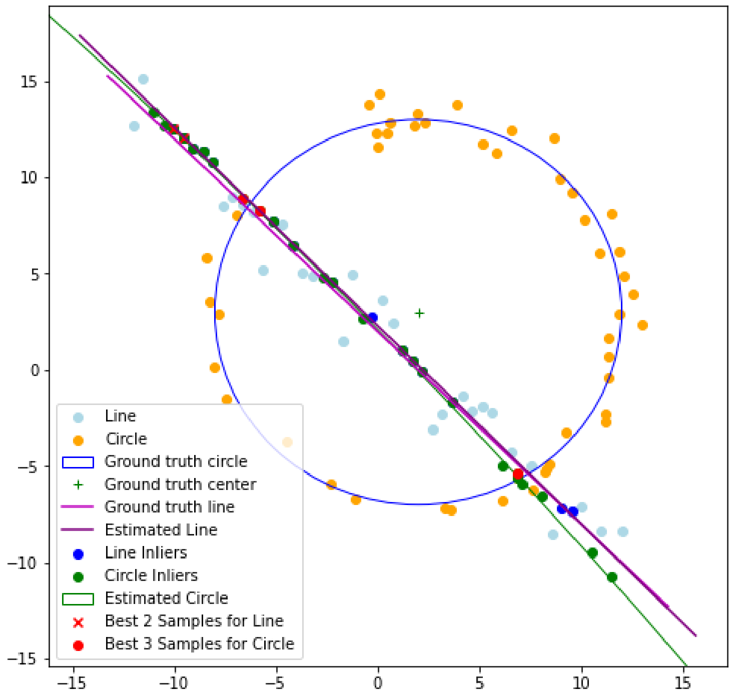
Image is converted to gray scale for to detect blobs more accurately. For different sigma values gray image is pass through a Gaussian Blur and Laplacian Filter. A blob mask has created using a threshold. Using findCounter function the found counters have been drawn on the gray image.

```
# Create an empty list to store detected circles
circles = []

# Loop through different sigma values to detect blobs at different scales
for sigma in np.linspace(min_sigma, max_sigma, num_sigma):
    # Apply LoG (Laplacian of Gaussian) to the grayscale image with the current sigma
    Gau = cv2.GaussianBlur(gray_image, (0, 0), sigma)
    Lap = cv2.Laplacian(Gau, cv2.CV_64F)

    # Calculate the absolute value of Laplacian values
    abs_Lap = np.abs(Lap)

    # Create a binary image where blobs are detected using the threshold
    blob_mask = abs_Lap > threshold * abs_Lap.max()

    # Find contours in the blob mask
    contours, _ = cv2.findContours(blob_mask.astype(np.uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Loop through the detected contours and fit circles to them
    for contour in contours:
        if len(contour) >= 5:
            (x, y), radius = cv2.minEnclosingCircle(contour)
            center = (int(x), int(y))
            radius = int(radius)
            circles.append((center, radius, sigma))
```

Q2)

By considering the threshold for the error is 1 is a good parameter since the data points are spread between -15 and 15 on x,y axis.

To calculate parameters of line and circle equation separate function have created. Separate RANSAC models have been created for line and circle.




```python
# Define a function to calculate the line equation from two points
def calculate_line_equation(x1, y1, x2, y2):
    # Calculate differences in x and y coordinates
    delta_x = x2 - x1
    delta_y = y2 - y1

    # Calculate magnitude
    magnitude = math.sqrt(delta_x**2 + delta_y**2)

    # Calculate coefficients of the line equation
    a = delta_y / magnitude
    b = -delta_x / magnitude
    d = (a * x1) + (b * y1)

    return a, b, d
```

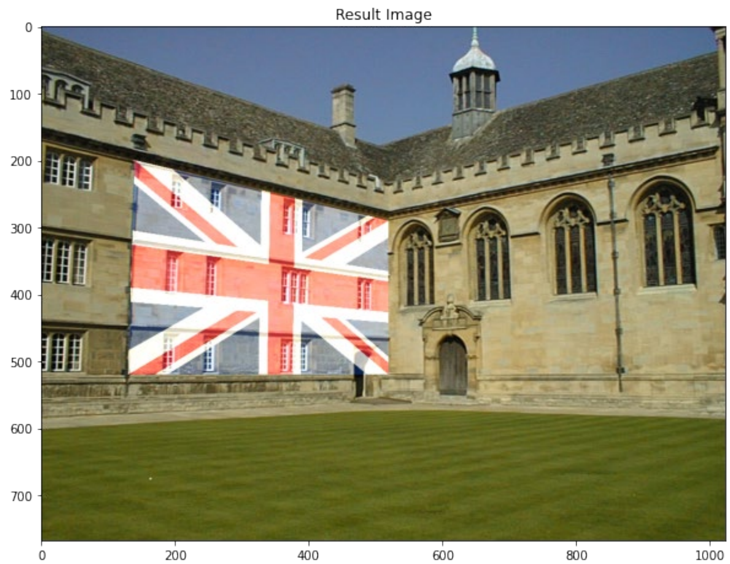
```python
# Define a function to compute the circle equation from three points
def compute_circle_equation(x1, y1, x2, y2, x3, y3):
    mx1, my1 = (x1 + x2) / 2, (y1 + y2) / 2
    mx2, my2 = (x2 + x3) / 2, (y2 + y3) / 2

    slope1 = (x2 - x1) / (y2 - y1) if y2 - y1 != 0 else 0
    slope2 = (x3 - x2) / (y3 - y2) if y3 - y2 != 0 else 0

    x_center = (slope1 * mx1 - slope2 * mx2 + my2 - my1) / (slope1 - slope2)
    y_center = -slope1 * (x_center - mx1) + my1

    radius = np.sqrt((x1 - x_center)**2 + (y1 - y_center)**2)

    return x_center, y_center, radius
```

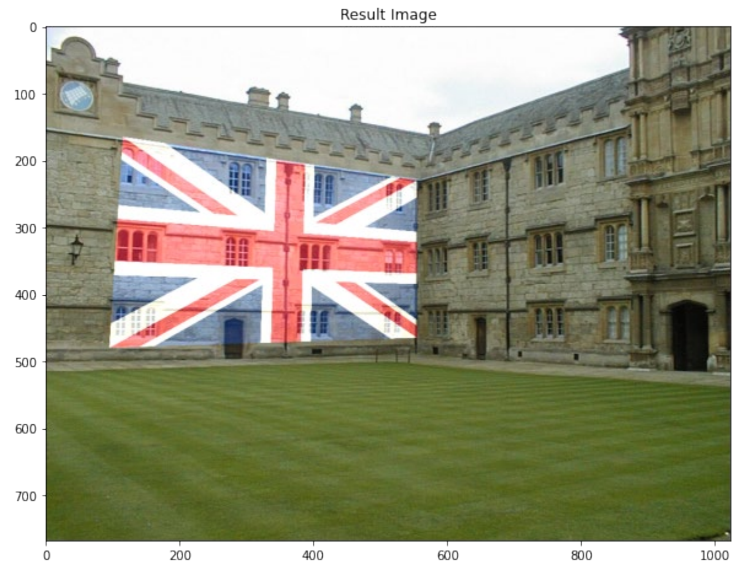
```python
# RANSAC to fit a line with unit normal constraint
def ransac_fit_line(X, iterations, threshold, min_inliers):
    best_model = None
    best_inliers = []

    for _ in range(iterations):
        # Randomly sample two points
        sample_indices = np.random.choice(len(X), 2, replace=False)
        x1, y1 = X[sample_indices[0]]
        x2, y2 = X[sample_indices[1]]

        # Calculate line equation coefficients
        a, b, d = calculate_line_equation(x1, y1, x2, y2)

        # Constraint: Ensure unit normal vector
        magnitude = np.sqrt(a**2 + b**2)
        a /= magnitude
        b /= magnitude

        # Calculate the distance of all points to the line
        distances = np.abs(a * X[:, 0] + b * X[:, 1] - d)

        # Find inliers based on the threshold
        inliers = np.where(distances < threshold)[0]

        if len(inliers) >= min_inliers:
            if len(inliers) > len(best_inliers):
                best_model = (a, b, d)
                best_inliers = inliers

    return best_model, best_inliers
```

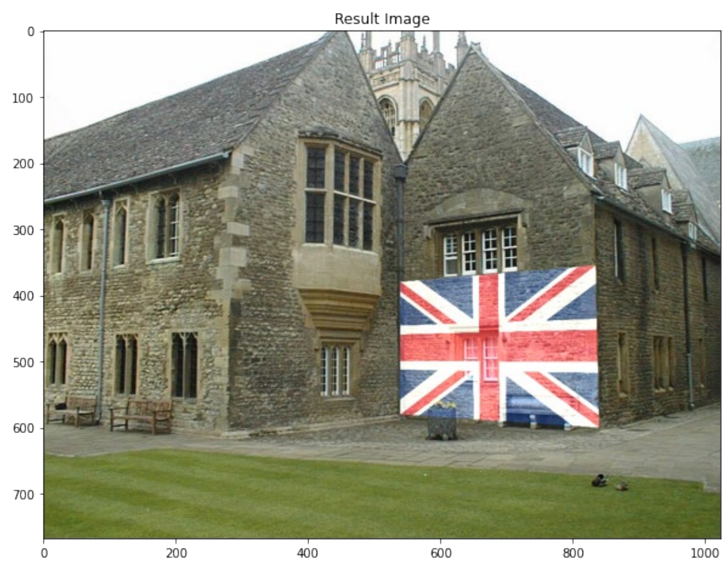
```python
# Use RANSAC to fit a circle
def ransac_circle(X, iterations, threshold, min_inliers):
    best_model = None
    best_inliers = []

    for _ in range(iterations):
        sample_indices = np.random.choice(len(X), 3, replace=False)
        x1, y1 = X[sample_indices[0]]
        x2, y2 = X[sample_indices[1]]
        x3, y3 = X[sample_indices[2]]

        x_center, y_center, radius = compute_circle_equation(x1, y1, x2, y2, x3, y3)

        errors = np.abs(np.sqrt((X[:, 0] - x_center)**2 + (X[:, 1] - y_center)**2) - radius)
        inliers = np.where(errors < threshold)[0]

        if len(inliers) >= min_inliers:
            if len(inliers) > len(best_inliers):
                best_model = (x_center, y_center, radius)
                best_inliers = inliers

    return best_model, best_inliers
```

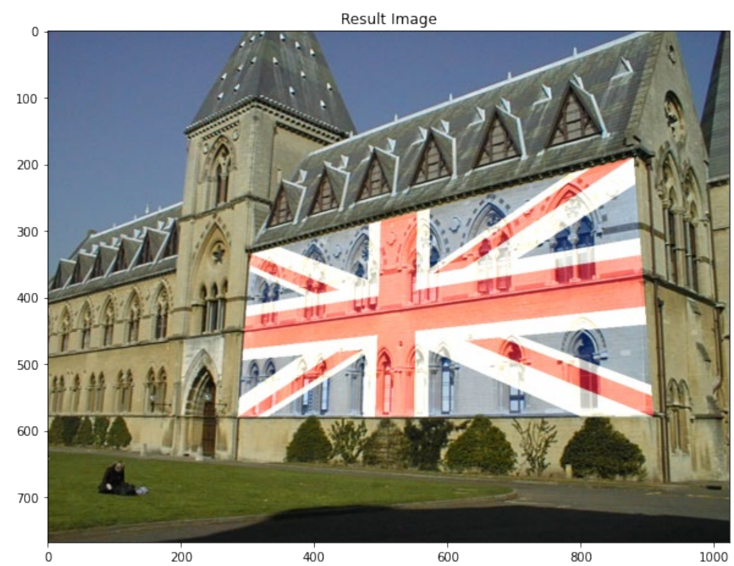
If we try to fit the circle first, then the circle will fit as previously but the point needs to predict the line will reduce. Specialty the circle and line crossing points. Thus the predicting line won’t be the optimal one.

Q3










```python
# Display the main image with a grid
display_image_with_grid(main_image)

# Define the four points on the main image that form a planar surface
# points_on_main = np.array([[140, 200], [520, 290], [520, 520], [130, 520]], dtype=np.float32)
# points_on_main = np.array([[115, 165], [555, 230], [555, 465], [95, 480]], dtype=np.float32)
# points_on_main = np.array([[540, 380], [835, 355], [840, 600], [540, 580]], dtype=np.float32)
points_on_main = np.array([[305, 335], [880, 190], [910, 580], [285, 580]], dtype=np.float32)


# Define the corresponding points on the overlay image (in the same order)
points_on_overlay = np.array([[0, 0], [overlay_image.shape[1], 0], [overlay_image.shape[1], overlay_image.shape[0]], [0, overlay_image.shape[0]]], dtype=np.float32)

# Compute the homography matrix
homography_matrix, _ = cv2.findHomography(points_on_overlay, points_on_main)

# Warp the overlay image onto the main image using the homography
overlay_warped = cv2.warpPerspective(overlay_image, homography_matrix, (main_image.shape[1], main_image.shape[0]))

# Blend the warped overlay image with the main image
result_image = cv2.addWeighted(main_image, 1, overlay_warped, 0.7, 0)
```

I have specifically chosen the above images so I can strongly show the different orientations that the flag can be projected. This will be a good demo for this assignment.

First I plotted the images with grid on and found the corner coordinates that the flags can be projected. Then using the findHolmography function I found the homography matrix then warp the flag onto the main image using the homography matrix. After that I have blend the warped overlay image with the main image. Which is the output image.

Q4)

Computed Homography

6.35329389e-01	5.19838188e-02	2.21629196e+02
2.33088406e-01	1.14415052e+00	-2.52127020e+01
5.19134819e-04	-7.50450702e-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

img1.ppm img5.ppm Stitched Image

```python
def compute_homographies(images, gray_images, p, sampleSize, outlierRatio):
    N = int(np.ceil(np.log(1 - p) / np.log(1 - (1 - outlierRatio) ** sampleSize)))
    Hs = []

    for i in range(4):
        sift = cv.SIFT_create()
        key_points_1, descriptors_1 = sift.detectAndCompute(gray_images[i], None)
        key_points_5, descriptors_5 = sift.detectAndCompute(gray_images[i + 1], None)
        bf_match = cv.BFMatcher(cv.NORM_L1, crossCheck=True)
        matches = sorted(bf_match.match(descriptors_1, descriptors_5), key=lambda x: x.distance)

        Source_points = [key_points_1[k.queryIdx].pt for k in matches]
        Destination_points = [key_points_5[k.trainIdx].pt for k in matches]
        threshold, best_inliers, best_H = 2, 0, 0

        for i in range(N):
            ran_points = calculate_random_numbers(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 = calculate_homography(f_points, t_points)

            inliers = 0
```

```python
            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)
    H1_H5 = Hs[3] @ Hs[2] @ Hs[1] @ Hs[0]
    H1_H5 /= H1_H5[-1][-1]
    return H1_H5, Hs


file_names = ['./Images/img1.ppm', './Images/img2.ppm', './Images/img3.ppm',
              './Images/img4.ppm', './Images/img5.ppm']

images, gray_images = load_images(file_names)

p, sampleSize, outlierRatio = 0.99, 4, 0.5
H1_H5, Hs = compute_homographies(images, gray_images, p, sampleSize, outlierRatio)

print("Computed Homography = ", H1_H5)
print("Provided Homography = ", open("./Images/H1to5p", 'r').read())

img_p = cv.warpPerspective(images[0], H1_H5, (images[4].shape[1], images[4].shape[0]))
ret, threshold = cv.threshold(img_p, 10, 1, cv.THRESH_BINARY_INV)
img2_thresholded = np.multiply(threshold, images[4])
img_blended = cv.addWeighted(img2_thresholded, 1, img_p, 1, 0)
```