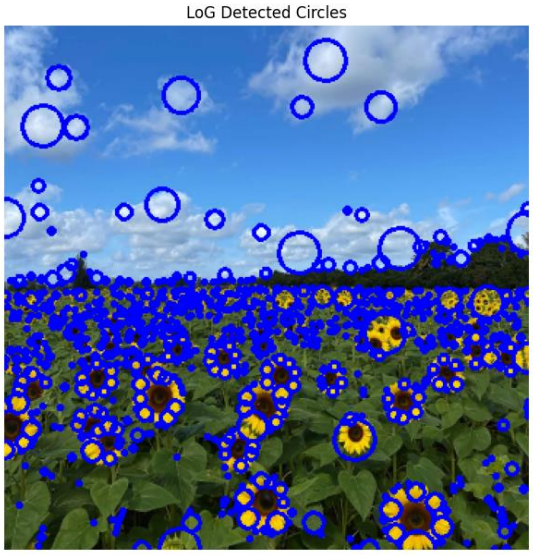
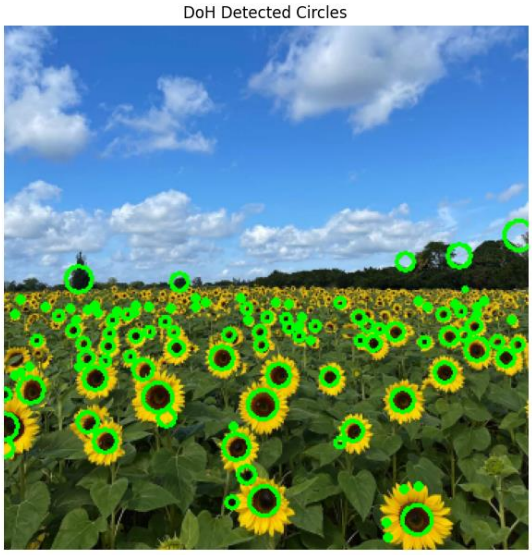
Image Processing and Machine Vision
EN3160 Assignment 2 on Fitting and Alignment
210222U HEWAGAMAGE K.L.N
https://github.com/KumalHewagamage/Image-Fitting-and-Alignment

Task 1 - Blob Detection

$\sigma = r/\sqrt{2}$ relationship is used to calculate the radius of blob. Both Laplacian of gaussian and Determinant (LoG) of Hessian (DoH) were used for blob detection. Out of two methods, DoH provided the best results. For both methods *min_sigma=1, max_sigma=10, num_sigma=20* were used.

```
# Detect blobs using Laplacian of Gaussian (LoG) method
blobs_log = blob_log(gray,min_sigma=1, max_sigma=10, num_sigma=20, threshold=0.1)
blobs_log[:, 2] = blobs_log[:, 2] * np.sqrt(2)  # Convert sigma to radius

# Detect blobs using Determinant of Hessian (DoH) method
blobs_doh = blob_doh(gray, min_sigma=1, max_sigma=10, num_sigma=20, threshold=0.01)
```





Largest circle using LoG: Center=(153.00, 271.00), Radius=14.14

Largest circle using DoH: Center=(255.00, 105.00), Radius=10.00

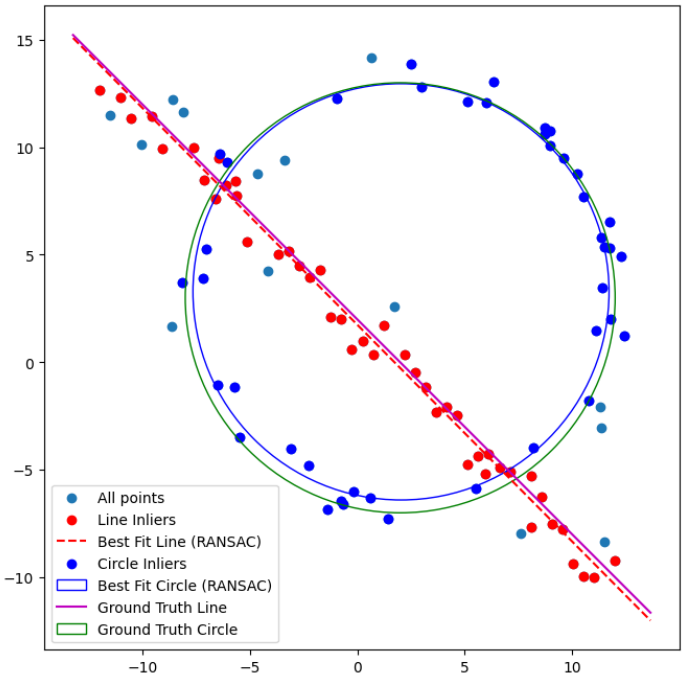
Task 2 - Fitting Using RANSAC

```
# Function to fit a line using RANSAC
def fit_line_ransac(X, threshold=1.0, iterations=100):
    best_inliers = []
    best_line = None
    for i in range(iterations):
        # Randomly select 2 points
        idx = np.random.choice(len(X), 2, replace=False)
        p1, p2 = X[idx]

        # Calculate the normal vector (a, b) to the line
        a, b = p2 - p1
        normal = np.array([-b, a]) / np.linalg.norm([a, b])

        # Calculate distance of all points to the line
        distances = np.abs((X - p1) @ normal)

        # Select inliers
        inliers = X[distances < threshold]
        if len(inliers) > len(best_inliers):
            best_inliers = inliers
            best_line = normal, p1

    return best_line, best_inliers
```

```
# Function to fit a circle using RANSAC
def fit_circle_ransac(X, threshold=1.0, iterations=100):
    best_inliers = []
    best_circle = None
    for i in range(iterations):
        # Randomly select 3 points
        idx = np.random.choice(len(X), 3, replace=False)
        p1, p2, p3 = X[idx]

        # Calculate the circle center and radius from the 3 points
        A = 2 * (p2 - p1)
        B = 2 * (p3 - p1)
        C = np.dot(p2, p2) - np.dot(p1, p1)
        D = np.dot(p3, p3) - np.dot(p1, p1)
        center = np.linalg.solve(np.array([A, B]), np.array([C, D]))
        radius = np.linalg.norm(center - p1)

        # Calculate distances from all points to the circle
        distances = np.abs(np.linalg.norm(X - center, axis=1) - radius)

        # Select inliers
        inliers = X[distances < threshold]
        if len(inliers) > len(best_inliers):
            best_inliers = inliers
            best_circle = center, radius

    return best_circle, best_inliers
```

Parameter	Line	Circle
S: Minimum number of points required to uniquely define the shape	2	3
t: Error threshold	1.0	1.0
Iterations	100	100

In this approach, the line is fitted first using RANSAC, and the inliers are then removed to allow fitting the circle. If the circle fitting were performed first, there could be a possibility that the three randomly selected points might be on the line, resulting in a large, inaccurate circle resembling a line. Fitting the line first helps in excluding the linear points, making it easier to subsequently fit the circle to the remaining data.



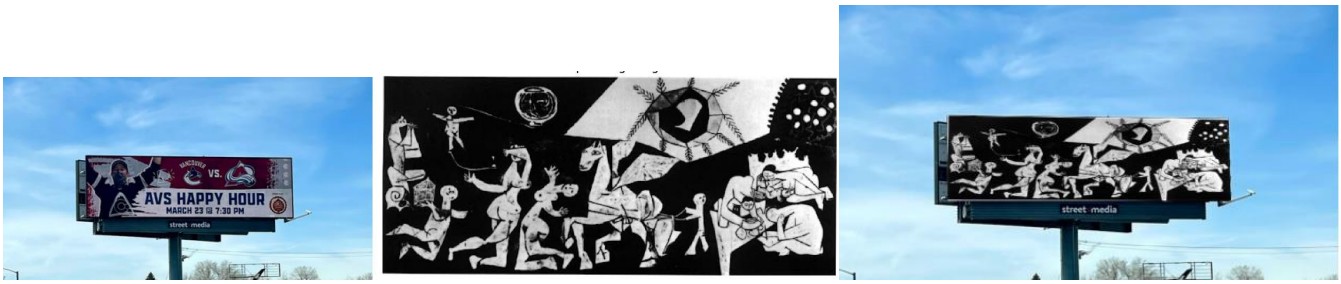

```
Best Fit Line:
Normal=[-0.71003654 -0.70416484],
Point on Line=[ 6.6122449  -4.91742667]

Best Fit Circle:
Center=[2.0370331  3.27607264],
Radius=9.679013715560805
```

Task 3 - Superimposing An Image On Another Image

```python
# Convert points to NumPy array of type float32
points_architectural = np.array(points_architectural, dtype='float32')

# Get the dimensions of the flag image
height, width, _ = flag_img.shape

# Define the four corners of the flag image
points_flag = np.array([[0, 0], [width - 1, 0], [width - 1, height - 1], [0, height - 1]], dtype='float32')

# Compute the homography matrix
homography_matrix, _ = cv.findHomography(points_flag, points_architectural)

# Warp the flag image to fit the selected region in the architectural image
warped_flag = cv.warpPerspective(flag_img, homography_matrix, (architectural_img.shape[1], architectural_img.shape[0]))

# Create a mask from the warped flag image
mask = np.zeros_like(architectural_img, dtype=np.uint8)
cv.fillConvexPoly(mask, points_architectural.astype(int), (255, 255, 255))

# Blend the warped flag image with reduced opacity to make it look more natural
alpha = 1  # Opacity level for blending
blended_warped_flag = cv.addWeighted(warped_flag, alpha, architectural_img, 1 - alpha, 0)

# Combine the blended flag with the architectural image
blended_img = cv.bitwise_and(architectural_img, cv.bitwise_not(mask)) + cv.bitwise_and(blended_warped_flag, mask)
```

A fantasy portal on big ben tower.

A failed to remove sticker on a satellite.



A Painting of Picasso on a highway billboard

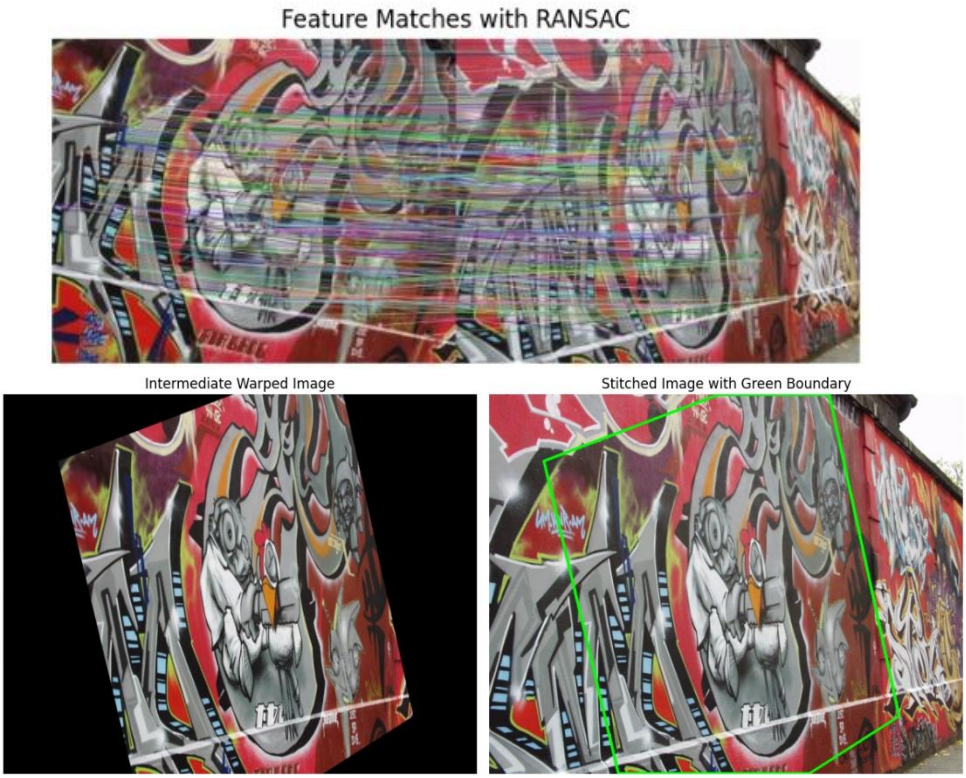
In this code homography matrix is computed between the four corners of the flag image and the selected points on the architectural image, allowing for alignment of the flag (or another image) with the target region. The flag image is then warped using this matrix, transforming it to fit the perspective of the selected area. A mask is created to define the target region, and the warped flag image is blended with the architectural image. Finally, the blended flag is combined with the original architectural image.

Task 4 - Image Stitching

```
def get_sift_features(img1, img2):
    # Convert to grayscale
    img1_gray = cv.cvtColor(img1, cv.COLOR_RGB2GRAY)
    img2_gray = cv.cvtColor(img2, cv.COLOR_RGB2GRAY)

    # Get SIFT descriptors
    sift = cv.SIFT_create(nOctaveLayers=3, contrastThreshold=0.09, edgeThreshold=25,
sigma=1)
    keypoints1, descriptors1 = sift.detectAndCompute(img1_gray, None)
    keypoints2, descriptors2 = sift.detectAndCompute(img2_gray, None)

    # Match using Brute Force Matcher
    bf = cv.BFMatcher()
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)
```

```
    # Lowe's ratio to filter best matches
    best_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            best_matches.append(m)

    return best_matches, keypoints1, keypoints2
```

Due to different perspective there not much feature to match with 1 and 5. Therefore we can see significant error in too homography matrices. This error is much lower in other image pairs.




```
Computed Homography Matrix:
 [[ 7.07093202e-01  2.75632974e-01  9.23409618e+01]
 [-2.36354357e-01  1.14081710e+00  1.12674304e+02]
 [ 2.18363476e-04  4.76726736e-05  1.00000000e+00]]
Actual Homography Matrix from Dataset:
 [[ 7.6285898e-01 -2.9922929e-01  2.2567123e+02]
 [ 3.3443473e-01  1.0143901e+00 -7.6999973e+01]
 [ 3.4663091e-04 -1.4364524e-05  1.0000000e+00]]
Error between Computed and Actual Homography:
 231.84901767445382
```