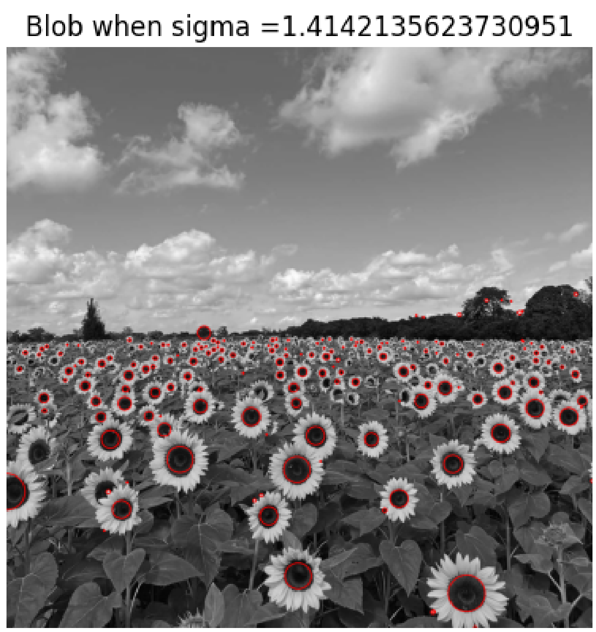
EN3160 - Image Processing and Machine Vision

Assignment 2 - Fitting and Alignment

SAMARAWEERA D. T. 200564J

Question 1



The script uses the Laplacian of Gaussian (LoG) filter to identify image regions where intensity changes are significant. It detects blobs by finding local maxima in the LoG-filtered image.

Threshold = 0.1

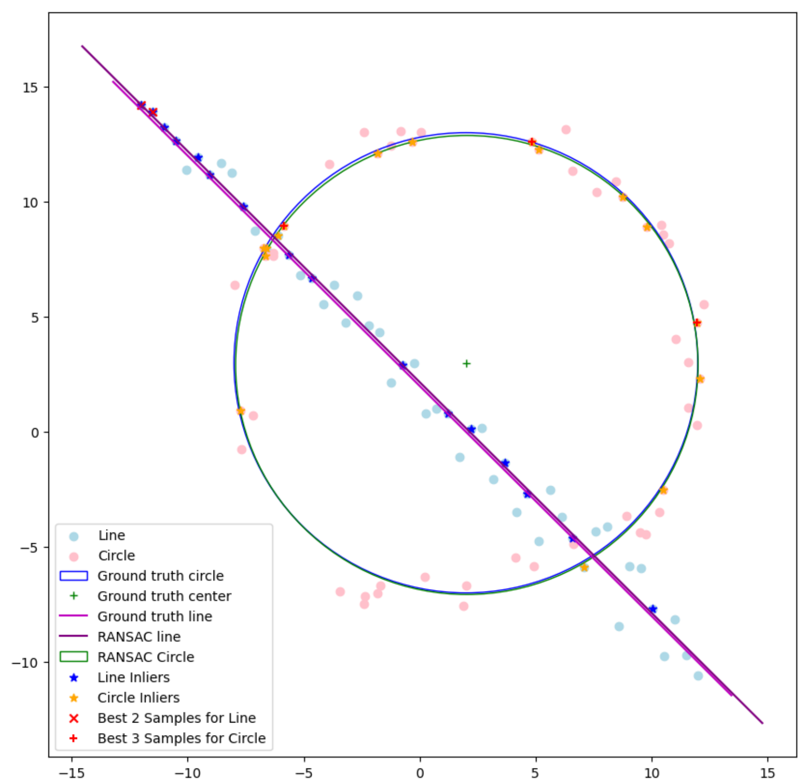
Sigma = 1.414

```
def log_kernel(sigma, size):
    if size % 2 == 0:
        size += 1
    sigma2 = sigma ** 2
    idx_range = np.linspace(-(size - 1) / 2., (size - 1) / 2., size)
    x_idx, y_idx = np.meshgrid(idx_range, idx_range)
    tmp_cal = -(np.square(x_idx) + np.square(y_idx)) / (2. * sigma2)
    kernel = np.exp(tmp_cal)
    kernel[kernel < np.finfo(float).eps * np.amax(kernel)] = 0
    k_sum = np.sum(kernel)
    if k_sum != 0:
        kernel /= np.sum(kernel)
    tmp_kernel = np.multiply(kernel, np.square(x_idx) + np.square(y_idx) -
2 * sigma2) / (sigma2 ** 2)
    kernel = tmp_kernel - np.sum(tmp_kernel) / (size ** 2)
    return kernel
```

Question 2

The code uses RANSAC (Random Sample Consensus) to robustly fit a line and circle to a set of noisy 2D points. It randomly samples subsets of points, estimates model parameters, and identifies inliers based on a threshold, iterating to find the best-fitting models.

```
def line_equation_from_points(x1, y1, x2, y2):
    delta_x = x2 - x1
    delta_y = y2 - y1
    magnitude = math.sqrt(delta_x**2 + delta_y**2)
    a = delta_y / magnitude
    b = -delta_x / magnitude
    d = (a * x1) + (b * y1)
    return a, b, d
```



The ransac_line function uses RANSAC to robustly estimate a line from 2D points. It randomly samples two points, calculates the line parameters, normalizes them, and identifies inliers based on a distance threshold.

```
def ransac_line(X, iterations, threshold, min_inliers):
    best_model = None
    best_inliers = []
    for _ in range(iterations):
        sample_indices = np.random.choice(len(X), 2, replace=False)
        x1, y1 = X[sample_indices[0]]
        x2, y2 = X[sample_indices[1]]
        a, b, d = line_equation_from_points(x1, y1, x2, y2)
        magnitude = np.sqrt(a**2 + b**2)
        a /= magnitude
        b /= magnitude
        distances = np.abs(a*X[:,0] + b*X[:,1] - d)
        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
```

The circle_equation_from_points function calculates the center and radius of a circle given three points. It uses the midpoints, slopes of two lines, and equations for circles.

```
def circle_equation_from_points(x1, y1, x2, y2, x3, y3):
    # Calculate the midpoints of two line segments
    mx1, my1 = (x1 + x2) / 2, (y1 + y2) / 2
    mx2, my2 = (x2 + x3) / 2, (y2 + y3) / 2
    # Calculate the slopes of the two lines
    if y2 - y1 == 0:
        slope1 = 0  # Avoid division by zero
    else:
        slope1 = (x2 - x1) / (y2 - y1)
    if y3 - y2 == 0:
        slope2 = 0  # Avoid division by zero
    else:
        slope2 = (x3 - x2) / (y3 - y2)
    # Calculate the center of the circle
    x_center = (slope1 * mx1 - slope2 * mx2 + my2 - my1) / (slope1 - slope2)
    y_center = -slope1 * (x_center - mx1) + my1
    # Calculate the radius
    radius = np.sqrt((x1 - x_center)**2 + (y1 - y_center)**2)
    return x_center, y_center, radius
```

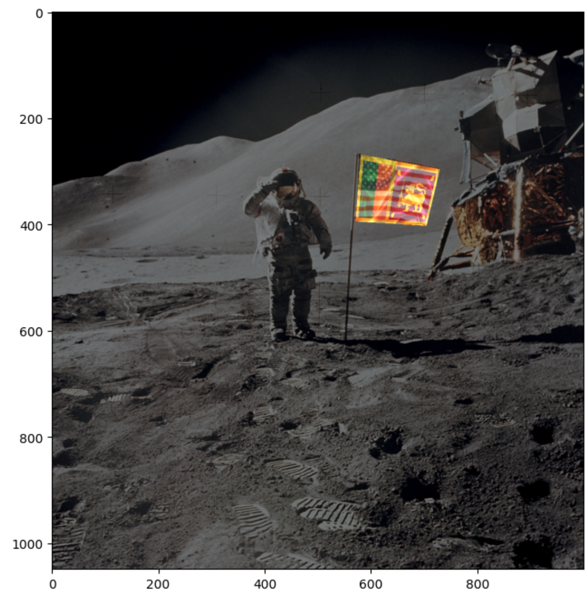
```
def ransac_circle(X, iterations, threshold, min_inliers):
    best_model = None
    best_inliers = []
    for _ in range(iterations):
        sample_indices = np.random.choice(len(X), 3, replace=False)
        x1, y1 = X[sample_indices[0]]
        x2, y2 = X[sample_indices[1]]
        x3, y3 = X[sample_indices[2]]
        x_center, y_center, radius = circle_equation_from_points(x1, y1, x2, y2, x3, y3)
        # Calculate the radial error of all points to the circle
        errors = np.abs(np.sqrt((X[:, 0] - x_center)**2 + (X[:, 1] - y_center)**2) - radius)
        # Find inliers based on the threshold
        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
```

Question 3

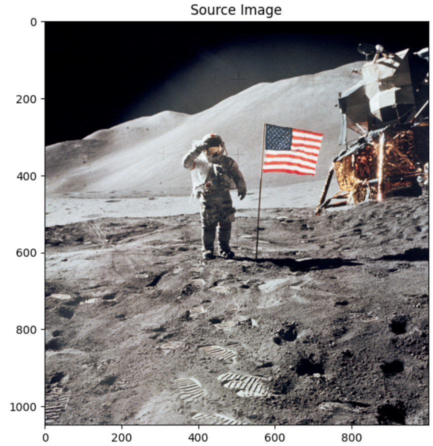
The code allows to select four corresponding points in two images, computes a homography matrix to align them, warps one image onto the other, blends the result, and displays it.

```
n = 0
cv.namedWindow('Image Flag', cv.WINDOW_AUTOSIZE)
param = [p_flag, im_flag_copy]
cv.setMouseCallback('Image Flag',draw_circle, param)
while(1):
    cv.imshow('Image Flag', im_flag_copy)
    if n == N:
        break
    if cv.waitKey(20) & 0xFF == 27:
        break
cv.destroyAllWindows()
```

```
h, status = cv.findHomography(p, p_flag)
print(h)
warped_img = cv.warpPerspective(im_flag, np.linalg.inv(h),
(im.shape[1],im.shape[0]))
# Blending
alpha = 0.5
beta = 0.8
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))
```



Source Image



Flag Image

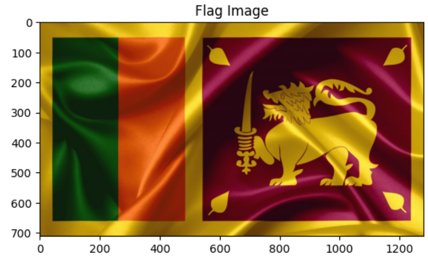
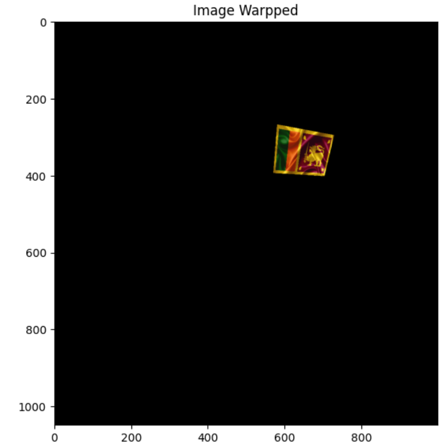


Image Warpped



Question 4

SIFT matching between img1.ppm onto img5.ppm

GOOD_MATCH_PERCENT = 0.65

```
def sift_match(im1, im2):
    GOOD_MATCH_PERCENT = 0.65
    sift = cv.SIFT_create()
    keypoint_1, descriptors_1 = sift.detectAndCompute(im1,None)
    keypoint_2, descriptors_2 = sift.detectAndCompute(im2,None)
    matcher = cv.BFMatcher()
    matches = matcher.knnMatch(descriptors_1, descriptors_2, k = 2)
    good_matches = []
    for a,b in matches:
        if a.distance < GOOD_MATCH_PERCENT*b.distance:
            good_matches.append(a)
    points1 = np.zeros((len(good_matches), 2), dtype=np.float32)
    points2 = np.zeros((len(good_matches), 2), dtype=np.float32)
    for i, match in enumerate(good_matches):
        points1[i, :] = keypoint_1[match.queryIdx].pt
        points2[i, :] = keypoint_2[match.trainIdx].pt
    fig, ax = plt.subplots(figsize = (15,15))
    ax.axis('off')
    matched_img = cv.drawMatches(im1, keypoint_1, im2, keypoint_2, good_matches, im2, flags = 2)
    plt.imshow(cv.cvtColor(matched_img,cv.COLOR_BGR2RGB))
    plt.show()
    result = np.concatenate((points1,points2), axis = 1)
    return result
```

The sift_match function uses SIFT (Scale-Invariant Feature Transform) to detect and match keypoint features between two images. It filters matches based on Lowe's ratio test and visualizes them.

The calculateHomography function estimates a homography matrix using a set of point correspondences. It constructs a linear equation system and solves it using singular value decomposition (SVD) for robust transformation estimation.

```
def calculateHomography(correspondences):
    temp_list = []
    for points in correspondences:
        p1 = np.matrix([points.item(0), points.item(1), 1])
        p2 = np.matrix([points.item(2), points.item(3), 1])
        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)
    u, s, v = np.linalg.svd(assemble_matrix)
    h = np.reshape(v[8], (3, 3))
    h = (1/h.item(8)) * h
    return h
```

Stitched Image

Calculated Homography between img1.ppm onto img5.ppm

```
[[ 4.42486546e-02 -2.84612059e-01 -1.77583734e+00]
 [ 3.18087311e-01 -8.04601220e-03 -2.38141568e+02]
 [-1.13171867e-03 -1.35188952e-03  1.00000000e+00]]
```