EN3160 Assignment 2 on Fitting and Alignment

Samaranayake M.A.S - 210559H

Blob Detection using Laplacians of Gaussian and Scale-Space Extrema Detection

Here to implement the blob detection, first define the sigma values and then blur the images. Then the Laplacian Gaussian of the images are computed. A threshold value of 3.0 is set get the accurate detections. The following function and code snippet shows the code to detect the blobs and the following figure shows the output results. Sigma values range from 1 to 50 and using np.linspace method, random 10 numbers are selected as sigma values.

```
# Generate blurred images to detect blobs at different scales
sigma_values = np.linspace(1, 50, 10)
blurred_images = [cv.GaussianBlur(im_gray, (0, 0), sigma) for sigma in sigma_values]
# Compute the Laplacian of Gaussian
blobs_log = [cv.Laplacian(img, cv.CV_64F) for img in blurred_images]
threshold = 3.00
# Find local extrema across the scale
def blob_detector(images, sigma_values, threshold):
    blobs = []
    blob_radii = []
    for i in range(1, len(images) - 1):
        cur, prev, nxt = images[i], images[i - 1], images[i + 1]
        extrema = np.logical_and(np.logical_and(cur > prev, cur > nxt), cur > threshold)
        y_x = np.argwhere(extrema)  # centers of the blobs
        if y_x.size > 0:
            blobs.append(y_x)
            blob_radii.append(sigma_values[i] * np.sqrt(2))  # Append radii for each
    detected blob
    return blobs, blob_radii
```

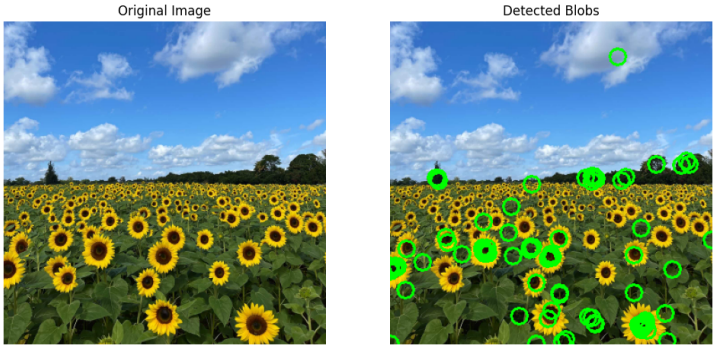


Figure 1: Blob detection output

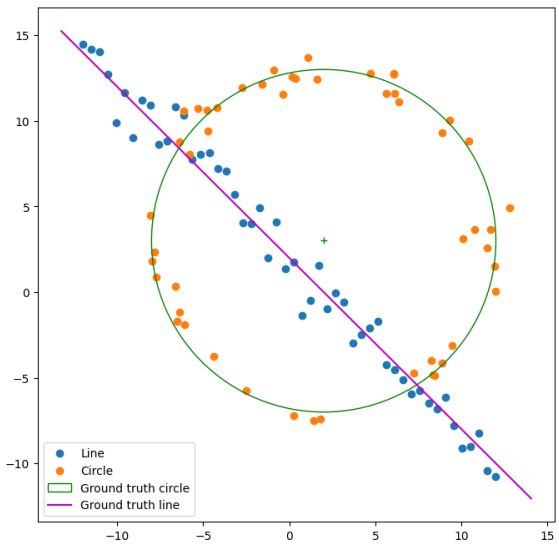
Fitting lines and circles using RANSAC

Here a line and circle is fitted using RANSAC. RANSAC handles the noisy and the outlier points effectively. In the line fitting 0.5 is used as the threshold which defines the maximum distance from a point to the fitted

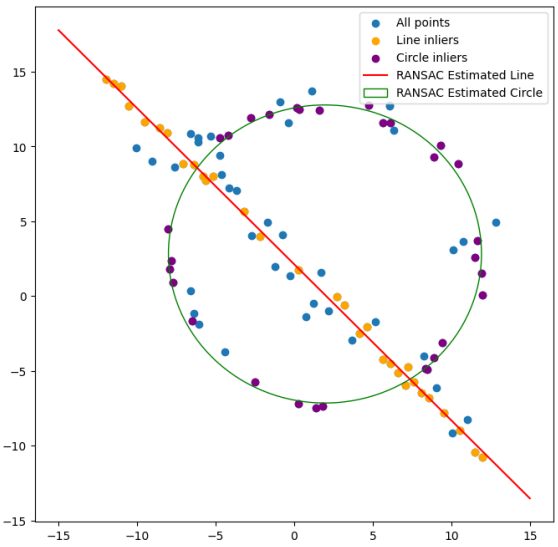
line for it to be considered an inlier. In the circle fitting 0.5 is used as threshold and random 3 points are used to define the circle.

```
def estimate_circle_ransac(X, threshold, iterations):
    best_inliers = []
    best_circle = None
    for _ in range(iterations):
        # Randomly sample three points to define a circle
        sample_indices = np.random.choice(X.shape[0], 3, replace=False)
        pts = X[sample_indices]
        # Estimate circle parameters using the three points
        A = 2 * (pts[1] - pts[0])
        B = 2 * (pts[2] - pts[0])
        C = np.array([np.sum(pts[1]**2 - pts[0]**2), np.sum(pts[2]**2 - pts[0]**2)])
        try:
            center = np.linalg.solve(np.array([A, B]), C)
            radius = np.linalg.norm(center - pts[0])
        except np.linalg.LinAlgError:
            continue
        # Compute distances to the circle and count inliers
        distances = np.abs(np.sqrt((X[:, 0] - center[0])**2 + (X[:, 1] - center[1])**2) -
 radius)
        inliers = X[distances < threshold]

        if len(inliers) > len(best_inliers):
            best_inliers = inliers
            best_circle = (center, radius)
    return best_circle[0], best_circle[1], np.array(best_inliers)
# Main execution
params_line, line_inliers = estimate_line_ransac(X, threshold=0.5, iterations=1000)
center_circle, radius_circle, circle_inliers = estimate_circle_ransac(X[~np.isin(X,
    line_inliers).all(axis=1)], threshold=0.5, iterations=1000)
```



(a) Noisy and outlier points



(b) RANSAC line and circle fitted

If circle is fitted first, then line points can be classify as inliers for the circle mistakenly. It's typically easier to distinguish a line from a circle than the other way around, hence the order matters.

Superimposing an image to a background image

Here the mouse callback function is used to store four points in the background image. Then perspective transformation matrix is calculated. Then the image will be wrapped using opencv methods. By creating a

mask in the background image and after blacking out it, the other image is combined with the photo. The following code snippet shows how to get the four points of the background and how wrapping was done.

```
# Mouse callback function to store points
def click_event(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        if len(dst_points) < 4:  # We only need 4 points
            dst_points.append((x, y))
            cv.circle(arch_image, (x, y), 5, (0, 0, 255), -1)
            print(f"Point {len(dst_points)}: ({x}, {y})")  # Print point for debugging
            cv.imshow("Select 4 Points", arch_image)
        if len(dst_points) == 4:
            cv.destroyWindow("Select 4 Points")
# Calculate the perspective transformation matrix
M = cv.getPerspectiveTransform(src_points, dst_points)
print("Transformation matrix:\n", M)  # Print matrix for debugging
# Warp the flag to fit into the selected region in the architectural image
warped_flag = cv.warpPerspective(flag_image, M, (arch_image.shape[1], arch_image.shape
    [0]))
# Create a mask for blending based on the non-black pixels in the transformed flag
flag_gray = cv.cvtColor(warped_flag, cv.COLOR_BGR2GRAY)
_, mask = cv.threshold(flag_gray, 1, 255, cv.THRESH_BINARY)
# Invert the mask to select the background region
mask_inv = cv.bitwise_not(mask)
# Black-out the area of the flag in the architectural image
arch_bg = cv.bitwise_and(arch_image, arch_image, mask=mask_inv)
# Take only the flag region from the transformed image
flag_fg = cv.bitwise_and(warped_flag, warped_flag, mask=mask)
# Add the background and flag regions
final_image = cv.add(arch_bg, flag_fg)
```

The following outputs show different blended images. The output images are some designs that can see commonly. But those are mostly implemented using Photoshop. I selected these two images to see it without these software

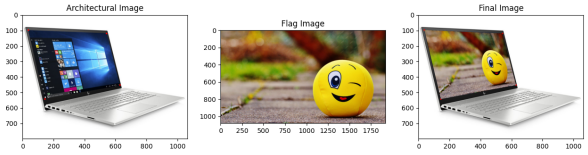


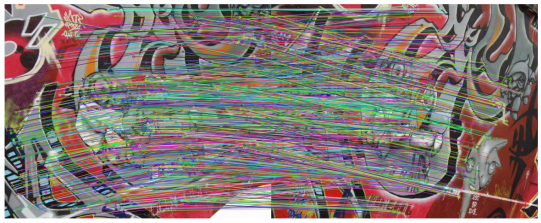
Figure 3: Image superimpose example 01

Figure 4: Image superimpose example 02

Image Stitching

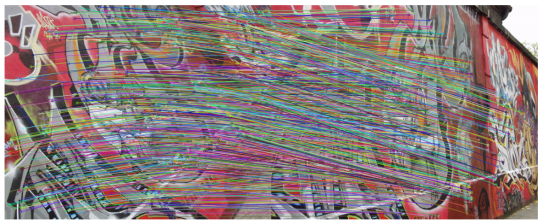
Since there are no enough matching between the image 1 and 5 we do the matching with 1-2, 1-3, 1-4 and carried out the image stitching. The following results were obtained.

(a) Matching 1



(b) Matching 02

(c) Matching 03



(d) Matching 04

```
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
```

As the final output we can get the stitched image of image 1 and image 5.

Figure 6: Final output