EN3160 – Image Processing and Machine Vision

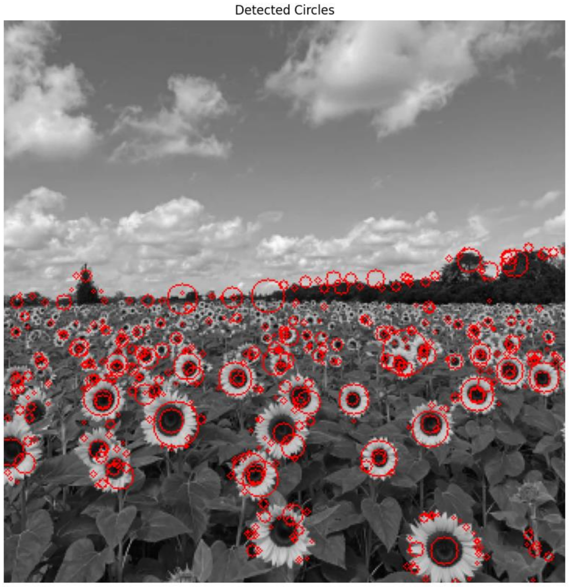
Assignment 02 – Fitting and Alignment

H.M.R.M. Pushpakumara – 200488E

Question 01

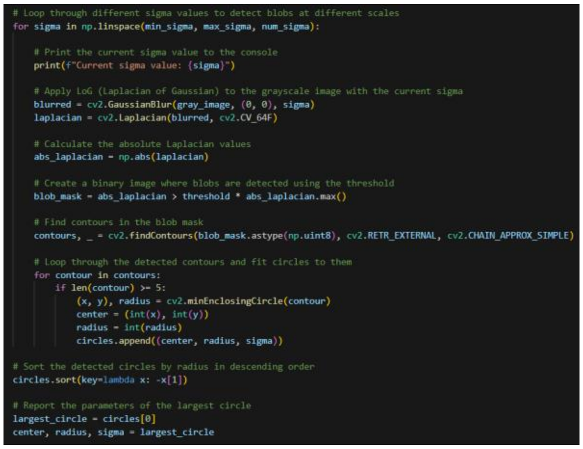
Here we have to detect blobs by using Gaussian and scale-space extrema detection.

I used sigma = 1 and sigma = 2 as my sigma values and my threshold value is 0.35.



Parameters of the largest circle:

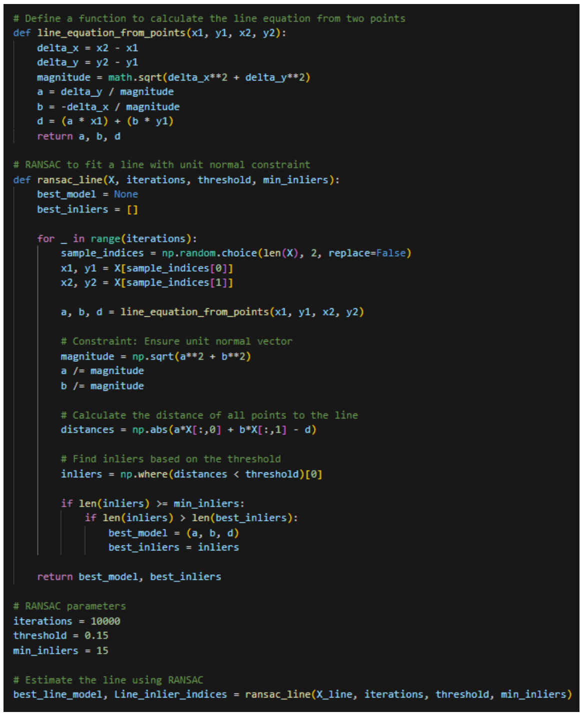
- Center: (110, 258)
- Radius: 20
- Sigma value: 2



```
# Loop through different sigma values to detect blobs at different scales
for sigma in np.linspace(min_sigma, max_sigma, num_sigma):

    # Print the current sigma value to the console
    print(f"Current sigma value: {sigma}")

    # Apply LoG (Laplacian of Gaussian) to the grayscale image with the current sigma
    blurred = cv2.GaussianBlur(gray_image, (0, 0), sigma)
    laplacian = cv2.Laplacian(blurred, cv2.CV_64F)

    # Calculate the absolute Laplacian values
    abs_laplacian = np.abs(laplacian)

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

# Sort the detected circles by radius in descending order
circles.sort(key=lambda x: -x[1])

# Report the parameters of the largest circle
largest_circle = circles[0]
center, radius, sigma = largest_circle
```

Question 02

(a)



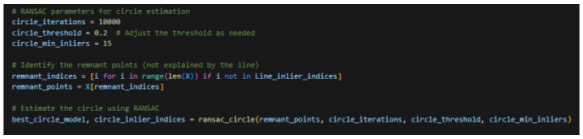
```
# Define a function to calculate the line equation from two points
def line_equation_from_points(x1, y1, x2, y2):
    delta_x = x2 - x1
    delta_y = y2 - y1
    magnitude = math.sqrt(delta_x**2 + delta_y**2)
    a = delta_y / magnitude
    b = -delta_x / magnitude
    d = (a * x1) + (b * y1)
    return a, b, d

# RANSAC to fit a line with unit normal constraint
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)

        # 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

# RANSAC parameters
iterations = 10000
threshold = 0.15
min_inliers = 15

# Estimate the line using RANSAC
best_line_model, Line_inlier_indices = ransac_line(X_line, iterations, threshold, min_inliers)
```

Number of inliers in this model: 16

Parameters of the estimated line:

- a: 0.7080986693010423
- b: 0.7061134997534697
- d: 1.5303717301982513

(b)



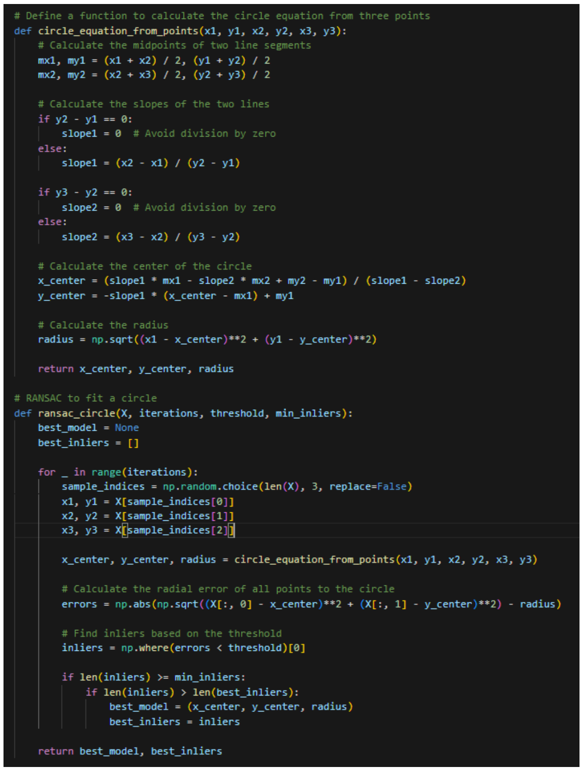
```
# RANSAC parameters for circle estimation
circle_iterations = 10000
circle_threshold = 0.2  # Adjust the threshold as needed
circle_min_inliers = 15

# Identify the remnant points (not explained by the line)
remnant_indices = [i for i in range(len(X)) if i not in Line_inlier_indices]
remnant_points = X[remnant_indices]

# Estimate the circle using RANSAC
best_circle_model, circle_inlier_indices = ransac_circle(remnant_points, circle_iterations, circle_threshold, circle_min_inliers)
```

Number of remnant points: 84

Number of inliers in the circle: 16

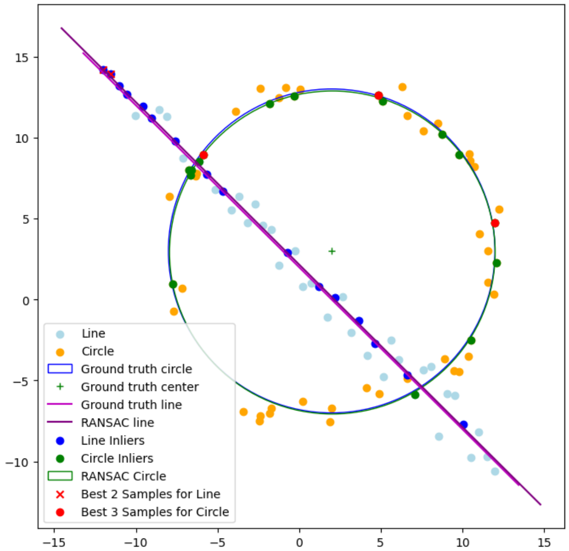
Parameters of the estimated circle:

- x_center: 2.0388812056906938
- y_center: 2.905374551155992
- radius: 9.971739443086292

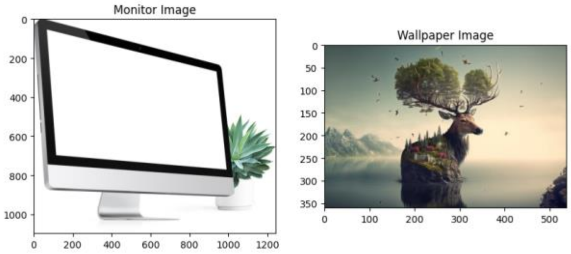


```
# Define a function to calculate the circle equation from three points
def circle_equation_from_points(x1, y1, x2, y2, x3, y3):
    # Calculate the midpoints of two line segments
    mx1, my1 = (x1 + x2) / 2, (y1 + y2) / 2
    mx2, my2 = (x2 + x3) / 2, (y2 + y3) / 2

    # 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

# 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 = 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
```

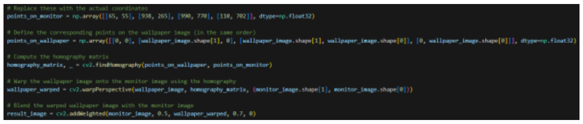
(c)



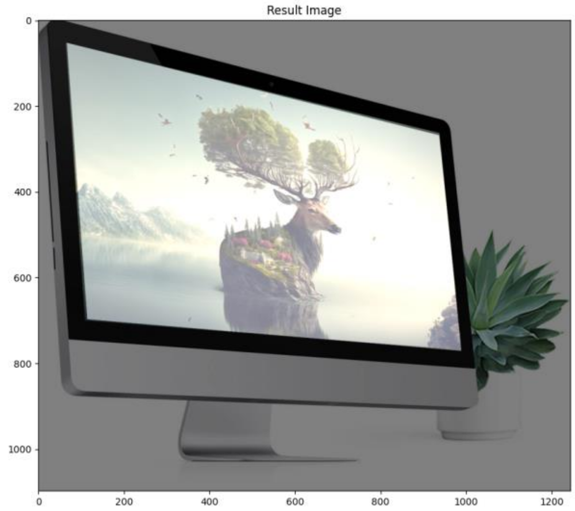
Question 03



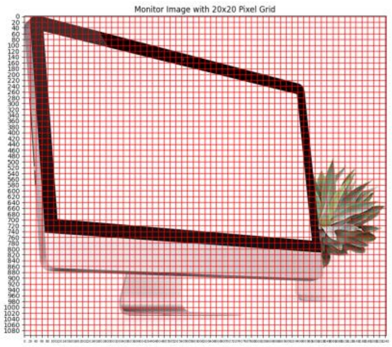
I chose a desktop image and an HD image to do this. My task was to add this HD image to the desktop as its wallpaper.



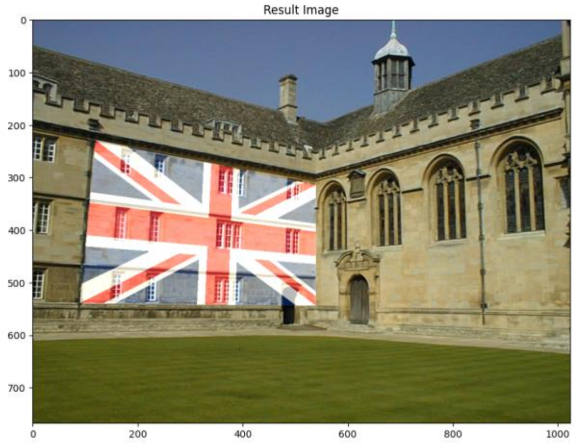
```
# Replace these with the actual coordinates
points_on_monitor = np.array([[65, 55], [938, 265], [990, 770], [110, 702]], dtype=np.float32)

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

# Compute the homography matrix
homography_matrix, _ = cv2.findHomography(points_on_wallpaper, points_on_monitor)

# Warp the wallpaper image onto the monitor image using the homography
wallpaper_warped = cv2.warpPerspective(wallpaper_image, homography_matrix, (monitor_image.shape[1], monitor_image.shape[0]))

# Blend the warped wallpaper image with the monitor image
result_image = cv2.addWeighted(monitor_image, 0.5, wallpaper_warped, 0.7, 0)
```



To define the 4 points on the desktop image, I generate a pixel grid on the image and by looking at that grid, I defined my four points.

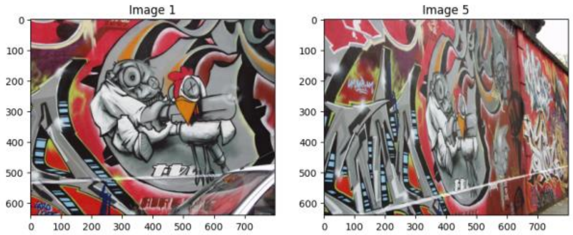


This image also generated by me by using the same approach.

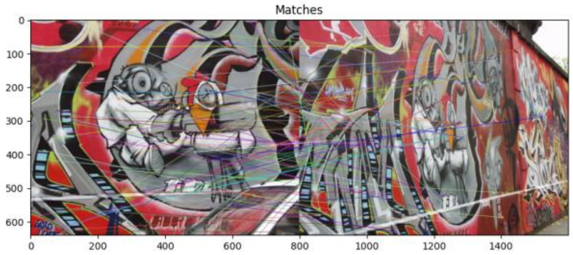


Question 04

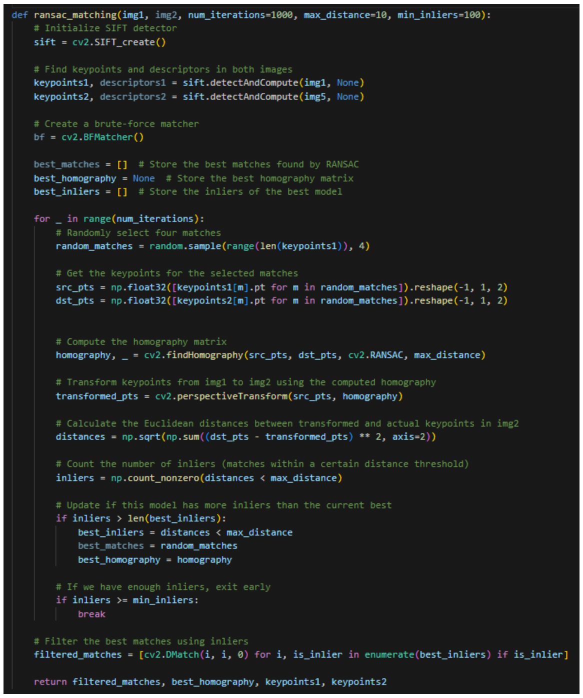
Image1 and Image5



When we use a brute-force matcher, it gives lots of crossmatches between two images. From that, we can't perform good image stitching.

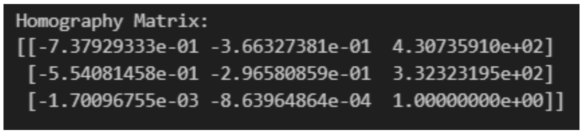


Therefore, I implemented a RANSAC matching function for this.

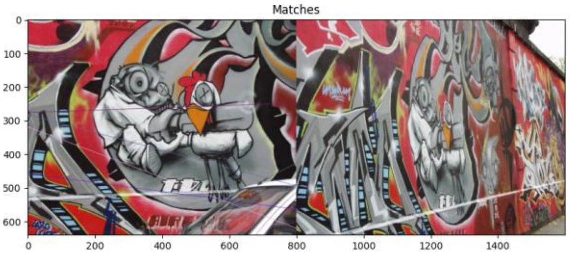


```
def ransac_matching(img1, img2, num_iterations=1000, max_distance=10, min_inliers=100):
    # Initialize SIFT detector
    sift = cv2.SIFT_create()

    # Find keypoints and descriptors in both images
    keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(img5, None)

    # Create a brute-force matcher
    bf = cv2.BFMatcher()

    best_matches = []  # Store the best matches found by RANSAC
    best_homography = None  # Store the best homography matrix
    best_inliers = []  # Store the inliers of the best model

    for _ in range(num_iterations):
        # Randomly select four matches
        random_matches = random.sample(range(len(keypoints1)), 4)

        # Get the keypoints for the selected matches
        src_pts = np.float32([keypoints1[m].pt for m in random_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([keypoints2[m].pt for m in random_matches]).reshape(-1, 1, 2)


        # Compute the homography matrix
        homography, _ = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, max_distance)

        # Transform keypoints from img1 to img2 using the computed homography
        transformed_pts = cv2.perspectiveTransform(src_pts, homography)

        # Calculate the Euclidean distances between transformed and actual keypoints in img2
        distances = np.sqrt(np.sum((dst_pts - transformed_pts) ** 2, axis=2))

        # Count the number of inliers (matches within a certain distance threshold)
        inliers = np.count_nonzero(distances < max_distance)

        # Update if this model has more inliers than the current best
        if inliers > len(best_inliers):
            best_inliers = distances < max_distance
            best_matches = random_matches
            best_homography = homography

        # If we have enough inliers, exit early
        if inliers >= min_inliers:
            break

    # Filter the best matches using inliers
    filtered_matches = [cv2.DMatch(i, i, 0) for i, is_inlier in enumerate(best_inliers) if is_inlier]

    return filtered_matches, best_homography, keypoints1, keypoints2
```

By using this, I generated a homography matrix.

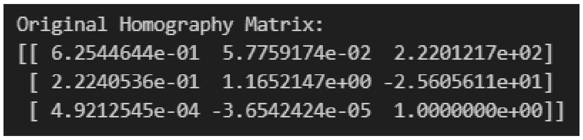


```
Homography Matrix:
[[-7.37929333e-01 -3.66327381e-01  4.30735910e+02]
 [-5.54081458e-01 -2.96580859e-01  3.32323195e+02]
 [-1.70096755e-03 -8.63964864e-04  1.00000000e+00]]
```



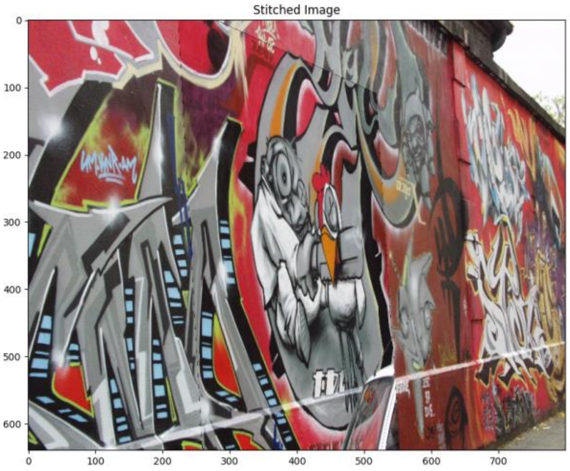
But this also not worked properly. I had non-cross exact matches, but only on the edge of the images.

Therefore, I used the given homography matrix for image1 to image5.



```
Original Homography Matrix:
[[ 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]]
```

Then I got this final image.



https://github.com/RavinduMPK/EN3160---Image-Processing-and-Machine-Vision

Direct to the assignment:
https://github.com/RavinduMPK/EN3160---Image-Processing-and-Machine-Vision/tree/main/Assignment_02%20%20-%20Fitting%20and%20Alignment