Assignment 2 - EN3160

Name Pathirana R. P. S.
Index Number 210451U
Github Link https://github.com/RPX2001/Assignment2_EN3160.git

Question 1 - Blob Detection

```
def generate_log_kernel(sigma):
    """Generate Laplacian of Gaussian (LoG) kernel."""
    radius = int(3 * sigma)
    x, y = np.meshgrid(np.arange(-radius, radius + 1),np.arange(radius, radius + 1))
    kernel = ((x**2 + y**2) / (2 * sigma**2) - 1) * np.exp(-(x**2 + y**2) / (2 *
        sigma**2)) / (np.pi * sigma**4)
    return kernel
def find_blob_centers(log_image, sigma):
    """Find local maxima in the LoG response image."""
    coords = []
    height, width = log_image.shape
    neighborhood = 1  # 3x3 window size
    for row in range(neighborhood, height-neighborhood):
        for col in range(neighborhood, width-neighborhood):
            local_window = log_image[row-neighborhood:row+neighborhood+1, col-
                neighborhood:col+neighborhood+1]
            max_response = np.max(local_window)
            if max_response >= 0.10:  # Adjust threshold for detection
                x_offset, y_offset = np.unravel_index(np.argmax(local_window),
                    local_window.shape)
                coords.append((row + x_offset - neighborhood, col + y_offset -
                    neighborhood))
    return set(coords)
```

To optimally detect a blob with radius r, the relationship was used.sigma values corresponding to r in the range of (1, 10) were applied for blob detection.This approach ensures that the scale parameter sigma is appropriately tuned for detecting blobs of different sizes, corresponding to the radi in the given range.

Figure 1: Blob Detection

Question 2 - Line and circle fitting using RANSAC

```python
def line_distance(points, a, b, d):
return np.abs(a * points[:, 0] + b * points[:, 1] + d) / np.sqrt(a**2 + b**2)
def fit_line(p1, p2):
    a = p1[1] - p2[1]  # y1 - y2
    b = p2[0] - p1[0]  # x2 - x1
    d = -(a * p1[0] + b * p1[1])  # ax1 + by1 + d = 0
    norm = np.sqrt(a**2 + b**2)
    return a / norm, b / norm, d / norm
# RANSAC for line fitting
def ransac_line(points, threshold, num_iterations=100):
    best_a, best_b, best_d = None, None, None
    best_consensus = []
    for _ in range(num_iterations):
        # Randomly select 2 points
        p1, p2 = points[sample(range(len(points)), 2)]
        a, b, d = fit_line(p1, p2)
        distances = line_distance(points, a, b, d)
        consensus_set = points[distances < threshold]
        if len(consensus_set) > len(best_consensus):
            best_a, best_b, best_d = a, b, d
            best_consensus = consensus_set
    return best_a, best_b, best_d, best_consensus
def circle_distance(points, x0, y0, r):
    return np.abs(np.sqrt((points[:, 0] - x0)**2 + (points[:, 1] - y0)**2) - r)
def fit_circle(p1, p2, p3):
    A = np.array([[p1[0], p1[1], 1],
                  [p2[0], p2[1], 1],
                  [p3[0], p3[1], 1]])
    B = np.array([-(p1[0]**2 + p1[1]**2),
                  -(p2[0]**2 + p2[1]**2),
                  -(p3[0]**2 + p3[1]**2)])
    sol = np.linalg.solve(A, B)
    x0, y0 = -0.5 * sol[0], -0.5 * sol[1]
    r = np.sqrt((sol[0]**2 + sol[1]**2) / 4 - sol[2])
    return x0, y0, r
def ransac_circle(points, threshold, num_iterations=100):
    best_x0, best_y0, best_r = None, None, None
    best_consensus = []
    for _ in range(num_iterations):
        # Randomly select 3 points
        p1, p2, p3 = points[sample(range(len(points)), 3)]
        x0, y0, r = fit_circle(p1, p2, p3)
        distances = circle_distance(points, x0, y0, r)
        consensus_set = points[distances < threshold]
        if len(consensus_set) > len(best_consensus):
            best_x0, best_y0, best_r = x0, y0, r
            best_consensus = consensus_set
    return best_x0, best_y0, best_r, best_consensus
threshold_line = 0.5
threshold_circle = 0.5
a, b, d, line_consensus = ransac_line(X, threshold_line)
X_remnant = np.array([point for point in X if point not in line_consensus])
x0, y0, r, circle_consensus = ransac_circle(X_remnant, threshold_circle)
```

This approach ensures that the scale parameter sigma is appropriately tuned for detecting blobs of different sizes, corresponding to the radii in the given range.

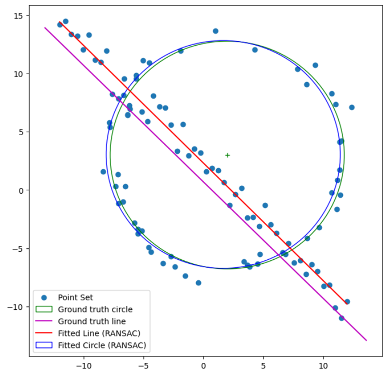


Figure 2: Line and Circle fitting

Figure 3: Superimposed image

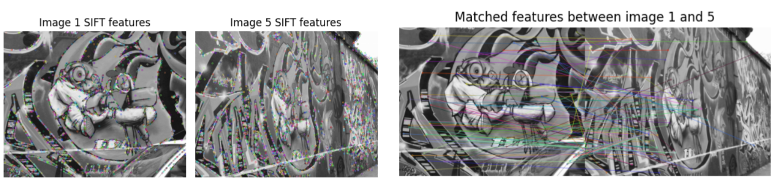
Question 3 - Superimposing an image on another

```python
def superimpose(image, logo, dst_points, beta=0.3, alpha=1):
    h, w, _ = logo.shape
    src_points = np.array([(0, 0), (w, 0), (w, h), (0, h)], dtype=np.float32)
    tform = transform.estimate_transform('projective', src_points, dst_points)
    tf_img = transform.warp(logo, tform.inverse, output_shape=image.shape[:2])
    tf_img = (tf_img * 255).astype(np.uint8)
    mask = np.any(tf_img > 0, axis=-1).astype(np.uint8)
    blended = cv.addWeighted(image, alpha, tf_img, beta, 0)
    result = np.where(mask[:, :, np.newaxis], blended, image)
    return result
```

Question 4 – Image stitching

There were insufficient matching features between images 1 and 5 to accurately compute a homography directly. As a result, homographies were calculated between consecutive image pairs, such as 1-2, 2-3, and so on, where the images had more similarities. Image 1 was then progressively transformed through each of these computed homographies to align it with image 5.

```python
def extract_SIFT_features(image1, image5, display = False):
    gray_img1 = cv.cvtColor(image1, cv.COLOR_RGB2GRAY)
    gray_img5 = cv.cvtColor(image5, cv.COLOR_RGB2GRAY)
    sift_detector = cv.SIFT_create(nOctaveLayers=3, contrastThreshold=0.09,
        edgeThreshold=25, sigma=1)
    kp1, desc1 = sift_detector.detectAndCompute(gray_img1, None)
    kp5, desc5 = sift_detector.detectAndCompute(gray_img5, None)
    matcher = cv.BFMatcher()
```



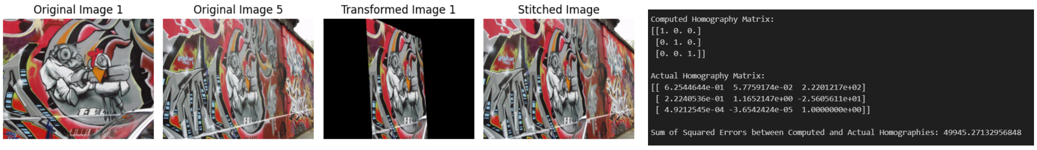


Figure 4: SIFT Matching

```python
    knn_matches = matcher.knnMatch(desc1, desc5, k=2)
    valid_matches = []
    for m, n in knn_matches:
        if m.distance < 0.75 * n.distance:
            valid_matches.append(m)

    return valid_matches, kp1, kp5

def identify_inliers(source_pts, destination_pts, homography, threshold):
    projected_pts = homography(source_pts)
    distances = np.sqrt(np.sum((projected_pts - destination_pts) ** 2, axis=1))

    return np.where(distances < threshold)[0]

def compute_best_homography(valid_matches, kp1, kp5):
    src_points_set = []
    dst_points_set = []
    for match in valid_matches:
        src_points_set.append(np.array(kp1[match.queryIdx].pt))
        dst_points_set.append(np.array(kp5[match.trainIdx].pt))

    src_points_set = np.array(src_points_set)
    dst_points_set = np.array(dst_points_set)
    ransac_points = 4
    error_threshold = 1
    min_inliers = 0.5 * len(valid_matches)
    max_iterations = 200
    best_homography = None
    max_inliers = 0
    best_inliers_indices = None
    for i in range(max_iterations):
        random_matches = np.random.choice(valid_matches, ransac_points, replace=False)
        src_ransac = []
        dst_ransac = []
        for match in random_matches:
            src_ransac.append(np.array(kp1[match.queryIdx].pt))
            dst_ransac.append(np.array(kp5[match.trainIdx].pt))
        src_ransac = np.array(src_ransac)
        dst_ransac = np.array(dst_ransac)
        homography_estimation = transform.estimate_transform('projective', src_ransac, dst_ransac)
        inliers_indices = identify_inliers(src_points_set, dst_points_set, homography_estimation, error_threshold)
        if len(inliers_indices) > max_inliers:
            max_inliers = len(inliers_indices)
            best_homography = homography_estimation
            best_inliers_indices = inliers_indices
    print(f'Maximum inliers count = {max_inliers}')
    return best_homography, best_inliers_indices
```