EN3160 Assignment 2 - Fitting and Alignment

Name: Kodikara U. S. S. | **Index No.:** 210293K | **Date:** 23rd October 2024

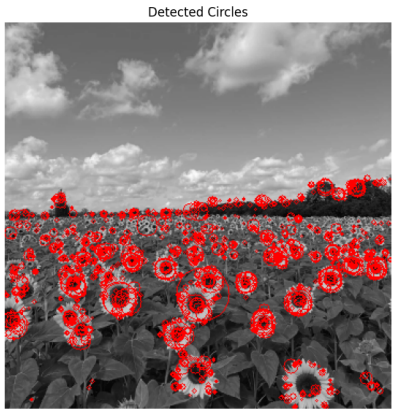
Github: https://github.com/uvinduuu/EN3160-ImageProcessingMachineVision/tree/main/Assignment02

1 Blob Detection

The algorithm iteratively applies Gaussian blur and the Laplacian operator to the grayscale image for different sigma values, detecting blobs by thresholding the Laplacian's absolute values. Contours are then extracted from the binary blob mask, and circles are fitted to the detected contours.
Range of σ Values: [0.71, 1.01, 1.31, 1.62, 1.92, 2.22, 2.53, 2.83]
The largest detected circle: Center: (184, 253), **Radius**: 25 pixels, **Sigma value**: 2.83

```python
# Set up parameters for scale-space extrema detection
min_sigma = 1.0  # Minimum sigma value (smaller values for smaller blobs)
max_sigma = 4.0  # Maximum sigma value (larger values for larger blobs)
num_sigma = 8   # Number of sigma values to test
threshold = 0.375  # Threshold for blob detection
circles = [] # Create an empty list to store detected circles
for sigma in np.linspace(min_sigma, max_sigma, num_sigma):
    sigma = sigma / 1.414
    print(f"Current sigma value: {sigma}")
    blurred = cv2.GaussianBlur(gray_image, (0, 0), sigma)
    laplacian = cv2.Laplacian(blurred, cv2.CV_64F)
    abs_laplacian = np.abs(laplacian)
    blob_mask = abs_laplacian > threshold * abs_laplacian.max()
    contours, _ = cv2.findContours(blob_mask.astype(np.uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
        )
    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))
```

Listing 1: Blob Detection using LoG and Contour Fitting

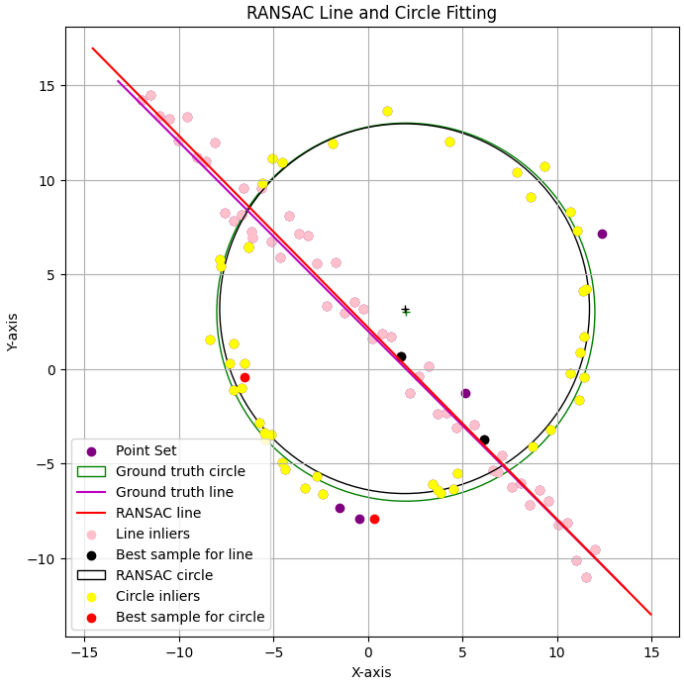


2 Line and circle fitting using RANSAC

- **Best line:** $[0.7121, 0.7021, 1.5328]$
  No. of inliers = 52, Best error = 8.0987
- **Best circle:** $[1.7751, 3.1418, 9.9013]$
  No. of inliers = 45, Best error = 8.0338

If the circle is fitted first,

There is a possibility that the three randomly chosen sample points might lie on the line, resulting in a large circle that resembles a line locally. However, because RANSAC runs for many iterations with varying sample points, it can still accurately fit the circle without needing to remove the line points.



RANSAC Parameters:

- **Minimum number of points (s):**
  - Line: 2
  - Circle: 3
- **Error threshold (t):**
  - Line: 1
  - Circle: 1.2
- **Consensus size (d):**
  - Line: 40
  - Circle: 40

Figuur 1: RANSAC Fitting Results and Parameters

```python
# Squared error calculation for line and circle
def tls_error_line(params, *args):
    a, b, d = params
    indices, X = args
    error = np.sum((a * X[indices, 0] + b * X[indices, 1] - d)**2)
    return error
def tls_error_circle(params, *args):
    cx, cy, r = params
    indices, X = args
    error = np.sum((dist((cx, cy), (X[indices, 0], X[indices, 1])) - r)**2)
    return error

def consensus_line(params, thres, X):
    a, b, d = params
    errors = np.abs(a * X[:, 0] + b * X[:, 1] - d)
    return np.where(errors < thres)[0]  # Return the indices of inliers
def consensus_circle(params, thres, X): # Determine inliers
    cx, cy, r = params
    errors = np.abs(dist((cx, cy), (X[:, 0], X[:, 1])) - r)  # Radial error
    return np.where(errors < thres)[0]  # Return indices of inliers
def constraint(params): # Constraint to normalize line parameters
    a, b, d = params
    return (a**2 + b**2)**0.5 - 1  # Should equal zero
def least_squares_line_fit(indices, initial, X):   # Fit a line using least squares optimization
    res = minimize(fun=tls_error_line, x0=initial, args=(indices, X), constraints=constraint_dict, tol=1e
        -6)
    print(res.x, res.fun)  # Print fitted parameters and error
    return res
def least_squares_circ_fit(indices, initial, X):   # Fit a circle using least squares optimization
    res = minimize(fun=tls_error_circle, x0=initial, args=(indices, X), tol=1e-6)
    print(res.x, res.fun)  # Print fitted parameters and error
    return res
# Fitting the line
iters = 100  # Number of iterations for fitting
min_points = 2  # Minimum points needed to define a line
N = X.shape[0]  # Total number of points
np.random.seed(14)  # Set random seed for reproducibility
thres = 1.  # Error threshold for inlier selection
d = 0.4 * N  # Minimum number of inliers for a valid fit
for i in range(iters):
    # Randomly select two points to define the line
    indices = np.random.choice(np.arange(0, N), size=min_points, replace=False)
    params = line_eq(X[indices[0]], X[indices[1]])  # Calculate line parameters
```

```
    inliers = consensus_line(params, thres, X)  # Get indices of inliers
    num_inliers = len(inliers)  # Get number of inliers from the array
    if num_inliers >= d:  # Check if inlier count meets the threshold
        res = least_squares_line_fit(inliers, params, X)  # Optimize the line fit
        if res.fun < best_error:  # Update best fit if error is lower
            best_error = res.fun
            best_model_line = params
            best_fitted_line = res.x
            best_line_inliers = inliers
            best_line_sample_points = indices

# Fitting the circle, Get remaining points
remaining_points = np.setdiff1d(np.arange(N), line_inliers)  # More efficient
X_rem = X[remaining_points]
iters = 100
min_points = 3  # Minimum points for circle estimation
thres = 1.2  # Error threshold
d = 0.4 * N  # Minimum inlier count
for i in range(iters):
    indices = np.random.choice(len(X_rem), size=min_points, replace=False)
    params = circle_eq(*X_rem[indices])  # Calculate circle parameters
    inliers = consensus_circle(params, thres, X_rem)  # Get inliers
    # Ensure inliers is treated correctly
    if isinstance(inliers, tuple) and len(inliers) > 0:
        inlier_indices = inliers[0]  # Get inlier indices
    elif isinstance(inliers, np.ndarray):
        inlier_indices = inliers
    else:
        inlier_indices = np.array([])  # Set to empty array if not valid
    num_inliers = len(inlier_indices)  # Count inliers
    if num_inliers >= d:  # Check inlier count
        res = least_squares_circ_fit(inlier_indices, params, X_rem)  # Optimize circle fit
        if res is not None and res.fun < best_error:  # Check validity of res
            best_error = res.fun
            best_model_circle = params
            best_fitted_circle = res.x
            best_circ_inliers = inlier_indices
            best_circ_sample_points = indices
```

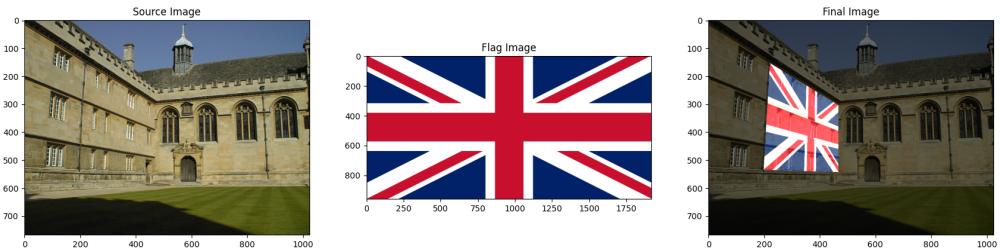
Listing 2: Line and Circle fitting using RANSAC

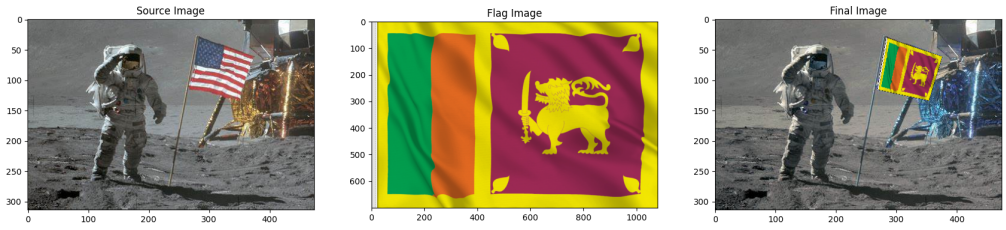
3 Superimposing an image on another

```
h, status = cv.findHomography(p, p_flag) # Calculating homography between image and flag
# Warping image of flag
warped_img = cv.warpPerspective(image_superimposed, np.linalg.inv(h), (image_background.shape[1],
    image_background.shape[0]))
blended = cv.addWeighted(image_background, 0.5, warped_img, 0.9, 0.0)
```

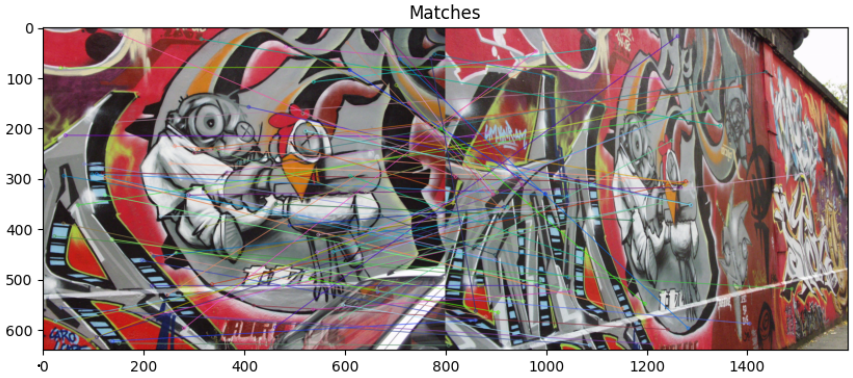
A mouse callback function is used to collect points from mouse clicks on background image. The coordinates of the clicked points are stored in the p array, while the corner locations of the flag image are stored in the p_{flag} array. Then computes the homography matrix to map the flag's corners to the selected points. The flag image is then warped using this matrix and blended with the background image to create the final output.

For my images I created a mask on the background image based on the selected pixel range, which sets the corresponding area to black. The flag image is then warped using this mask, allowing for accurate placement on the background without blending the two images.





4 Image stitching



First, I computed the SIFT matching between image 1 and image 5 using a threshold of 0.75 to identify good matches.

Then homography between images was computed using the `calculateHomography` function. Initially, the calculated homography showed significant differences from the provided homography. To improve accuracy, individual homographies for five images were calculated and then multiplied sequentially to obtain the final homography between images one and five. Comparing this final calculated homography with the provided one indicates that they are quite similar but still have some differences.

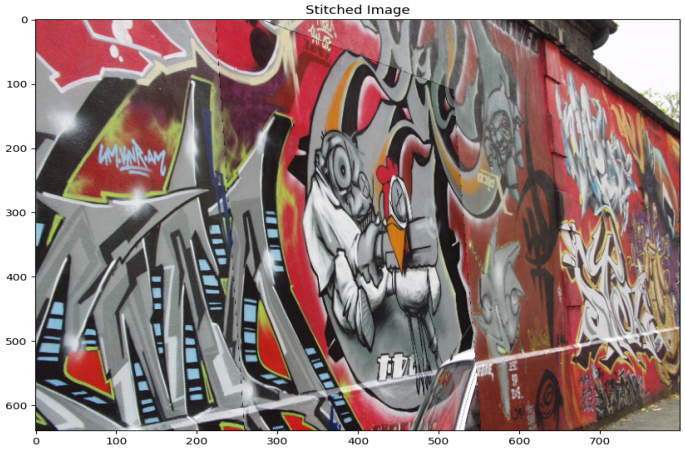
```python
def compute_homography(correspondences):
    matrix_list = []
    for point_set in correspondences:
        point1 = np.matrix([point_set.item(0),
            point_set.item(1), 1])
        point2 = np.matrix([point_set.item(2),
            point_set.item(3), 1])
        row1 = [-point2.item(2) * point1.item(0),
            -point2.item(2) * point1.item(1), -
            point2.item(2) * point1.item(2), 0, 0,
             0,
                point2.item(0) * point1.item(0),
                    point2.item(0) * point1.item
                    (1), point2.item(0) * point1.
                    item(2)]
        row2 = [0, 0, 0, -point2.item(2) * point1.
            item(0), -point2.item(2) * point1.item
            (1), -point2.item(2) * point1.item(2),
                point2.item(1) * point1.item(0),
                    point2.item(1) * point1.item
                    (1), point2.item(1) * point1.
                    item(2)]
        matrix_list.append(row1)
        matrix_list.append(row2)
    matrix_to_solve = np.matrix(matrix_list)
    u, s, v = np.linalg.svd(matrix_to_solve)
    homography_matrix = np.reshape(v[8], (3, 3))
    homography_matrix = (1 / homography_matrix.
        item(8)) * homography_matrix
    return homography_matrix
```

Provided Matrix

$$\begin{bmatrix} 6.25446 \times 10^{-1} & 5.77591 \times 10^{-2} & 2.22012 \times 10^{2} \\ 2.22405 \times 10^{-1} & 1.16521 \times 10^{0} & -2.56056 \times 10^{1} \\ 4.92125 \times 10^{-4} & -3.65424 \times 10^{-5} & 1.00000 \times 10^{0} \end{bmatrix}$$

Calculated Matrix

$$\begin{bmatrix} 7.25696 \times 10^{-1} & -1.21646 \times 10^{0} & 8.31404 \times 10^{1} \\ 1.35638 \times 10^{-1} & -9.94981 \times 10^{0} & 4.31717 \times 10^{1} \\ 8.21085 \times 10^{-4} & -4.46336 \times 10^{-3} & 1.00000 \times 10^{0} \end{bmatrix}$$



Figuur 2: Homography calculation, matrices comparison and Final stitched output

In the final stitched image, there is a part of a vehicle showing, which can be seen at the bottom between pixels 500 and 600.