# Assignment 02

Alignment and Fitting

submitted for

## EN3160 - Image Processing and Machine Vision

Department of Electronic and Telecommunication Engineering
University of Moratuwa

**Udugamasooriya P. H. J.**
220658U
Progress on GitHub

9 September 2025

**Question 01** We construct the scale-normalized LoG kernel, convolve the image with it at several scales, and square the response at each scale, to construct its squared normalized LoG scale-space $S[x, y, \sigma]$. Then, we flag each local extremum $[x, y, \sigma]$ of $S$ as a blob at $[x, y]$ with radius $\sigma\sqrt{2}$.

```
# Constructing the LoG kernel; snippet from LoG_kernel function
R2 = X**2 + Y**2 # X, Y formed by np.meshgrid
log = ((R2 - 2*sigma**2) / (2*np.pi*sigma**6)) * np.exp(-R2 / (2*sigma**2))
# ...
# Constructing the normalized LoG scale-space; snippet from LoG_scale_space function
for (i, sigma) in enumerate(sigmas):
       log = LoG_kernel(sigma, int(6*sigma + 1)) * (sigma**2)
       scale_space[:, :, i] = cv2.filter2D(im, -1, log)
# ...
# Squaring the response at each scale
S = LoG_scale_space(im, sigmas)
for i in range(len(sigmas)): S[:, :, i] = np.square(S[:, :, i])
# ...
# Snippet from local_maxima_3D function; accept V[x, y, z], loop over all x, y, z and:
cube = V[y - K:y + K + 1, x - K:x + K + 1, z - K:z + K + 1] # Look around a (2K+1)^3 nbhd
cube_max = np.max(cube)
if (cube_max == cube[0, 0, 0]) and (np.count_nonzero(cube == cube_max) == 1) and (cube_max >
    thresh):
       maxima.append((x, y, z))
# ...
# Plotting the detected blobs
for blob in local_maxima_3D(S, 3, 0.105): # 0.105 = threshold found experimentally
    x, y, k = blob
    sigma = sigmas[k]
    ax.add_patch(plt.Circle((x, y), np.sqrt(2)*sigma))
```

We construct the scale-space with $\sigma \in [1, 11]$. Figure 1 shows the blobs detected at each value of $\sigma$, with a circle with radius $\sigma\sqrt{2}$ round it.

Figure 1: Blobs detected at various scales

The blob with the largest radius was found corresponding to $\sigma = 8$, with a radius of around $11.31$ pixels.

Clearly, **Question 02** The relevant portion of the `RANSAC` function implementing RANSAC is as follows:

```
for _ in range(iterations):
        samples = data[np.random.randint(0, len(data), num_samples)] # Sample randomly
        params = params_from_samples(samples) # Compute parameters
        inliers = data[np.abs(error_given_params(params, data)) < inlier_thresh]
        if ((len(inliers) > max_inlier_count)) and (len(inliers) >= min_inlier_count):
                max_inlier_count, best_samples, best_inliers, best_params = len(inliers), samples,
                    inliers, params
```

Further, if the inlier-ratio is $r$, we iterate $M$ times, sample $N$ points, and want to hit at least one set of inliers with probability $p$,

$$M > \frac{\log(1-p)}{\log(1-r^N)}.$$

1. We ensure that $a^2 + b^2 = 1$ by parametrizing the line as $x\cos\theta + y\sin\theta = d$. The error for a point $(x, y)$ is computed as $|x\cos\theta + y\sin\theta - d|$.

   We define functions to find these parameters given two points, and to compute the error given a model and a data set, to be passed to `RANSAC` as callbacks.

   For a line, $N = 2$. Setting $p = 0.99$ and $r = 0.5$, we find $M > 16$. We run RANSAC with $M = 1000$; `RANSAC(X, 2, line_params_from_samples, line_error, 1, 1000)`; and obtain the following parametrization of the line:

   $$0.70x + 0.71y = 1.52.$$

2. We parametrize a circle as $(x-h)^2 + (y-k)^2 = r^2$, and define the error for a point $(x, y)$ as $\left|r - \sqrt{(x-h)^2 + (y-k)^2}\right|$.

   For a circle, $N = 3$. If sampling from the raw data set without removing line-inliers, for $p = 0.99$ and $r = 0.5$, we need $M > 34$. But, we remove line-inliers; `remaining_points = np.array([x for x in X if x not in line_best_inliers])`; and run RANSAC with $M = 1000$ and suitably defined callbacks; `RANSAC(remaining_points, 3, circle_params_from_samples, circle_error, 1, 1000)`; and obtain the following parametrization of the circle:

   $$(x - 2.12)^2 + (y - 3.01)^2 = 10.07^2.$$

Parameters for both models found above were further optimized by performing regression:

$$0.70x + 0.71y = 1.31, \quad \text{and} \quad (x - 2.13)^2 + (y - 2.85)^2 = 10.01^2.$$

3.

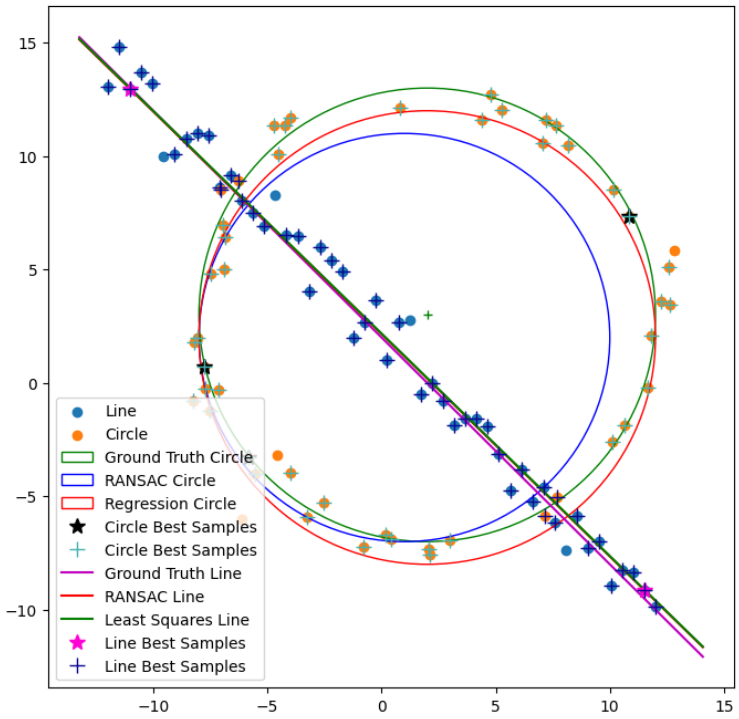


Figure 2: RANSAC models and regression-optimized models computed from a noisy point set

4. With enough iterations, it would not matter whether we try to fit the circle first or the line first; RANSAC will discover a correct set of inliers inliers eventually, assuming there are sufficiently many inliers. Trying this out in the code also yielded very similar results to the above.

   However, the probability that we find at least one set of inliers in $M$ iterations is $1 - \left(1 - r^N\right)^M$. For fixed $M$ (and $r$), this probability is greater for the line ($N = 2$) than for the circle ($N = 3$). Hence, if $M$ is small, and we try to fit the circle first, we are more likely to pick up wrong points, possibly even points belonging to the line, and end up with a wrong estimate.

**Question 03** We start with four point correspondences, and compute the homography matrix defining the required mapping of points across two images with the Direct Linear Transform (DLT) algorithm. The following condensed code snippet shows the basic steps followed, after this matrix is computed:

```
H = compute_homography(correspondences)
warped_overlay = cv2.warpPerspective(overlay_img, H, (base_img.shape[1], base_img.shape[0]))
tgt_pts = np.array([tgt_pt for (_, tgt_pt) in correspondences])
base_img = base_img.copy()
cv2.fillConvexPoly(base_img, tgt_pts, (0, 0, 0))
final_img = cv2.add(base_img, warped_overlay)
```

In Figure 3 we show the results of this activity in two different cases.


Figure 3: Overlaying images on planar surfaces

**Question 04** While the question was attempted using `img5.ppm`, proper results could not be obtained. Even with built-in functions of OpenCV such as `cv2.findHomography` and `cv2.Stitcher_create`, the computed homography matrix was very far from the ground-truth or enough matches could not be found. Therefore, we present results obtained with `img3.ppm` and `img1.ppm`.

The following code snippet shows the computation of SIFT features, matching descriptors across the images, and picking out the best matches, to get a set of point correspondences:

```
src_kp, src_des = sift.detectAndCompute(im1, None)
tgt_kp, tgt_des = sift.detectAndCompute(im2, None)
bf = cv2.BFMatcher()
matches = bf.knnMatch(src_des, tgt_des, k=2)
matches = [m for (m, n) in matches if m.distance < 0.75*n.distance]
```

We use RANSAC on these correspondences to construct the homography matrix relating the two images, and optimize it further through regression. The ground-truth matrix $\mathbf{H}_{gt}$, RANSAC-estimate $\mathbf{H}_0$, and regression-optimized estimate $\mathbf{H}_1$ obtained, were as follows:

$$\mathbf{H}_{gt} = \begin{bmatrix} 0.76 & -0.30 & 225.67 \\ 0.33 & 1.01 & -77.00 \\ 0.00 & -0.00 & 1.00 \end{bmatrix}, \mathbf{H}_0 = \begin{bmatrix} 0.39 & -0.30 & 220.92 \\ 0.00 & 0.42 & 124.56 \\ -0.00 & -0.00 & 1.00 \end{bmatrix}, \mathbf{H}_1 = \begin{bmatrix} 0.76 & -0.28 & 223.71 \\ 0.33 & 1.03 & -78.86 \\ 0.00 & 0.00 & 1.00 \end{bmatrix}.$$

Some entries in the RANSAC estimate are way off; with regression, this error has significantly reduced. The following code snippet shows the important steps followed:

```
im1_warped = cv2.warpPerspective(im1, T @ H, (w3, h3)) # T = from im2 to im3, H = from im1 to im2
mask = cv2.warpPerspective(np.ones_like(im1), T @ H, (w3, h3))
im3 = cv2.warpPerspective(im2, T, (w3, h3))
im3[mask == 1] = 0
im3 = cv2.add(im3, im1_warped)
```

Figure 4 shows the final stitched image, along with intermediate results.

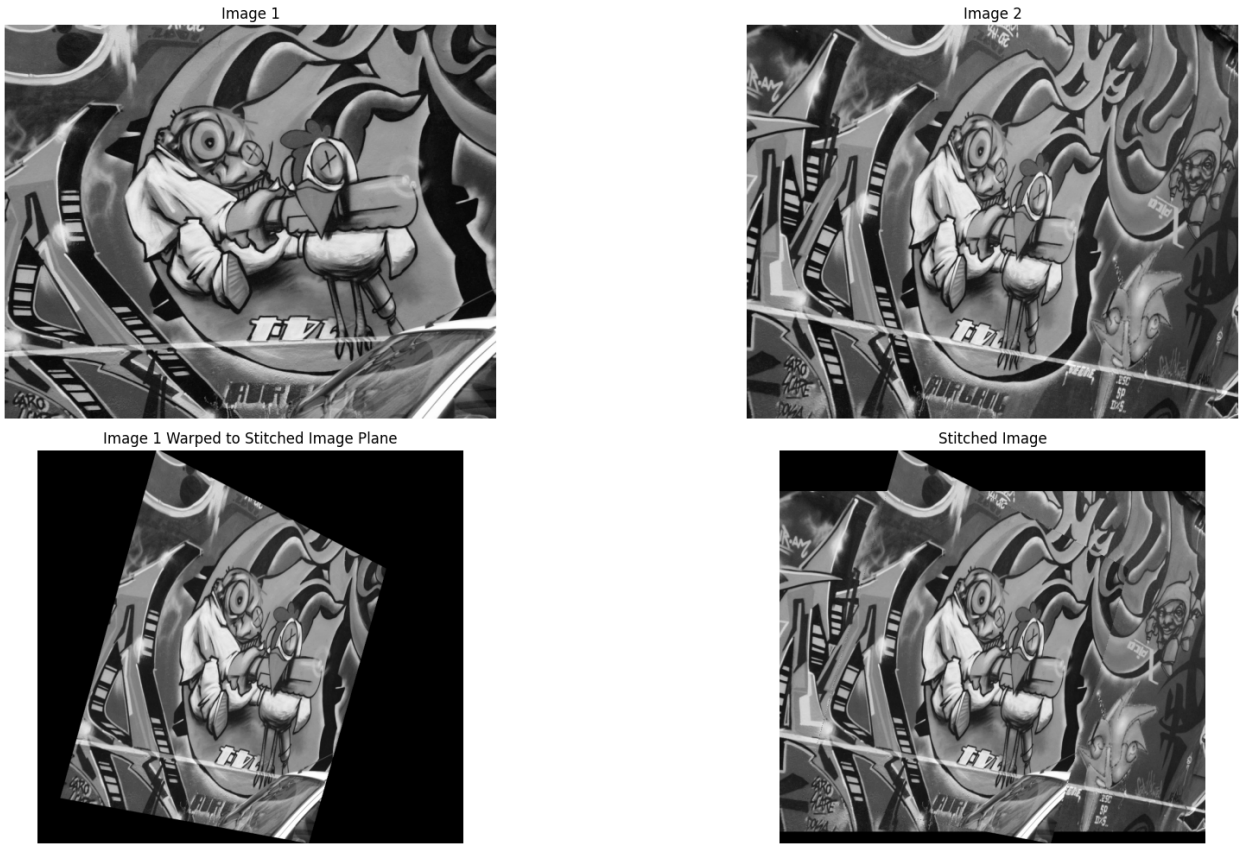


Figure 4: Stitching `img3.ppm` and `img1.ppm`