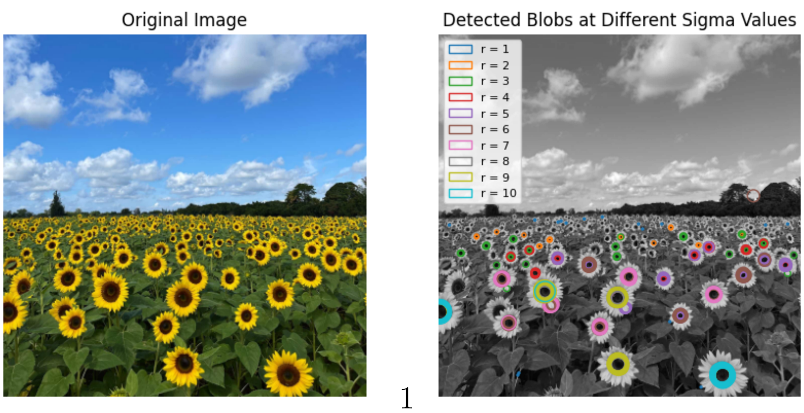
# EN3160 – Image Processing and Machine Vision
# Assignment 2 – Fitting and Alignment

Name: Sehara G.M.M.
Index No.: 210583B
GitHub Repository: GitHub Link
Submission Date: 23rd October 2024

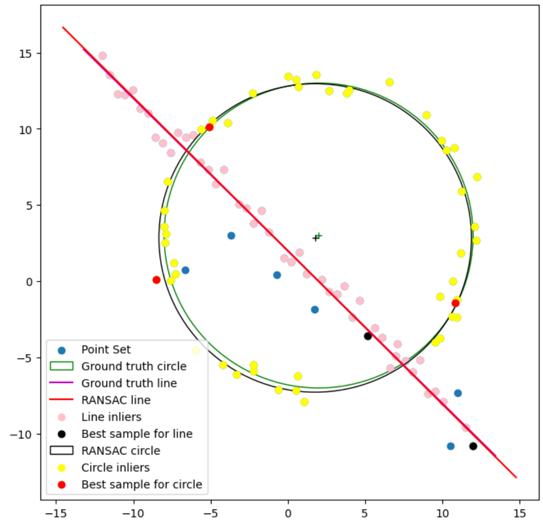
## 1 Question 01 - Blob Detection

To optimally detect a blob with radius $r$, the relationship $\sigma = \frac{r}{\sqrt{2}}$ was applied. The values of $\sigma$ corresponding to $r$ within the range of 1 to 10 were used for blob detection.

```
def laplace_of_gaussian(sigma):
    # Calculate kernel half-width
    hw = int(3 * sigma)
    # Create 2D coordinate grid
    x = np.arange(-hw, hw + 1)
    X, Y = np.meshgrid(x, x)
    # Precompute common terms for clarity
    norm_factor = 1 / (np.pi * sigma**4)
    exp_term = np.exp(-(X**2 + Y**2) / (2 * sigma**2))
    scale_term = (X**2 + Y**2) / (2 * sigma**2) - 1
    # Calculate Laplacian of Gaussian
    log = norm_factor * scale_term * exp_term
    return log

def detect_max(img_log, sigma):
    coordinates = set()
    k = 1 # Radius for neighborhood
    threshold = 0.09
    # Get the dimensions of the image
    h, w = img_log.shape
    # Iterate over the valid region to avoid boundary issues
    for i in range(k, h - k):
        for j in range(k, w - k):
            # Extract the local neighborhood
            patch = img_log[i - k:i + k + 1, j - k:j + k + 1]
            # Check if the max value meets the threshold
            if patch.max() >= threshold:
                # Get the coordinates of the max value within the patch
                x, y = np.unravel_index(np.argmax(patch), patch.shape)
                # Adjust to the image's coordinate space and store the result
                coordinates.add((i + x - k, j + y - k))

    return coordinates
```



# 2 Question 2 – Line and circle fitting using RANSAC




**Parameters of RANSAC used:**

For the line, $s = 2$, and for the circle, $s = 3$. These values represent the minimum number of points needed to accurately determine each shape.

The error threshold is set to $t = 1$ for the line and $t = 1.2$ for the circle. Since the noise characteristics of the points are unknown, the best threshold value was identified through experimentation.

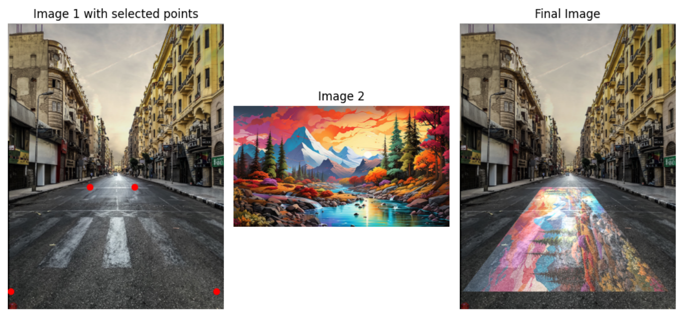
The consensus size is $d = 40$ for both the line and the circle. With 50 points associated with each shape in the dataset, a threshold of $d = 40$ effectively captures a sufficient number of inliers.

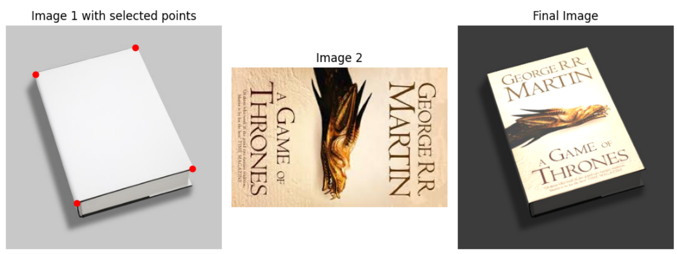
```python
# Squared error calculation for line and circle
def tls_error_line(params, *args):
    """Calculate the total least squares error for a line."""
    a, b, d = params
    indices, X = args
    errors = a * X[indices, 0] + b * X[indices, 1] - d
    return np.sum(errors**2)

def tls_error_circle(params, *args):
    """Calculate the total least squares error for a circle."""
    cx, cy, r = params
    indices, X = args
    distances = dist((cx, cy), (X[indices, 0], X[indices, 1]))
    return np.sum((distances - r)**2)

def least_squares_line_fit(indices, initial, X):
    """Perform least squares line fitting using SciPy's minimize."""
    res = minimize(fun=tls_error_line, x0=initial, args=(indices, X),
                   constraints=constraint_dict, tol=1e-6)
    print(res.x, res.fun)
    return res

# Fitting the line using RANSAC (procedure is similar for circle)
for i in range(iterations):
    # Randomly select points to fit the line
    indices = np.random.choice(np.arange(num_points), size=min_inlier_points, replace=
        False)
    params = line_eq(X[indices[0]], X[indices[1]])
    # Determine inliers based on the consensus function
    inliers = consensus_line(params, error_threshold, X)[0]
    print(f'Iteration {i}: No. of inliers = {len(inliers)}')
    # Recompute if the number of inliers meets the requirement
    if len(inliers) >= min_inlier_required:
        res = least_squares_line_fit(inliers, params, X)
        # Update the best model if the error is lower
```

```
        if res.fun < best_error:
            best_error = res.fun
            best_model_line = params
            best_fitted_line = res.x
            best_line_inliers = inliers
            best_sample_points = indices
# Final consensus for the best fitting line
final_line_inliers = consensus_line(best_fitted_line, 1.2, X)[0]
```

If the circle is fitted first, there's a possibility that the three random points may all lie on the line, resulting in a large circle resembling a line. However, since RANSAC runs multiple iterations with different samples, it can still accurately fit the circle without needing to remove the line points.

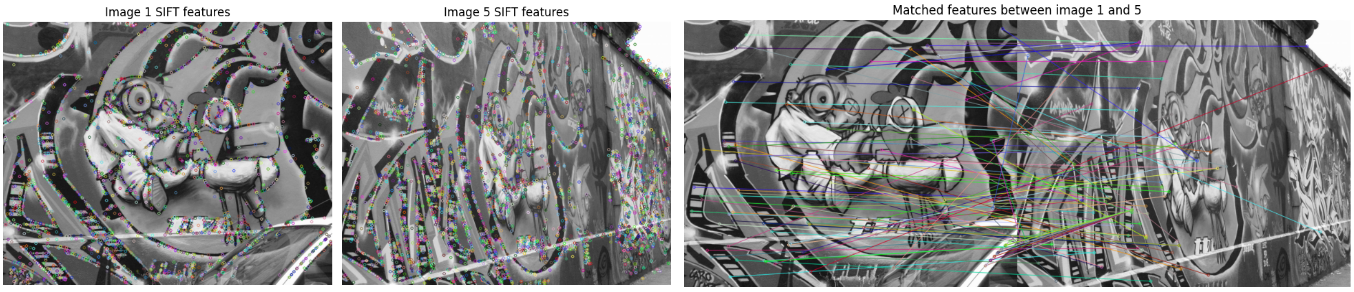
# 3 Question 3 – Superimposing an image on another




```
def superimpose(image, logo, dst_points, beta=0.3, alpha=1):
    # Get logo dimensions
    logo_height, logo_width, _ = logo.shape
    # Define source points for the logo's corners
    src_points = np.array([(0, logo_height), (logo_width, logo_height),
                           (logo_width, 0), (0, 0)]) # bl, br, tr, tl
    # Pad the logo if it's smaller than the destination image
    if logo_height <= image.shape[0]:
        logo = np.pad(logo, ((0, image.shape[0] - logo_height), (0, 0), (0, 0)), 'constant')
    if logo_width <= image.shape[1]:
        logo = np.pad(logo, ((0, 0), (0, image.shape[1] - logo_width), (0, 0)), 'constant')
    # Estimate the transformation
    tform = transform.estimate_transform('projective', src_points, dst_points)
    tf_img = transform.warp(logo, tform.inverse)
    tf_img = (tf_img * 255).astype(np.uint8)
    # Crop the transformed logo if it's larger than the destination image
    tf_img = tf_img[:image.shape[0], :image.shape[1]] if logo_height > image.shape[0] or
        logo_width > image.shape[1] else tf_img
    # Blend the images together
    blended_image = cv.addWeighted(image, alpha, tf_img, beta, 0)
    return blended_image
```

# 4 Question 4 – Image stitching




There are insufficient matching features between images 1 and 5 to compute a homography accurately. As a result, homographies were calculated between more similar pairs of images, such as 1-2, 2-3, and so on. Image 1 was then transformed through all these mappings to align it with image 5.



**Computed Homography**

$$\begin{bmatrix} 6.11404405\times10^{-1} & 5.03189278\times10^{-2} & 2.21391689\times10^{2} \\ 2.11980188\times10^{-1} & 1.14096497\times10^{0} & -2.14952560\times10^{1} \\ 4.74861259\times10^{-4} & -5.18621428\times10^{-5} & 9.90604813\times10^{0} \end{bmatrix}$$

**Actual Homography**

$$\begin{bmatrix} 6.2544644\times10^{-1} & 5.7759174\times10^{-2} & 2.2201217\times10^{2} \\ 2.2240536\times10^{-1} & 1.1652147\times10^{0} & -2.5605611\times10^{1} \\ 4.9212545\times10^{-4} & -3.6542424\times10^{-5} & 1.0000000\times10^{0} \end{bmatrix}$$

```python
# Perform RANSAC
for i in range(iters):
    chosen_matches = np.random.choice(good_matches, num_points, replace=False)
    src_points = np.array([np.array(keypoints1[match.queryIdx].pt) for match in
        chosen_matches])
    dst_points = np.array([np.array(keypoints5[match.trainIdx].pt) for match in
        chosen_matches])
    # Estimate projective transformation
    tform = transform.estimate_transform('projective', src_points, dst_points)
    # Calculate inliers using the get_inliers function
    inliers = get_inliers(src_full, dst_full, tform, thres)
    # Update best homography if more inliers are found
    if len(inliers) > best_inlier_count:
        best_inlier_count = len(inliers)
        best_homography = tform
        best_inliers = inliers
print(f'Best no. of inliers = {best_inlier_count}')
return best_homography, best_inliers
```