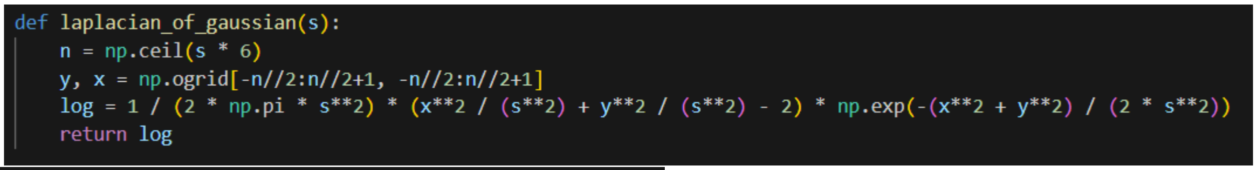
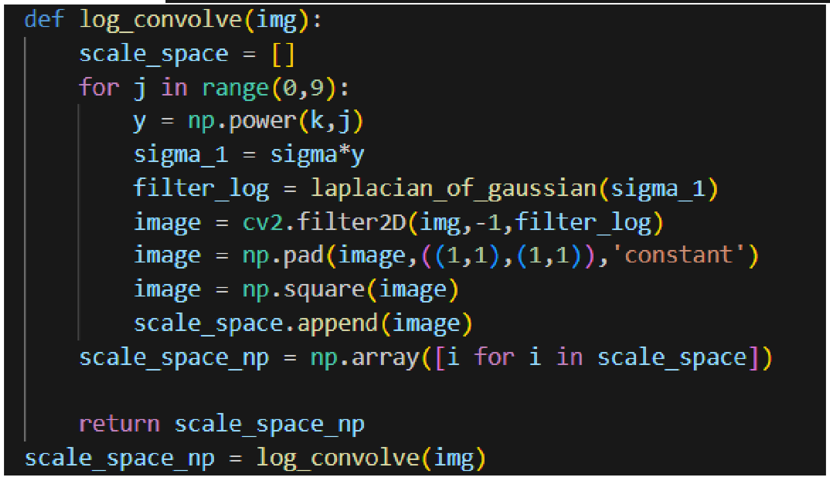
EN3160 Assignment 02 M. Navindu Gunawardena 200201V

Q - 1

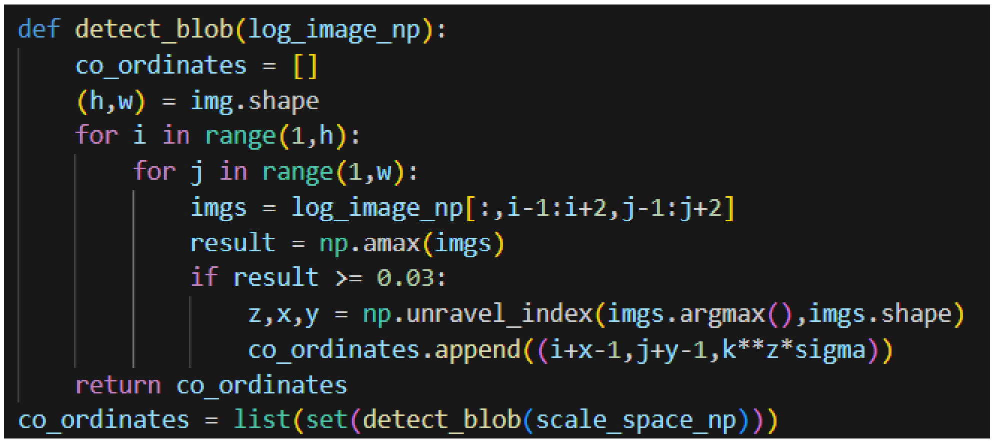


```python
def laplacian_of_gaussian(s):
    n = np.ceil(s * 6)
    y, x = np.ogrid[-n//2:n//2+1, -n//2:n//2+1]
    log = 1 / (2 * np.pi * s**2) * (x**2 / (s**2) + y**2 / (s**2) - 2) * np.exp(-(x**2 + y**2) / (2 * s**2))
    return log
```



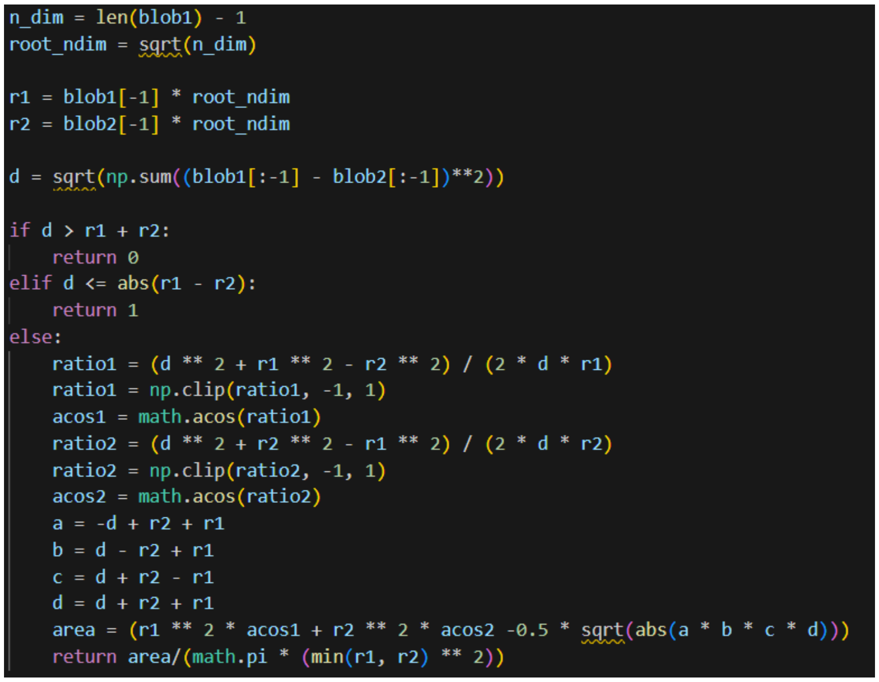
```python
def log_convolve(img):
    scale_space = []
    for j in range(0,9):
        y = np.power(k,j)
        sigma_1 = sigma*y
        filter_log = laplacian_of_gaussian(sigma_1)
        image = cv2.filter2D(img,-1,filter_log)
        image = np.pad(image,((1,1),(1,1)),'constant')
        image = np.square(image)
        scale_space.append(image)
    scale_space_np = np.array([i for i in scale_space])

    return scale_space_np
scale_space_np = log_convolve(img)
```

For blob detection the image was convolved with the Laplacian of a gaussian filter.



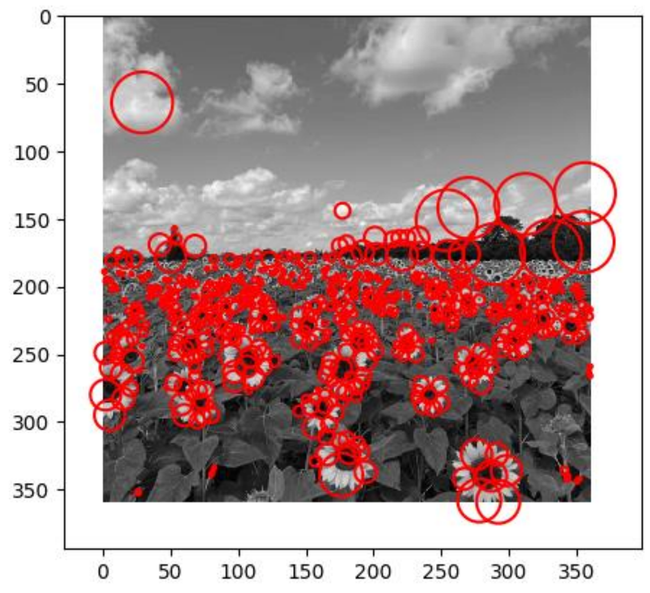
```python
def detect_blob(log_image_np):
    co_ordinates = []
    (h,w) = img.shape
    for i in range(1,h):
        for j in range(1,w):
            imgs = log_image_np[:,i-1:i+2,j-1:j+2]
            result = np.amax(imgs)
            if result >= 0.03:
                z,x,y = np.unravel_index(imgs.argmax(),imgs.shape)
                co_ordinates.append((i+x-1,j+y-1,k**z*sigma))
    return co_ordinates
co_ordinates = list(set(detect_blob(scale_space_np)))
```

Local extremums are used to find the blobs. Here this function loops over the scale space.

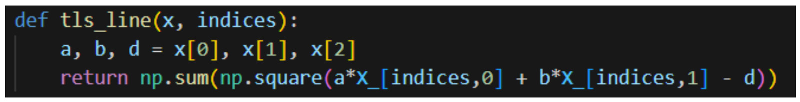


```python
n_dim = len(blob1) - 1
root_ndim = sqrt(n_dim)

r1 = blob1[-1] * root_ndim
r2 = blob2[-1] * root_ndim

d = sqrt(np.sum((blob1[:-1] - blob2[:-1])**2))

if d > r1 + r2:
    return 0
elif d <= abs(r1 - r2):
    return 1
else:
    ratio1 = (d ** 2 + r1 ** 2 - r2 ** 2) / (2 * d * r1)
    ratio1 = np.clip(ratio1, -1, 1)
    acos1 = math.acos(ratio1)
    ratio2 = (d ** 2 + r2 ** 2 - r1 ** 2) / (2 * d * r2)
    ratio2 = np.clip(ratio2, -1, 1)
    acos2 = math.acos(ratio2)
    a = -d + r2 + r1
    b = d - r2 + r1
    c = d + r2 - r1
    d = d + r2 + r1
    area = (r1 ** 2 * acos1 + r2 ** 2 * acos2 -0.5 * sqrt(abs(a * b * c * d)))
    return area/(math.pi * (min(r1, r2) ** 2))
```

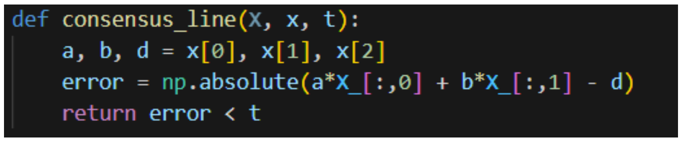
Overlapping blobs are removed.



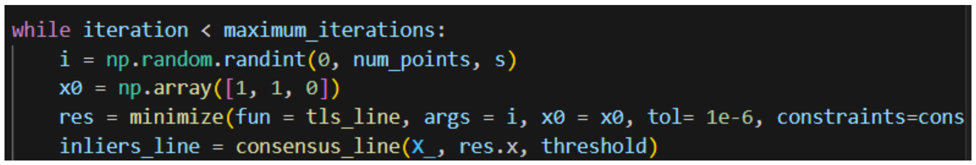
Q – 2



```
def tls_line(x, indices):
    a, b, d = x[0], x[1], x[2]
    return np.sum(np.square(a*X_[indices,0] + b*X_[indices,1] - d))
```



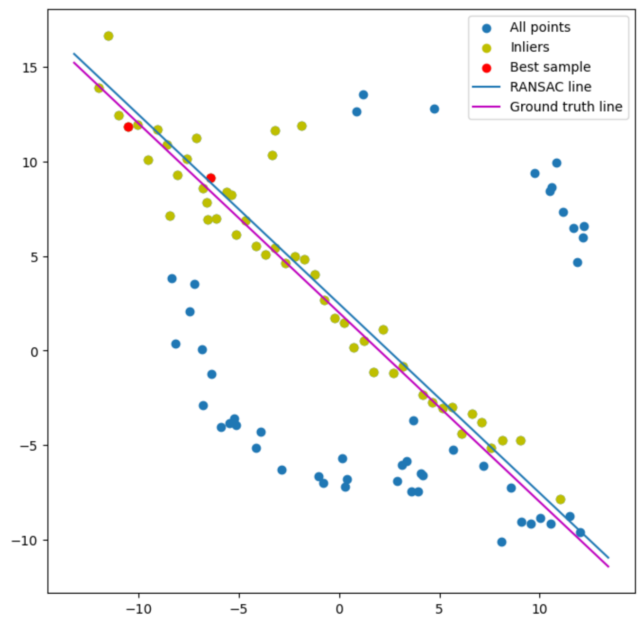
```
def consensus_line(X, x, t):
    a, b, d = x[0], x[1], x[2]
    error = np.absolute(a*X_[:,0] + b*X_[:,1] - d)
    return error < t
```

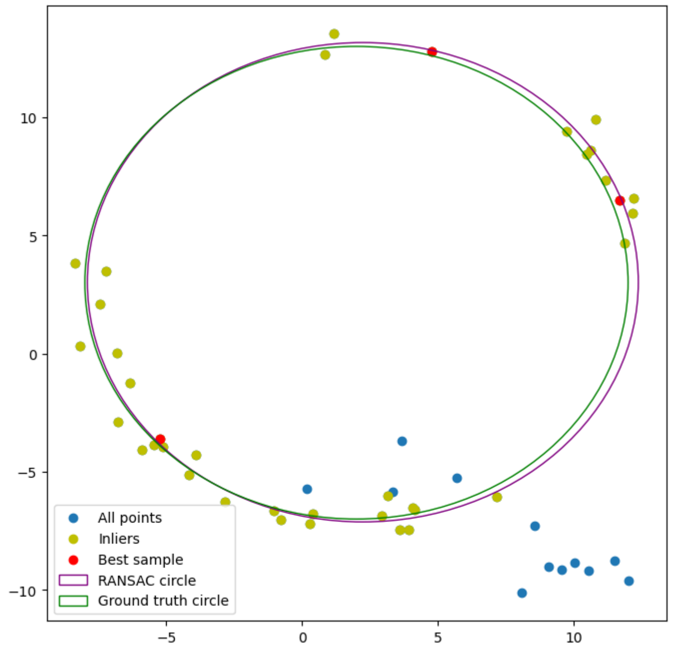


```
while iteration < maximum_iterations:
    i = np.random.randint(0, num_points, s)
    x0 = np.array([1, 1, 0])
    res = minimize(fun = tls_line, args = i, x0 = x0, tol= 1e-6, constraints=cons
    inliers_line = consensus_line(X_, res.x, threshold)
```

Here we estimate a line and a circle using the RANSAC method.

We first select a random sample with minimum no of points and then estimate the model parameters using this. We repeat this for a certain no of iterations and then finally select the better model. Then this model is refined .



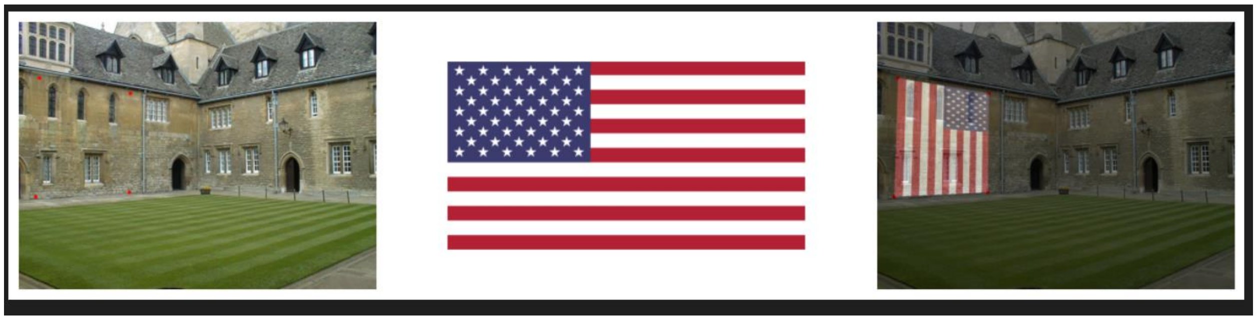


Q – 3

Here we superimpose an image on top of another. We do this by computing the homography using the below function.

```python
points_image1 = np.array(selected_points).astype('float32')
points_image2 = np.array([ [0,0], [image2.shape[1] , 0], [image2.shape[1], image2.shape[0]], [0, image2.shape[0]] ]).astype('float32')
```

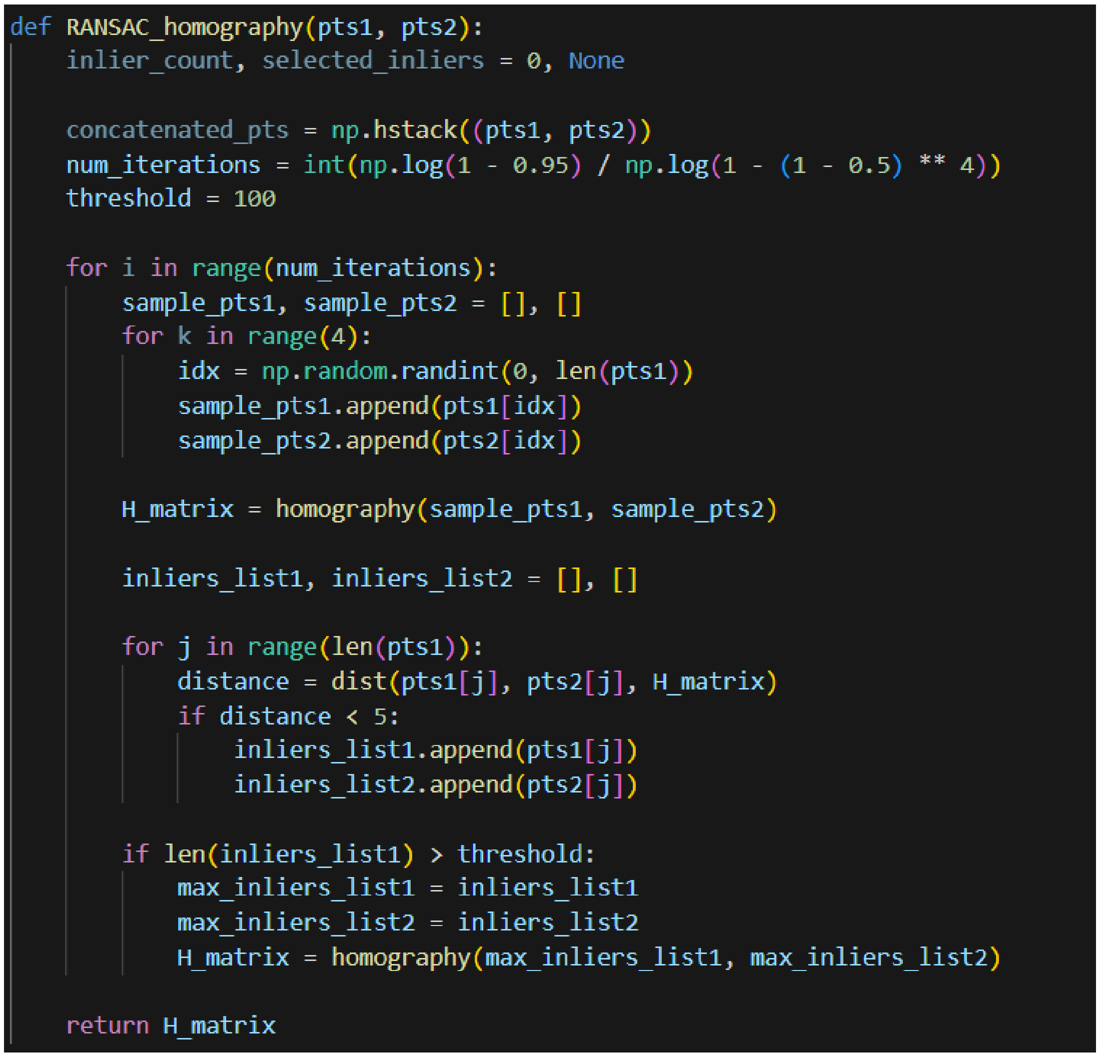
```python
homography_matrix= cv2.findHomography(points_image2 , points_image1)[0]

warped_image2 = cv2.warpPerspective(image2, homography_matrix, (image.shape[1], image.shape[0] ))

#using simple averaging to blend the images

alpha = 0.5
blended_image = cv2.addWeighted(image, alpha, warped_image2, 1- alpha, 0)
```



Q – 4

```python
def sift_match(img1, img2):

    GMP = 0.8

    # Detect sift features
    s = cv.SIFT_create(nOctaveLayers=3, contrastThreshold=0.1, edgeThreshold=25, sigma=1)
    kpts1, desc1 = s.detectAndCompute(img1, None)
    kpts2, desc2 = s.detectAndCompute(img2, None)

    # Match features.
    m = cv.BFMatcher()
    matches = m.knnMatch(desc1, desc2, k=2)

    # Filter good matches using ratio test in Lowe's paper
    good_matches, pts1, pts2 = [], [], []

    for m1, m2 in matches:
        if m1.distance < GMP * m2.distance:
            good_matches.append(m1)
            pts1.append(kpts1[m1.queryIdx].pt)
            pts2.append(kpts2[m1.trainIdx].pt)

    good_matches, pts1, pts2 = np.array(good_matches), np.array(pts1), np.array(pts2)

    # Plot the matching
    f, a = plt.subplots(figsize=(15, 15))
    a.axis('off')
    m_img = cv.drawMatches(img1, kpts1, img2, kpts2, good_matches, img2, flags=2)
    plt.imshow(cv.cvtColor(m_img, cv.COLOR_BGR2RGB))
    plt.show()

    return pts1, pts2
```

Here we use stitch 2 images by matching the SIFT features between the 2 images.

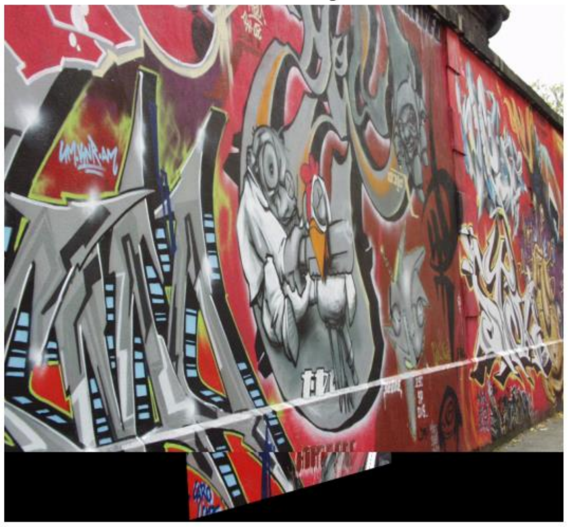


```
def RANSAC_homography(pts1, pts2):
    inlier_count, selected_inliers = 0, None

    concatenated_pts = np.hstack((pts1, pts2))
    num_iterations = int(np.log(1 - 0.95) / np.log(1 - (1 - 0.5) ** 4))
    threshold = 100

    for i in range(num_iterations):
        sample_pts1, sample_pts2 = [], []
        for k in range(4):
            idx = np.random.randint(0, len(pts1))
            sample_pts1.append(pts1[idx])
            sample_pts2.append(pts2[idx])

        H_matrix = homography(sample_pts1, sample_pts2)

        inliers_list1, inliers_list2 = [], []

        for j in range(len(pts1)):
            distance = dist(pts1[j], pts2[j], H_matrix)
            if distance < 5:
                inliers_list1.append(pts1[j])
                inliers_list2.append(pts2[j])

        if len(inliers_list1) > threshold:
            max_inliers_list1 = inliers_list1
            max_inliers_list2 = inliers_list2
            H_matrix = homography(max_inliers_list1, max_inliers_list2)

    return H_matrix
```

Homography is computed using RANSAC.

img5.ppm

Stitched Image



GitHub link - https://github.com/mnavindug/Machine-Vision-and-Image-Processing