EN3160 Assignment 2: Fitting and Alignment

Name: Ranaweera R.P.D.D.H.
Index: 220511N

October 17, 2025

Project Repository: Fitting-and-Alignment

Question 1: Blob Detection on Sunflower Field

```python
sigma_values = np.linspace(sigma_min, sigma_max, num_sigma)
log_images = []
for sigma in sigma_values:
    size = int(2*np.ceil(4*sigma)+1)
    blur = cv.GaussianBlur(gray, (size, size), sigma)
    log = cv.Laplacian(blur, cv.CV_64F, ksize=1)
    log_images.append(sigma**2*np.abs(log))
log_stack = np.stack(log_images, axis=-1)
local_max = log_stack == maximum_filter(log_stack, size=(3,3,2))
blob_mask = local_max & (log_stack > 0.05)
blobs = [(x,y,i,sigma_values[i],log_stack[y,x,i])
         for y,x,i in np.argwhere(blob_mask)]
blobs_sorted = sorted(blobs, key=lambda b:b[4], reverse=True)
output = im.copy()
for x,y,i,sigma,_ in blobs_sorted[:min(100,len(blobs_sorted))]:
    cv.circle(output, (int(x),int(y)), int(np.sqrt(2)*sigma),
              (0,0,255), 2)
```

Figure 1: Blob Detection using Scale-space LoG

Detected Sunflower Centers using LoG Blob Detection

Figure 2: Detected Blobs over Multiple Scales

LoG Scale-Space Parameters

σ range	[2, 20]
Scales	50
Shape	(360, 360, 50)
Total blobs	4611
Threshold	0.05
Neighborhood	(3, 3, 2)

Top 5 Largest Circles (by radius)

Rank	X	Y	σ	Radius	Response
1	47	0	20.000	28.28	0.102926
2	295	0	20.000	28.28	0.069862
3	68	31	20.000	28.28	0.075601
4	69	33	20.000	28.28	0.075608
5	310	50	20.000	28.28	0.065867

Code Explaination

LoG (Laplacian of Gaussian) detects blobs at different scales. Creates a scale-space by applying Gaussian blur with multiple σ values ($2 \rightarrow 20$) and computing the Laplacian. Finds local maxima in 3D (x, y, scale) using a maximum filter. Only maxima above a threshold are considered valid blobs.

Uses scale-normalized LoG:

$$\text{LoG}_{\text{norm}} = \sigma^2 \cdot |\text{Laplacian}|$$

Stacks responses into a 3D array: (height $\times$ width $\times$ scales). Extracts $(x, y, \text{scale\_index}, \sigma, \text{response})$ for each detected blob.

Sorts blobs by response strength or size (σ). Computes radius for visualization:

$$\text{radius} = \sqrt{2} \cdot \sigma$$

Listing 1: RANSAC Line Fitting

```python
def fit_line(points):
    """Fit line ax + by + d = 0 through two points."
        ""
    (x1, y1), (x2, y2) = points
    a = y1 - y2
    b = x2 - x1
    d = -(a * x1 + b * y1)
    norm = np.sqrt(a**2 + b**2)
    return a/norm, b/norm, d/norm

def line_distance(a, b, d, X):
    """Compute normal distance from line to all
        points."""
    return np.abs(a*X[:,0] + b*X[:,1] + d)

def ransac_line(X, n_iter=1000, threshold=0.6,
     min_inliers=35):
    best_inliers = []
    best_model = None
    for _ in range(n_iter):
        sample = X[np.random.choice(len(X), 2,
             replace=False)]
        a, b, d = fit_line(sample)
        distances = line_distance(a, b, d, X)
        inliers = X[distances < threshold]
        if len(inliers) > len(best_inliers):
            best_inliers = inliers
            best_model = (a, b, d)
    return best_model, best_inliers
```

Explanation: Randomly picks 2 points and fits a line $ax + by + d = 0$. Measures how far all other points are from that line. Points within threshold distance are inliers. Repeats this for n_{iter} times and keeps the line with the most inliers. **Output:** best-fitting line (a, b, d) and its inlier points.

Listing 2: RANSAC Circle Fitting

```python
def fit_circle(pts):
    """Fit a circle through 3 points."""
    A = np.array([[2*(pts[1,0]-pts[0,0]), 2*(pts[1,1]-pts[0,1])],
                  [2*(pts[2,0]-pts[0,0]), 2*(pts[2,1]-pts[0,1])]])
    b = np.array([(pts[1,0]**2 - pts[0,0]**2) +
                  (pts[1,1]**2 - pts[0,1]**2),
                  (pts[2,0]**2 - pts[0,0]**2) +
                  (pts[2,1]**2 - pts[0,1]**2)])
    center = np.linalg.solve(A, b)
    radius = np.sqrt((pts[0,0]-center[0])**2 +
                     (pts[0,1]-center[1])**2)
    return center, radius

def ransac_circle(X, n_iter=1000, threshold=1.5, min_inliers=45):
    best_inliers = []
    best_model = None
    for _ in range(n_iter):
        sample = X[np.random.choice(len(X), 3, replace=False)]
        try:
            center, r = fit_circle(sample)
        except np.linalg.LinAlgError:
            continue
        dist = np.sqrt((X[:,0]-center[0])**2 + (X[:,1]-center[1])**2)
        inliers = X[np.abs(dist - r) < threshold]
        if len(inliers) > len(best_inliers):
            best_inliers = inliers
            best_model = (center, r)
    return best_model, best_inliers
```

Explanation: Randomly picks 3 points and fits a circle passing through them. Finds circle center (c_x, c_y) and radius r. Checks how close all other points lie to that circle (distance from center $\approx r$). Keeps the circle model with the most inliers. **Output:** best circle (center, radius) and its inliers.

Question 2: Line and Circle Fitting with RANSAC

Line Estimation using RANSAC	
a	0.71
b	0.70
d	-1.48
Number of inliers	35

Circle Estimation using RANSAC	
Center (x_0, y_0)	(2.12, 2.35)
Radius r	9.88
Number of inliers	52

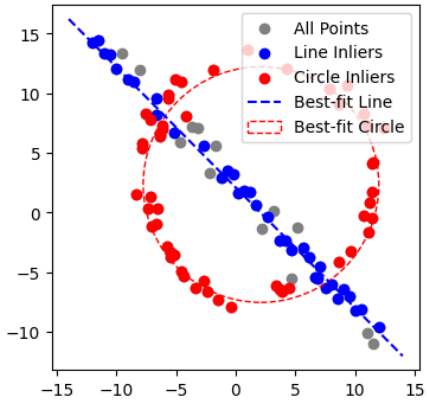


Figure 3: RANSAC line and circle fitting with inliers and sample points

(d) Circle-First Approach Analysis

If we fit the circle first, many line points appear as outliers and distort the estimate. The circle fitting RANSAC would try to include them, causing incorrect center and radius, poor inlier classification, and degraded line fitting due to misclassified points. Thus, it's better to fit the line first, remove its inliers, then fit the circle.

Question 3: Homography and Flag Superimposition

Listing 3: Homography Computation and Warping

```
def compute_homography(src_pts, dst_pts):
    assert src_pts.shape[0] == dst_pts.shape[0] >= 4
    n = src_pts.shape[0]
    A = []
    for i in range(n):
        x, y = src_pts[i]
        u, v = dst_pts[i]
        A.append([-x, -y, -1, 0, 0, 0, u*x, u*y, u])
        A.append([0, 0, 0, -x, -y, -1, v*x, v*y, v])
    A = np.array(A)
    U, S, Vt = np.linalg.svd(A)
    H = Vt[-1].reshape(3, 3)
    H = H / H[2, 2]
    return H

def warp_image(src_img, H, dst_shape):
    h, w = dst_shape[:2]
    warped = np.zeros((h, w, 3), dtype=np.uint8)
    mask = np.zeros((h, w), dtype=np.uint8)
    H_inv = np.linalg.inv(H)
    y_coords, x_coords = np.mgrid[0:h, 0:w]
    ones = np.ones_like(x_coords)
    dst_coords = np.stack([x_coords, y_coords, ones], axis=-1)
    dst_coords_flat = dst_coords.reshape(-1, 3).T
    src_coords_flat = H_inv @ dst_coords_flat
    src_x = src_coords_flat[0] / src_coords_flat[2]
    src_y = src_coords_flat[1] / src_coords_flat[2]
    src_x = src_x.reshape(h, w)
    src_y = src_y.reshape(h, w)
    src_h, src_w = src_img.shape[:2]
    valid_mask = (src_x >= 0) & (src_x < src_w - 1) & \
                 (src_y >= 0) & (src_y < src_h - 1)
    x0 = np.floor(src_x).astype(int); x1 = x0 + 1
    y0 = np.floor(src_y).astype(int); y1 = y0 + 1
    x0 = np.clip(x0, 0, src_w - 1); x1 = np.clip(x1, 0, src_w
         - 1)
```

Listing 4: Blending and Processing

```
    y0 = np.clip(y0, 0, src_h - 1); y1 = np.clip(y1, 0, src_h
         - 1)
    fx = src_x - x0; fy = src_y - y0
    for c in range(3):
        wa = (1 - fx) * (1 - fy); wb = fx * (1 - fy)
        wc = (1 - fx) * fy; wd = fx * fy
        warped[:, :, c] = (wa * src_img[y0, x0, c] +
                           wb * src_img[y0, x1, c] +
                           wc * src_img[y1, x0, c] +
                           wd * src_img[y1, x1, c])
    mask[valid_mask] = 255
    return warped, mask

def blend_images(dst_img, src_warped, mask, alpha=1.0):
    result = dst_img.copy()
    mask_3channel = cv.cvtColor(mask, cv.COLOR_GRAY2BGR) /
         255.0
    result = (1 - alpha * mask_3channel) * result + alpha *
         mask_3channel * src_warped
    return result.astype(np.uint8)

def process_image_pair(src_img, dst_img, dst_points=None,
     alpha=1.0):
    h_src, w_src = src_img.shape[:2]
    src_pts = np.array([[0,0],[w_src-1,0],[w_src-1,h_src
         -1],[0,h_src-1]],dtype=np.float32)
    dst_pts = np.array(dst_points,dtype=np.float32) if
         dst_points is not None else select_points_interactive(
         dst_img)
    H = compute_homography(src_pts, dst_pts)
    warped, mask = warp_image(src_img, H, dst_img.shape)
    result = blend_images(dst_img, warped, mask, alpha)
    return result, H, dst_pts, warped, mask
```

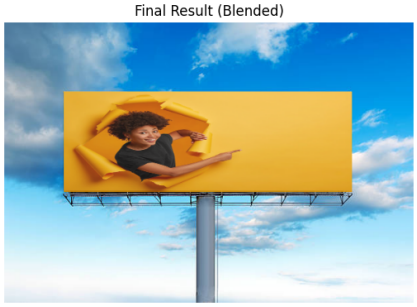
Code Explanation

Computes a 3×3 transformation matrix (H) that maps points from the source image to the destination plane. It uses the Direct Linear Transformation method and Singular Value Decomposition to solve: $Ah = 0$ where h is the 9-element vector of the homography.

Homography represents a projective transformation, handling rotation, translation, scaling, and perspective distortion.

Applies the inverse of H to map every pixel in the destination image back to where it comes from in the source. Uses bilinear interpolation for smooth pixel values (averaging nearby pixels). Only pixels that map inside the source image are valid Combines the warped source image with the destination using an alpha-blending formula.

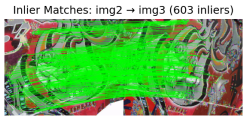
Homography: $\begin{bmatrix} 0.7950 & 0.0005649 & 88.0 \\ -0.001277 & 0.4167 & 103.0 \\ -1.2397 \times 10^{-5} & 6.4197 \times 10^{-6} & 1 \end{bmatrix}$



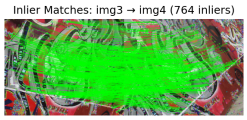
Homography: $\begin{bmatrix} 0.2201 & -0.08589 & 334.0 \\ 0.06482 & 0.11898 & 347.0 \\ 1.3509 \times 10^{-4} & -1.8554 \times 10^{-4} & 1 \end{bmatrix}$

Question 4: Image Stitching

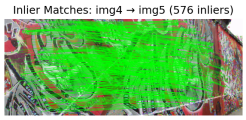
(a) img1→img2



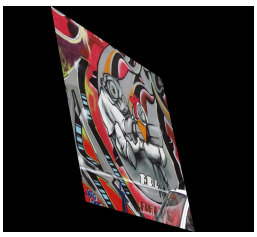
(b) img2→img3



(c) img3→img4



(d) img4→img5



(e) Warped img1

(f) Final Panorama: The final stitched panorama was created by warping img1 into img5's coordinate frame using the accumulated homography H_{15} and blending the two images. (a–d) show RANSAC inlier matches for each image pair, (e) shows img1 warped using H_{15}, and (f) displays the final panorama beside its caption.

Matrix	Mean Abs. Diff.
$H_{1\to2}$	0.0810
$H_{1\to3}$	5.071
$H_{1\to4}$	16.561
$H_{1\to5}$	8.681

Table: Mean absolute difference between practically obtained and reference homography matrices.

What the Code Does:

- **Sequential Pair Processing:** Extracts SIFT keypoints and descriptors for each image pair, followed by Lowe's ratio test for good matches.
- **Custom RANSAC:** Randomly selects four points to compute homography, projects all source points, and counts inliers ($error < threshold$) across 200 trials.
- **Homography Accumulation:** Combines H_{12}, H_{23}, H_{34}, and H_{45} to obtain H_{15}.
- **Final Stitching:** Uses H_{15} to warp img1 into img5, producing the final stitched panorama in (f).

SIFT Feature Matching:

```
def compute_sift_matches(img1, img2):
    img1_gray = cv.cvtColor(img1, cv.COLOR_BGR2GRAY)
    img2_gray = cv.cvtColor(img2, cv.COLOR_BGR2GRAY)
    sift = cv.SIFT_create(nOctaveLayers=3, contrastThreshold
        =0.04,
                          edgeThreshold=10, sigma=1.6)
    kp1, des1 = sift.detectAndCompute(img1_gray, None)
    kp2, des2 = sift.detectAndCompute(img2_gray, None)
    bf = cv.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)
    good = [m for m, n in matches if m.distance < 0.75 * n.
        distance]
    print(f"Good matches: {len(good)}")
    return kp1, kp2, good
```

Warping:

```
def stitch_images(img1, img2, H):
    h2, w2 = img2.shape[:2]; h1, w1 = img1.shape[:2]
    H_matrix = H.params if hasattr(H, 'params') else H
    corners_img1 = np.float32([[0,0],[0,h1],[w1,h1],[w1,0]]).
        reshape(-1,1,2);warped_corners = cv.perspectiveTransform
        (corners_img1, H_matrix);all_corners = np.vstack((
        warped_corners, np.float32([[0,0],[0,h2],[w2,h2],[w2
        ,0]]).reshape(-1,1,2)));[xmin, ymin], [xmax, ymax] = np.
        int32(all_corners.min(axis=0).ravel()), np.int32(
        all_corners.max(axis=0).ravel())
    translation = [-xmin, -ymin];H_translate = np.array([[1,0,
        translation[0]],[0,1,translation[1]],[0,0,1]]);result =
        cv.warpPerspective(img1, H_translate @ H_matrix, (xmax -
         xmin, ymax - ymin));result[translation[1]:h2+
        translation[1], translation[0]:w2+translation[0]] = img2
    return result
```

RANSAC Homography Estimation:

```
def ransac_homography(kp1, kp2, matches):
    src_pts = np.array([kp1[m.queryIdx].pt for m in
        matches], np.float32)
    dst_pts = np.array([kp2[m.trainIdx].pt for m in
        matches], np.float32)
    num, thres, iters, min_in = 4, 1.0, 200, 4
    best_H, best_inliers, best_count = None, None, 0
    np.random.seed(62)
    for _ in range(iters):
        idx = np.random.choice(len(matches), num,
            replace=False)
        src, dst = src_pts[idx], dst_pts[idx]
        tform = transform.estimate_transform('
            projective', src, dst)
        inliers = get_inliers(src_pts, dst_pts,
            tform, thres)
        if len(inliers) > best_count:
            best_H, best_count, best_inliers = tform,
                len(inliers), inliers
    if best_count >= min_in:
        src_in, dst_in = src_pts[best_inliers],
            dst_pts[best_inliers]
        best_H = transform.estimate_transform('
            projective', src_in, dst_in)
        return best_H, best_inliers
    return None, None
```